Exploring Music Structure and Clustering in Spotify Tracks Using Unsupervised Learning

Data Preparation

In [3]:
```
# Importing Libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from scipy.cluster.hierarchy import dendrogram, linkage, cut_tree
from sklearn.metrics import confusion_matrix, adjusted_rand_score
from scipy.cluster.hierarchy import dendrogram, linkage, cut_tree
from matplotlib.colors import ListedColormap
import warnings
```

In [4]:
```
# Loading Dataset
spotify_df = pd.read_csv('SpotifyFeatures.csv')
```

In [5]:
```
spotify_df.head(5)
```

Out[5]:

	genre	artist_name	track_name	track_id	popularity	acousticness	danceability	duration_ms	energy	instrur
0	Movie	Henri Salvador	C'est beau de faire un Show	0BRjO6ga9RKCKjfDqeFgWV	0	0.611	0.389	99373	0.910	
1	Movie	Martin & les fées	Perdu d'avance (par Gad Elmaleh)	0BjC1NfoEOOusryehmNudP	1	0.246	0.590	137373	0.737	
2	Movie	Joseph Williams	Don't Let Me Be Lonely Tonight	0CoSDzoNIKCRs124s9uTVy	3	0.952	0.663	170267	0.131	
3	Movie	Henri Salvador	Dis-moi Monsieur Gordon Cooper	0Gc6TVm52BwZD07Ki6tIvf	0	0.703	0.240	152427	0.326	
4	Movie	Fabien Nataf	Ouverture	0IuslXpMROHdEPvSl1fTQK	4	0.950	0.331	82625	0.225	

In [6]:
```
# Data Exploration
# Total Rows
total_rows = len(spotify_df)
print("Total number of rows:", total_rows)
```

```
Total number of rows: 232725
```

In [7]:
```
# Data Description
spotify_df.info()
spotify_df.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 232725 entries, 0 to 232724
Data columns (total 18 columns):
 #   Column            Non-Null Count   Dtype
---  ------            --------------   -----
 0   genre             232725 non-null  object
 1   artist_name       232725 non-null  object
 2   track_name        232724 non-null  object
 3   track_id          232725 non-null  object
 4   popularity        232725 non-null  int64
 5   acousticness      232725 non-null  float64
 6   danceability      232725 non-null  float64
 7   duration_ms       232725 non-null  int64
 8   energy            232725 non-null  float64
 9   instrumentalness  232725 non-null  float64
 10  key               232725 non-null  object
 11  liveness          232725 non-null  float64
 12  loudness          232725 non-null  float64
 13  mode              232725 non-null  object
 14  speechiness       232725 non-null  float64
 15  tempo             232725 non-null  float64
 16  time_signature    232725 non-null  object
 17  valence           232725 non-null  float64
dtypes: float64(9), int64(2), object(7)
memory usage: 32.0+ MB
```

Out[7]:

	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	liveness	loudness
count	232725.000000	232725.000000	232725.000000	2.327250e+05	232725.000000	232725.000000	232725.000000	232725.000000
mean	41.127502	0.368560	0.554364	2.351223e+05	0.570958	0.148301	0.215009	-9.569885
std	18.189948	0.354768	0.185608	1.189359e+05	0.263456	0.302768	0.198273	5.998204
min	0.000000	0.000000	0.056900	1.538700e+04	0.000020	0.000000	0.009670	-52.457000
25%	29.000000	0.037600	0.435000	1.828570e+05	0.385000	0.000000	0.097400	-11.771000
50%	43.000000	0.232000	0.571000	2.204270e+05	0.605000	0.000044	0.128000	-7.762000
75%	55.000000	0.722000	0.692000	2.657680e+05	0.787000	0.035800	0.264000	-5.501000
max	100.000000	0.996000	0.989000	5.552917e+06	0.999000	0.999000	1.000000	3.744000

In [8]:
```
# Unique Features
print("Number of unique genres:", spotify_df['genre'].nunique())
print("Number of unique artist names:", spotify_df['artist_name'].nunique())
print("Number of unique track IDs:", spotify_df['track_id'].nunique())
```

```
Number of unique genres: 27
Number of unique artist names: 14564
Number of unique track IDs: 176774
```

In [9]:
```
# Duplicate Track ID count
count = spotify_df['track_id'].value_counts()
num_duplicated_ids = (count > 1).sum()
print("Number of unique track ID values that are duplicated:", num_duplicated_ids)
```

```
Number of unique track ID values that are duplicated: 35124
```

In [10]:
```
# Inspect a Duplicated Track
first_duplicate_id = spotify_df[spotify_df.duplicated(subset='track_id', keep='first')]['track_id'].iloc[0]
print("First duplicated track_id:", first_duplicate_id)
spotify_df[spotify_df['track_id'] == first_duplicate_id]
```

```
First duplicated track_id: 6iOvnACn4ChlAw4lWUU4dd
```

Out[10]:

	genre	artist_name	track_name	track_id	popularity	acousticness	danceability	duration_ms	energy
257	R&B	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.7
1348	Alternative	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.7
77710	Children's Music	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.7
93651	Indie	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.7
113770	Pop	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.7

In [11]:
```
# Clean duplicated Tracks
spotify_df= spotify_df.drop_duplicates(subset=['track_id'])
count = spotify_df['track_id'].value_counts()
num_duplicated_ids = (count > 1).sum()
print("Number of unique tracks that are duplicated after cleaning:", num_duplicated_ids)
```

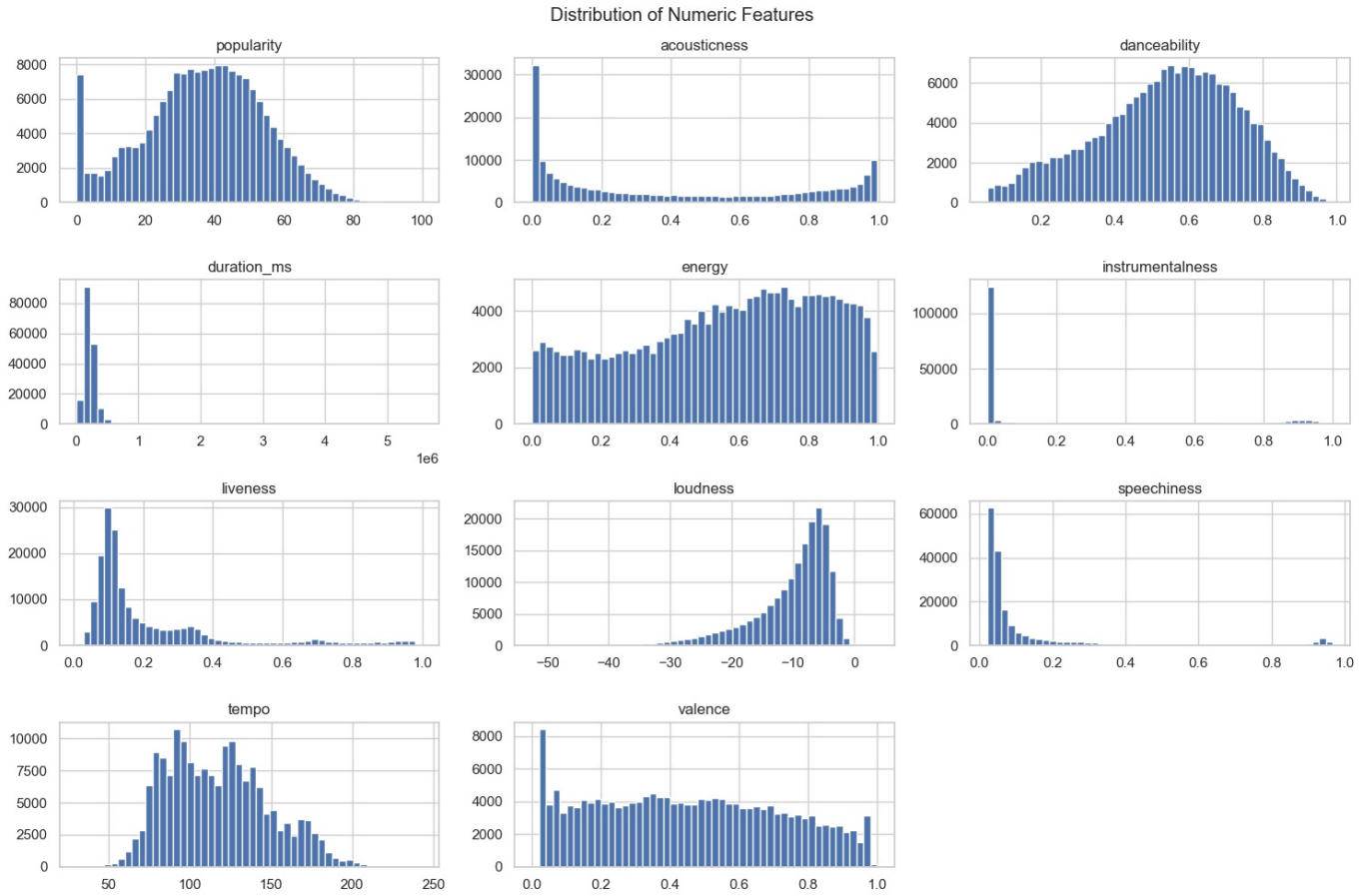
```
Number of unique tracks that are duplicated after cleaning: 0
```

In [12]:
```
# Strip Leading/Trailing Whitespace and Fix Case
spotify_df['genre'] = spotify_df['genre'].str.replace('’', "'", regex=False)
spotify_df['genre'].unique()
```

Out[12]:
```
array(['Movie', 'R&B', 'A Capella', 'Alternative', 'Country', 'Dance',
       'Electronic', 'Anime', 'Folk', 'Blues', 'Opera', 'Hip-Hop',
       "Children's Music", 'Rap', 'Indie', 'Classical', 'Pop', 'Reggae',
       'Reggaeton', 'Jazz', 'Rock', 'Ska', 'Comedy', 'Soul', 'Soundtrack',
       'World'], dtype=object)
```

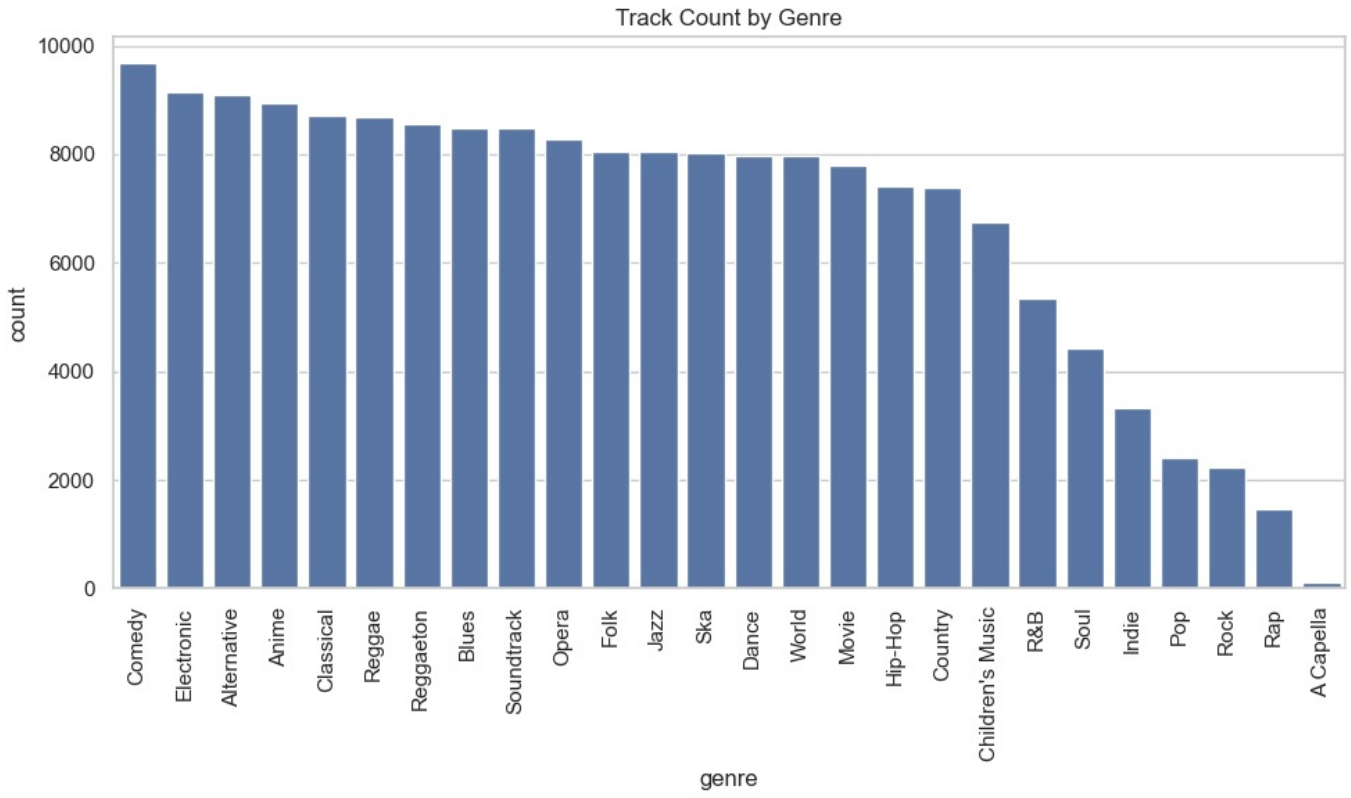
EDA

In [14]:
```
# Data Visualisation
sns.set(style='whitegrid')

# Select Numeric Columns
df_numeric = spotify_df.select_dtypes(include='number')

# Plot Histograms
df_numeric.hist(bins=50, figsize=(15, 10))
plt.suptitle('Distribution of Numeric Features')
plt.tight_layout()
plt.show()
```



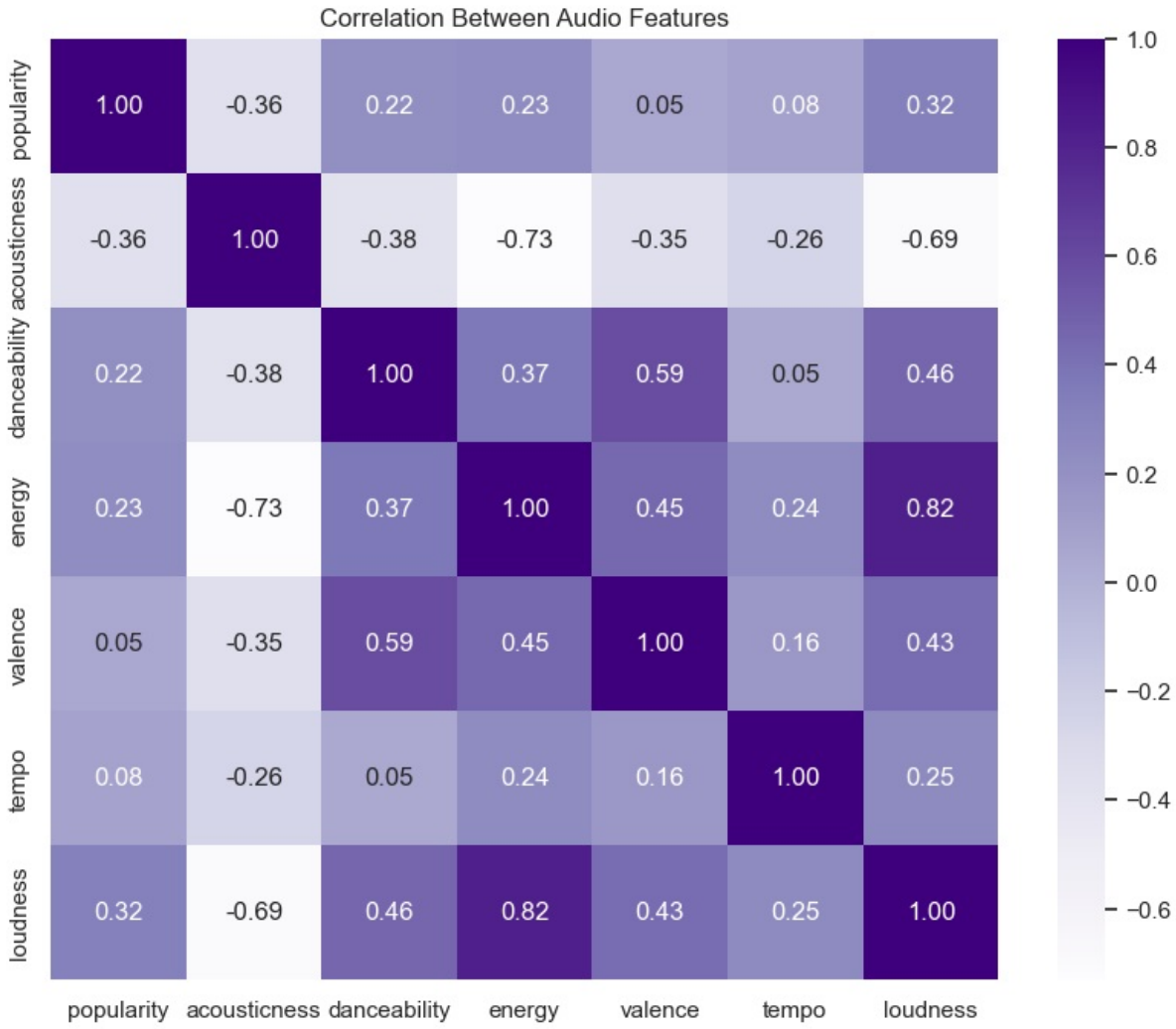
- As we can see time, instrumentalness and speechiness are highly skewed, it requires transformation before scaling it to reduce variance.

In [16]:
```
# Genre Distribution
sns.set(style="whitegrid")
plt.figure(figsize=(10, 6))
sns.countplot(x='genre', data=spotify_df, order=spotify_df['genre'].value_counts().index)
plt.title('Track Count by Genre')
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```



- So, in total there are 26 different genre and sample size of each genre is almost above 600 expet some and henxce there is no highly imbalance in the dataset.

In [18]:
```
# Correlation Heatmap
plt.figure(figsize=(10, 8))
features = ['popularity', 'acousticness', 'danceability', 'energy', 'valence', 'tempo', 'loudness']
corr = spotify_df[features].corr()
sns.heatmap(corr, annot=True, cmap='Purples', fmt=".2f")
plt.title('Correlation Between Audio Features')
plt.show()
```



- As we can see from above plot, popularity is weekly correlated with other musical features and hence it is not highlt correlated.
- While, energy and acoustiness are negatively correalted and loudness is positively correlated with energy.

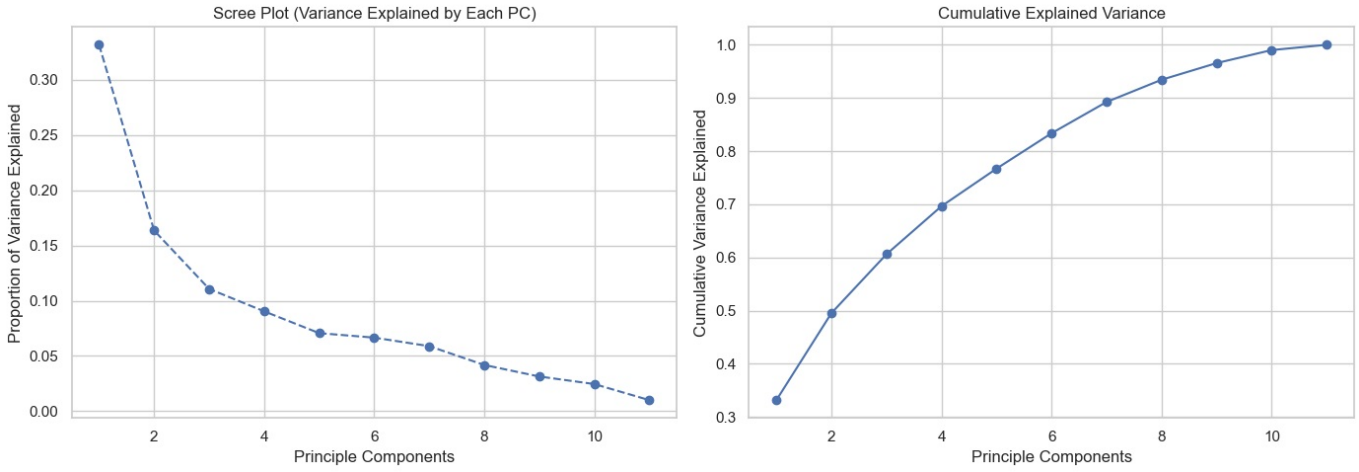
Data Transformation

In [21]:
```
# Log Transformation
spotify_df['duration_ms'] = np.log1p(spotify_df['duration_ms'])
spotify_df['instrumentalness'] = np.log1p(spotify_df['instrumentalness'])
spotify_df['speechiness'] = np.log1p(spotify_df['speechiness'])

# Encode Categorical Data
for col in ['key', 'mode', 'time_signature']:
    le = LabelEncoder()
    spotify_df[col] = le.fit_transform(spotify_df[col])

#Drop Columns
spotify_df_cleaned = spotify_df.drop(columns=['track_id', 'artist_name', 'track_name'])

features = ['popularity', 'acousticness', 'danceability', 'duration_ms', 'energy',
            'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo',
            'valence']

# Scale features
scaler = StandardScaler()
spotify_scaled = scaler.fit_transform(spotify_df_cleaned[features])

# Create DataFrame
spotify_scaled_df = pd.DataFrame(spotify_scaled, columns=features)
```

Method-1:Principal Component Analysis

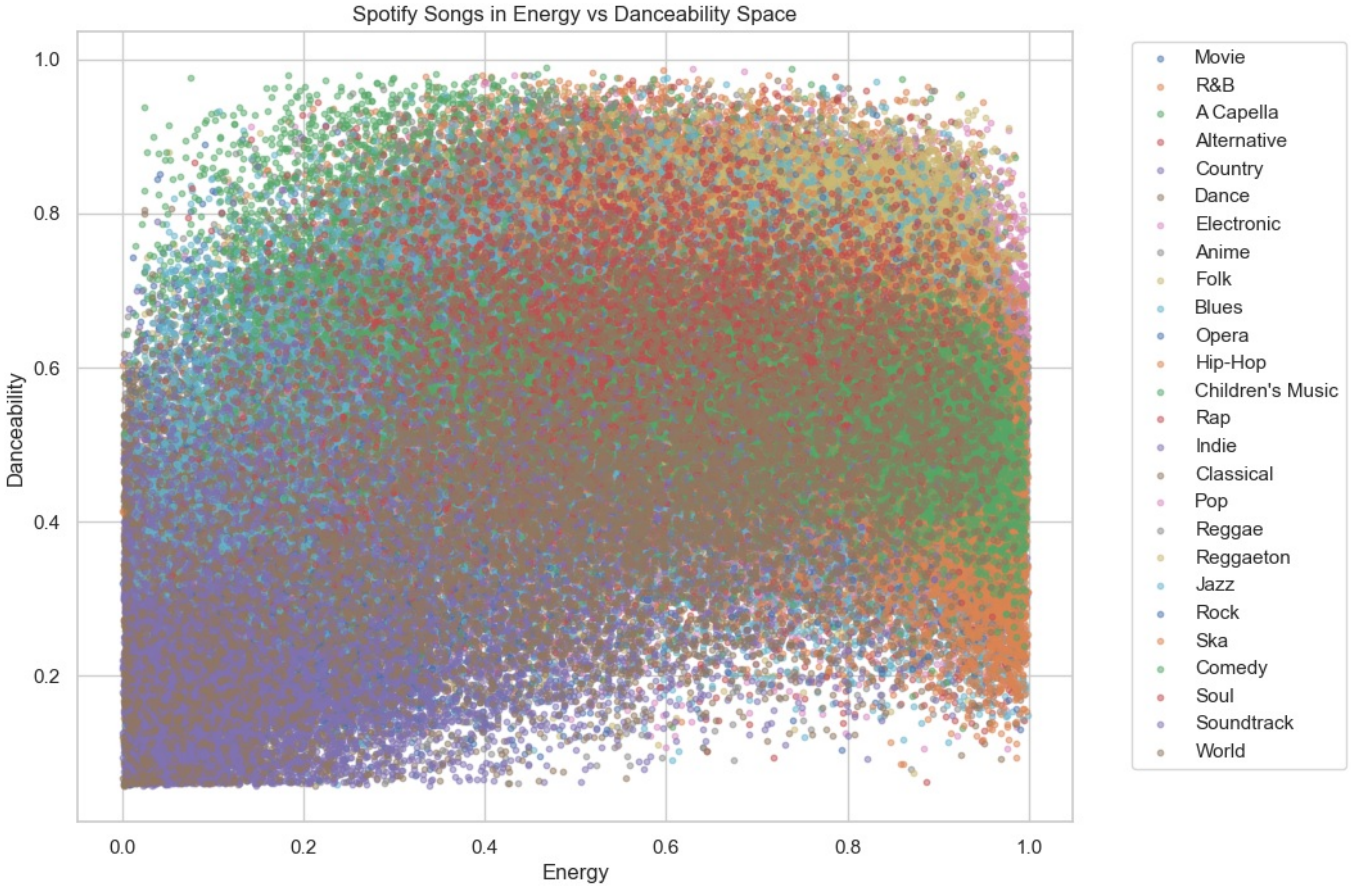
In [23]:
```
# Performing PCA
pca = PCA()
spotify_pca = pca.fit_transform(spotify_scaled_df)

# Proportion of Variance Explained by Each Component
explained_variance = pca.explained_variance_ratio_
```

```
# Cumulative Variance Explained
cumulative_variance = np.cumsum(explained_variance)

# Plots
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(range(1, len(explained_variance)+1), explained_variance, marker='o', linestyle='--')
axs[0].set_title("Scree Plot (Variance Explained by Each PC)")
axs[0].set_xlabel("Principle Components")
axs[0].set_ylabel("Proportion of Variance Explained")
axs[0].grid(True)

axs[1].plot(range(1, len(cumulative_variance)+1), cumulative_variance, marker='o', linestyle='-')
axs[1].set_title("Cumulative Explained Variance")
axs[1].set_xlabel("Principle Components")
axs[1].set_ylabel("Cumulative Variance Explained")
axs[1].grid(True)

plt.tight_layout()
plt.show()
```

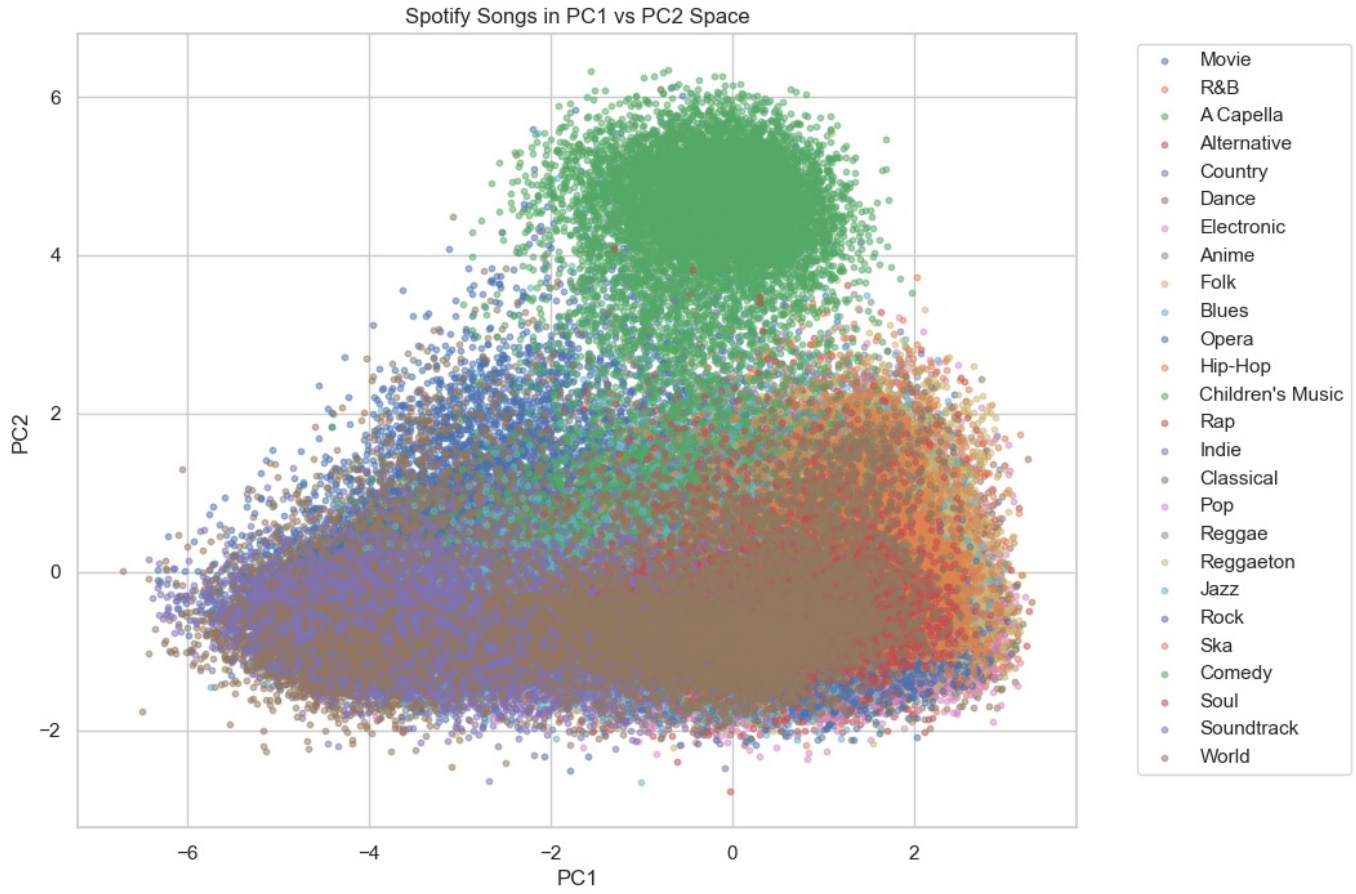


Compare the PC1-PC2 scatter plot with a scatter of two original features (for example, energy vs. danceability) and provide interpretation.

In [25]:
```
plt.figure(figsize=(10, 8))
for genre in spotify_df['genre'].unique():
    subset = spotify_df[spotify_df['genre'] == genre]
    plt.scatter(subset['energy'], subset['danceability'], label=genre, alpha=0.5, s=10)
plt.title('Spotify Songs in Energy vs Danceability Space')
plt.xlabel('Energy')
plt.ylabel('Danceability')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True)
plt.show()
```



In [26]:
```
# DataFrame for PCA results with PC1, PC2, and Genre
spotify_pca_df = pd.DataFrame(spotify_pca[:, :2], columns=['PC1', 'PC2'])
spotify_pca_df['genre'] = spotify_df['genre'].values  # Include genre for color coding

# Plot PC1 vs PC2 with Genre Colors
plt.figure(figsize=(10, 8))
for genre in spotify_pca_df['genre'].unique():
    subset = spotify_pca_df[spotify_pca_df['genre'] == genre]
    plt.scatter(subset['PC1'], subset['PC2'], label=genre, alpha=0.5, s=10)
plt.title('Spotify Songs in PC1 vs PC2 Space')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True)
plt.show()
```



Spotify Songs in PC1 vs PC2 Space When we mapped Spotify's entire music catalog into PCA space, something amazing happened. Suddenly, thousands of songs—each with their unique rhythm and energy—compressed into a neat, two-dimensional scatterplot. Here, each point represents a song, blending a cocktail of features like energy, danceability, valence, and more.

What's striking is how some genres (shown by clusters of similar colors) seem to gravitate toward each other. It's as if PCA exposes invisible musical friendships, revealing which genres share common audio traits. Compared to the messy overlaps of the original features, this PCA view gives a much clearer picture of Spotify's musical landscape.

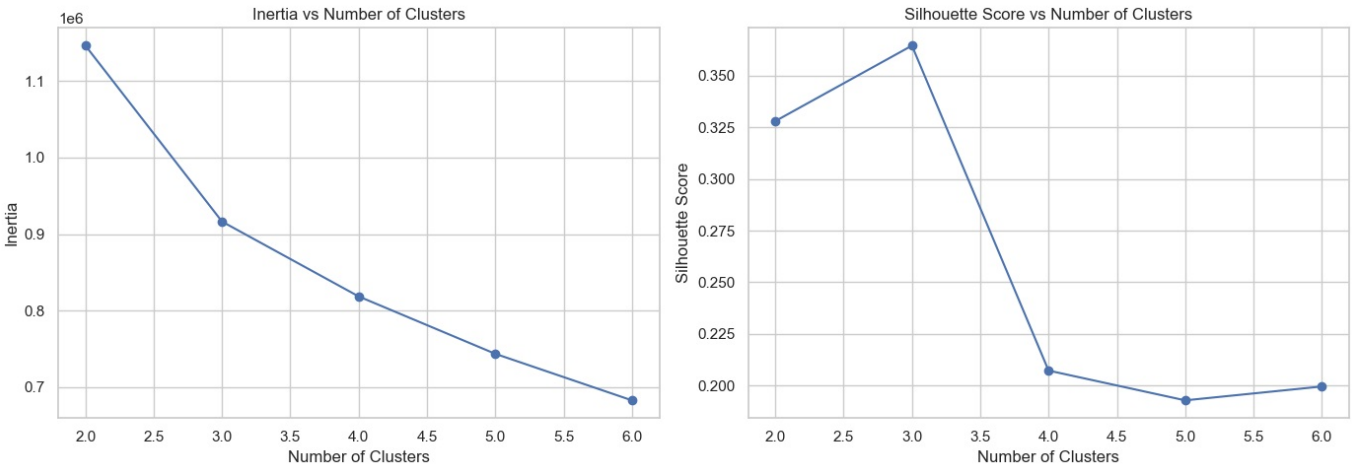
Spotify Songs in Energy vs Danceability Space Now, when we look at a plot based solely on energy and danceability, the story changes. Sure, these two features tell us a lot about mood and movement, but the overall spread is much more uniform. Genres blend together here, making it harder to distinguish one from another.

It's a reminder that while these features are important, they're just two threads in the rich tapestry of musical characteristics. Many nuances—like tempo, loudness, or valence—get lost when we focus on only two features.

Method-2: K-means Clustering

In [29]:
```
# Performing K-Means
inertias = []
silhouette_scores = []
k_range = range(2, 7)

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42, max_iter=100, n_init=5)
    kmeans.fit(spotify_pca[:, :6])
    inertias.append(kmeans.inertia_)
    silhouette_scores.append(silhouette_score(spotify_pca[:, :6], kmeans.labels_))

# Plots
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(k_range, inertias, marker='o')
axs[0].set_title('Inertia vs Number of Clusters')
axs[0].set_xlabel('Number of Clusters')
axs[0].set_ylabel('Inertia')
axs[0].grid(True)

axs[1].plot(k_range, silhouette_scores, marker='o')
axs[1].set_title('Silhouette Score vs Number of Clusters')
axs[1].set_xlabel('Number of Clusters')
axs[1].set_ylabel('Silhouette Score')
axs[1].grid(True)

plt.tight_layout()
```

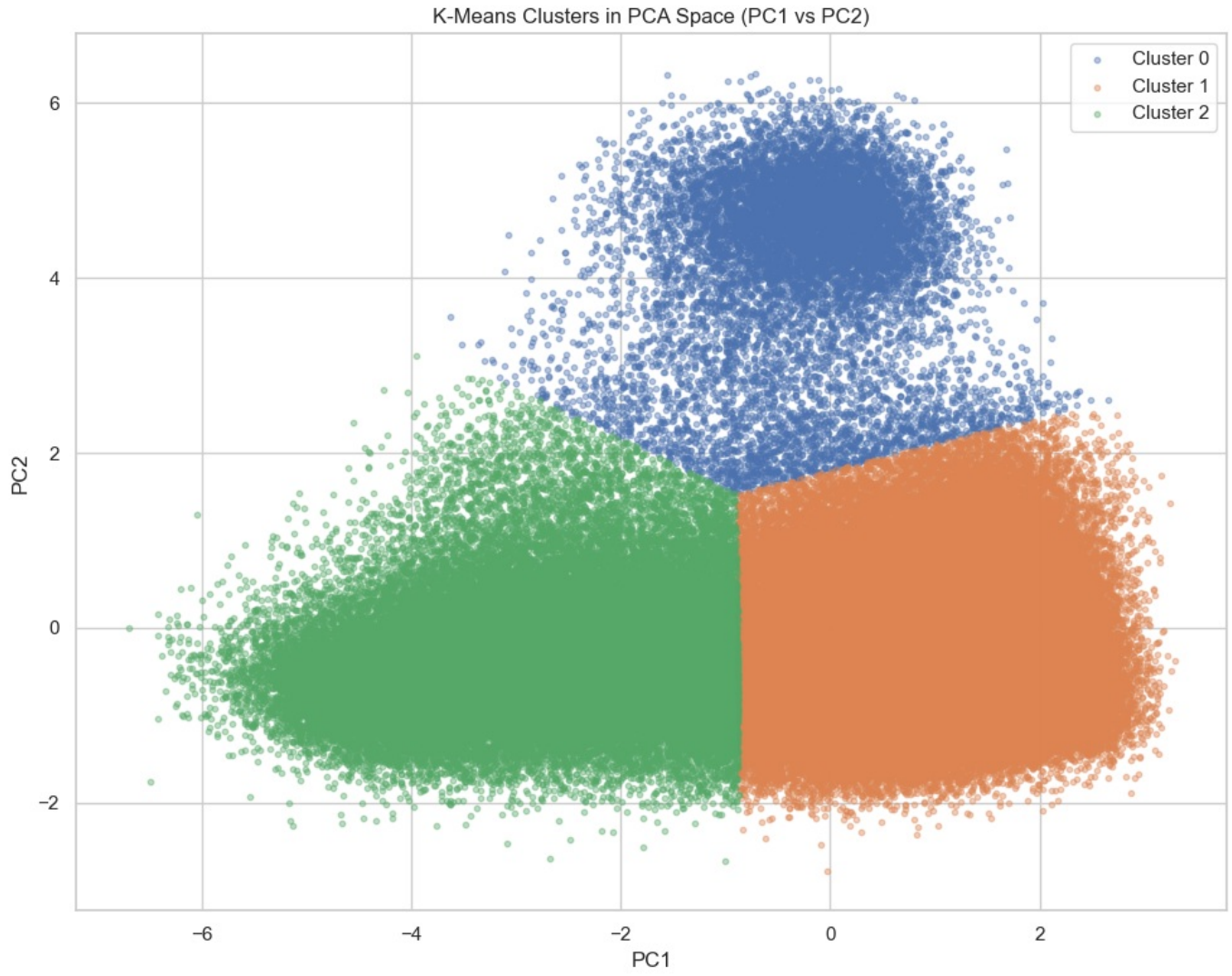
```
plt.show()
```



These two plots help decide the best number of clusters for KMeans on Spotify track data. The left plot shows inertia, which measures how tightly the data points fit within their clusters, lower is better. There's a noticeable drop until 3 clusters, after which the improvement slows, suggesting 3 might be a good choice. The right plot shows the silhouette score, which measures how well-separated the clusters are. The highest score is also at 3 clusters, confirming it's likely the optimal number for this dataset.

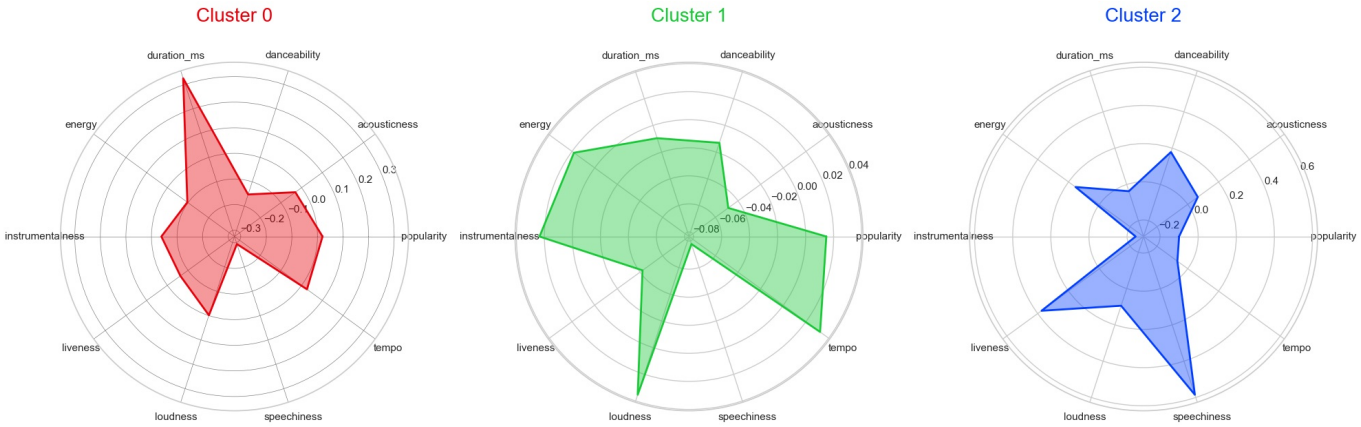
In [31]:
```
# Apply KMeans on PC1 and PC2
kmeans = KMeans(n_clusters=3, random_state=42)
spotify_df['cluster'] = kmeans.fit_predict(spotify_pca[:, [0, 1]])
```

In [32]:
```
plt.figure(figsize=(10, 8))

for cluster in range(3):
    idx = spotify_df['cluster'] == cluster
    plt.scatter(spotify_pca[idx, 0], spotify_pca[idx, 1], label=f'Cluster {cluster}', s=10, alpha=0.4)

plt.title('K-Means Clusters in PCA Space (PC1 vs PC2)')
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



In [33]:
```
colors = ['#e8000b', '#1ac938', '#023eff']

# Standardize the data (excluding the cluster column)
#scaler = StandardScaler()
df_standardized = spotify_df.copy()

# Create a new dataframe with standardized values and add the cluster column back
df_standardized = pd.DataFrame(spotify_scaled_df, columns=spotify_scaled_df.columns[:-1], index=spotify_scaled_
df_standardized['cluster'] = spotify_df['cluster']
# Calculate the centroids of each cluster
cluster_centroids = df_standardized.groupby('cluster').mean()

# Function to create a radar chart
def create_radar_chart(ax, angles, data, color, cluster):
    # Plot the data and fill the area
    ax.fill(angles, data, color=color, alpha=0.4)
    ax.plot(angles, data, color=color, linewidth=2, linestyle='solid')

    # Add a title
    ax.set_title(f'Cluster {cluster}', size=20, color=color, y=1.1)

# Set data
labels=np.array(cluster_centroids.columns)
num_vars = len(labels)

# Compute angle of each axis
angles = np.linspace(0, 2 * np.pi, num_vars, endpoint=False).tolist()

# The plot is circular, so we need to "complete the loop" and append the start to the end
labels = np.concatenate((labels, [labels[0]]))
angles += angles[:1]

# Initialize the figure
fig, ax = plt.subplots(figsize=(20, 10), subplot_kw=dict(polar=True), nrows=1, ncols=3)

# Create radar chart for each cluster
for i, color in enumerate(colors):
    data = cluster_centroids.loc[i].tolist()
    data += data[:1]  # Complete the loop
    create_radar_chart(ax[i], angles, data, color, i)
```

```
# Add input data
ax[0].set_xticks(angles[:-1])
ax[0].set_xticklabels(labels[:-1])

ax[1].set_xticks(angles[:-1])
ax[1].set_xticklabels(labels[:-1])

ax[2].set_xticks(angles[:-1])
ax[2].set_xticklabels(labels[:-1])

# Add a grid
ax[0].grid(color='grey', linewidth=0.5)

# Display the plot
plt.tight_layout()
plt.show()
```



These radar charts show the characteristics of each of the three clusters found in the Spotify track dataset using KMeans.

- **Cluster 0 (Red)**: Songs in this category are typically lengthier, more instrumental, and moderately live. Their low speechiness and loudness scores imply that they might be ambient or instrumental tracks with a lower vocal intensity.
- **Cluster 1 (Green)**: Although the qualities of this cluster are comparatively balanced, they tend to be louder, more instrumental, and more energetic. There is also a minor increase in popularity and tempo. These could be well-produced, lively, or high-energy songs that are also perhaps more well-liked in general.
- **Cluster 2 (Blue)**: Songs in this cluster are very danceable and loud and talkative. They are less instrumental and shorter. These are probably upbeat, vocally heavy songs like pop, dance, or rap.

project_population_binary

June 13, 2025

0.1 BINARY CLASSIFICATION OF TRACK POPULARITY

0.1.1 Importing imp. libraries

[2]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix
from sklearn.tree import DecisionTreeClassifier, export_text, plot_tree
from sklearn.preprocessing import StandardScaler
import xgboost as xgb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.metrics import AUC
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import (
    precision_score, recall_score, f1_score,
    roc_auc_score, confusion_matrix, classification_report
)
```

[3]:
```
# Loading Dataset
spotify_df = pd.read_csv(r"C:\Users\alekh\Downloads\SpotifyFeatures.csv")
```

[4]:
```
# Columns
spotify_df.columns
```

[4]:
```
Index(['genre', 'artist_name', 'track_name', 'track_id', 'popularity',
       'acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'key', 'liveness', 'loudness', 'mode',
       'speechiness', 'tempo', 'time_signature', 'valence'],
```

dtype='object')

0.1.2 Data Cleaning

[4]:
```
# Duplicate Track ID count
count = spotify_df['track_id'].value_counts()
num_duplicated_ids = (count > 1).sum()
print("Number of unique track ID values that are duplicated:",␣
 ↪num_duplicated_ids)
```

```
Number of unique track ID values that are duplicated: 35124
```

[5]:
```
# Inspect a Duplicated Track
first_duplicate_id = spotify_df[spotify_df.duplicated(subset='track_id',␣
 ↪keep='first')]['track_id'].iloc[0]
print("First duplicated track_id:", first_duplicate_id)
spotify_df[spotify_df['track_id'] == first_duplicate_id]
```

```
First duplicated track_id: 6iOvnACn4ChlAw4lWUU4dd
```

[5]:

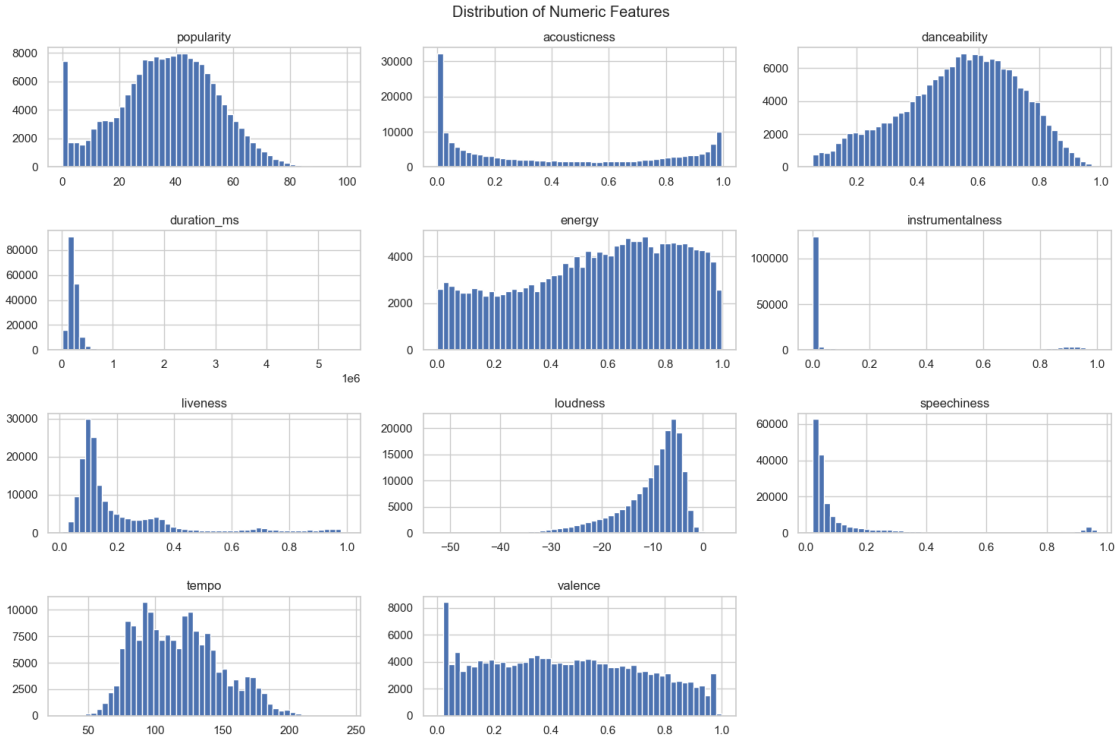
	genre	artist_name	track_name	track_id	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	time_signature	valence
257	R&B	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	-2.474	Major	0.0579	169.944	4/4	0.7
1348	Alternative	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	-2.474	Major	0.0579	169.944	4/4	0.7
77710	Children's Music	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	-2.474	Major	0.0579	169.944	4/4	0.7
93651	Indie	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	-2.474	Major	0.0579	169.944	4/4	0.7
113770	Pop	Doja Cat	Go To Town	6iOvnACn4ChlAw4lWUU4dd	64	0.0716	0.71	217813	0.71	0.000001	C	0.206	-2.474	Major	0.0579	169.944	4/4	0.7

[6]:
```
# Clean Duplicated Tracks
spotify_df= spotify_df.drop_duplicates(subset=['track_id'])
count = spotify_df['track_id'].value_counts()
num_duplicated_ids = (count > 1).sum()
print("Number of unique tracks that are duplicated after cleaning:",␣
 ↪num_duplicated_ids)
```

```
Number of unique tracks that are duplicated after cleaning: 0
```

0.1.3 EDA

[7]:
```
# Data Visualisation
sns.set(style='whitegrid')

# Select Numeric Columns
df_numeric = spotify_df.select_dtypes(include='number')

# Plot Histograms
df_numeric.hist(bins=50, figsize=(15, 10))
plt.suptitle('Distribution of Numeric Features')
plt.tight_layout()
plt.show()
```



We removed out songs with zero popularity as it does not signify any measurable value from

audience and categorized the remaining tracks into two classes:

- **Low popularity** (1–40)
- **High popularity** (41–80)

[7]:
```
spotify_df = spotify_df[spotify_df['popularity'] != 0]

#Popularity col. classification
def popularity_class(popularity):
    if 1 <= popularity <= 40:
        return 'low'
    elif 41 <= popularity <= 80:
        return 'high'
    else:
        return None  #values outside 1-80

spotify_df['popularity_class'] = spotify_df['popularity'].
 ↪apply(popularity_class)

#Drop rows with popularity outside 1-80
spotify_df = spotify_df[spotify_df['popularity_class'].notnull()]
```

0.1.4 Feature Selection & Data Preparation

- We selected 10 audio features from Spotify tracks to predict song popularity.
- The dataset was split into training and testing sets (80/20 split), and all features were standardized to ensure equal contribution during model training.

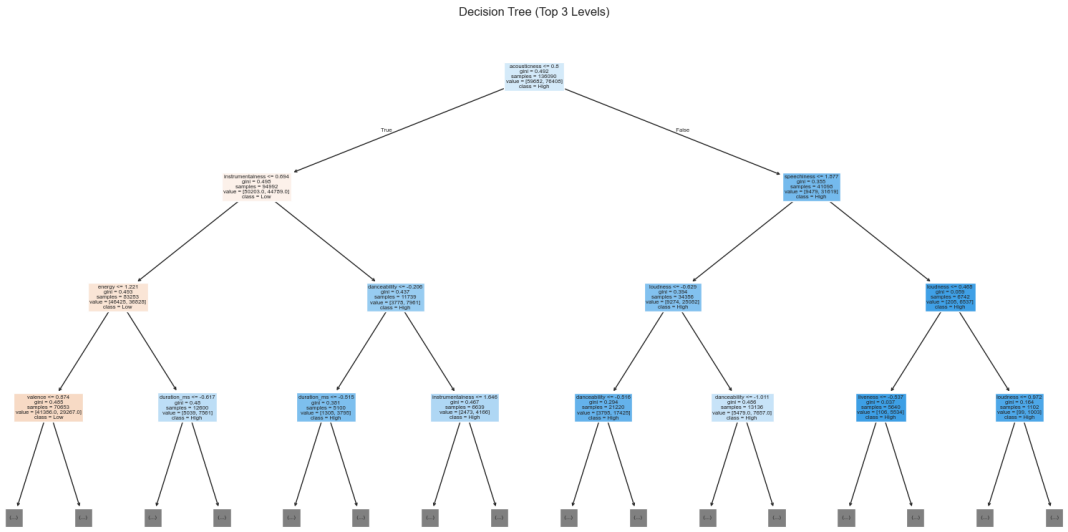
[8]:
```
features = [
    'acousticness', 'danceability', 'duration_ms', 'energy',
    'instrumentalness', 'liveness', 'loudness', 'speechiness',
    'tempo', 'valence'
]


X = spotify_df[features]
y = spotify_df['popularity_class']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("Training set size:", X_train_scaled.shape)
print("Test set size:", X_test_scaled.shape)
```

```
Training set size: (136090, 10)
Test set size: (34023, 10)
```

0.1.5 Unpruned Decision Tree

[11]:
```
tree_model = DecisionTreeClassifier(random_state=42)
tree_model.fit(X_train_scaled, y_train)

# Predict on test set
y_pred_dt = tree_model.predict(X_test_scaled)

# Evaluate
print("Unpruned Decision Tree Accuracy:", accuracy_score(y_test, y_pred_dt))
print(classification_report(y_test, y_pred_dt))

# View of tree depth=3
plt.figure(figsize=(20,10))
plot_tree(tree_model, feature_names=X_train.columns,
          class_names=['Low', 'High'],
          filled=True, max_depth=3)
plt.title('Decision Tree (Top 3 Levels)')
plt.show()

# Printing only first 1500
tree_summary = export_text(tree_model, feature_names=list(X_train.columns))
print(tree_summary[:1500])
```

```
Unpruned Decision Tree Accuracy: 0.6209328983334803
              precision    recall  f1-score   support

        high       0.56      0.58      0.57     14816
         low       0.67      0.65      0.66     19207

    accuracy                           0.62     34023
   macro avg       0.62      0.62      0.62     34023
weighted avg       0.62      0.62      0.62     34023
```



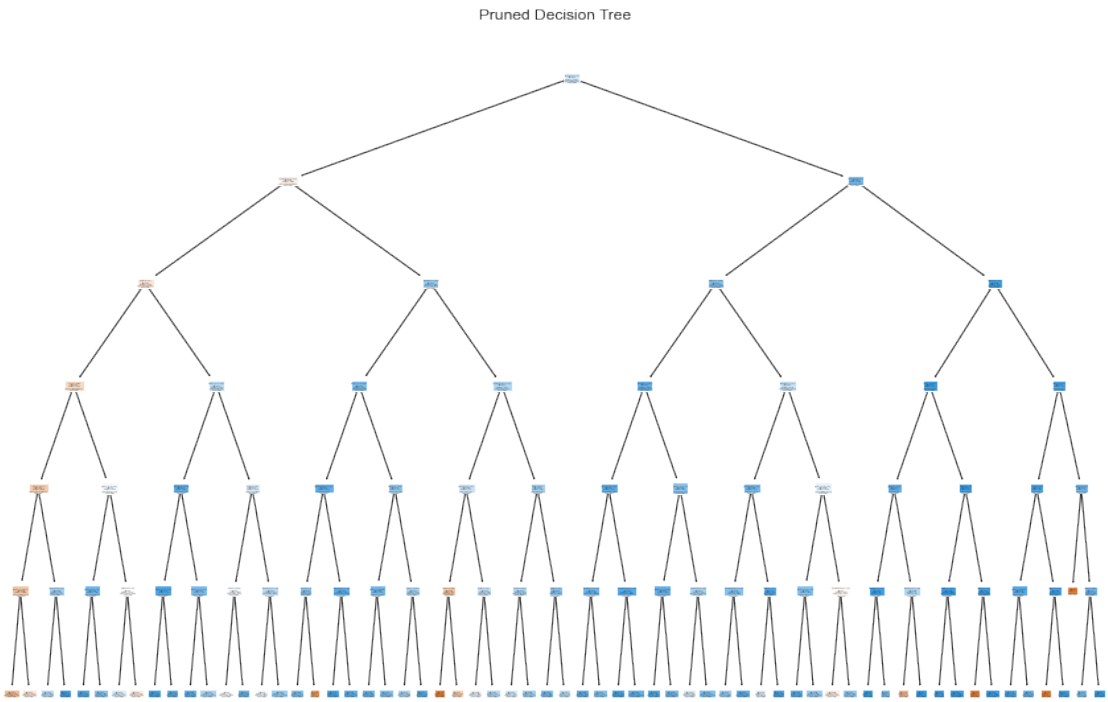

```
|--- acousticness <= 0.80
|   |--- instrumentalness <= 0.69
|   |   |--- energy <= 1.22
|   |   |   |--- valence <= 0.87
|   |   |   |   |--- liveness <= 2.12
|   |   |   |   |   |--- duration_ms <= 0.21
|   |   |   |   |   |   |--- loudness <= -0.07
|   |   |   |   |   |   |   |--- duration_ms <= -0.99
|   |   |   |   |   |   |   |   |--- liveness <= -0.40
|   |   |   |   |   |   |   |   |   |--- valence <= 0.51
|   |   |   |   |   |   |   |   |   |   |--- tempo <= 1.30
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 10
|   |   |   |   |   |   |   |   |   |   |--- tempo >  1.30
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 4
|   |   |   |   |   |   |   |   |   |--- valence >  0.51
|   |   |   |   |   |   |   |   |   |   |--- tempo <= 1.78
|   |   |   |   |   |   |   |   |   |   |   |--- class: low
|   |   |   |   |   |   |   |   |   |   |--- tempo >  1.78
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 2
|   |   |   |   |   |   |   |   |--- liveness >  -0.40
|   |   |   |   |   |   |   |   |   |--- speechiness <= -0.48
|   |   |   |   |   |   |   |   |   |   |--- duration_ms <= -1.10
|   |   |   |   |   |   |   |   |   |   |   |--- class: high
|   |   |   |   |   |   |   |   |   |   |--- duration_ms >  -1.10
|   |   |   |   |   |   |   |   |   |   |   |--- class: low
|   |   |   |   |   |   |   |   |   |--- speechiness >  -0.48
|   |   |   |   |   |   |   |   |   |   |--- speechiness <= 2.35
|
```

[ ]:
```
# Tuning tree size using cross-validation
param_grid = {'max_leaf_nodes': range(2, 20)}
grid_search_dt = GridSearchCV(DecisionTreeClassifier(random_state=42),
                              param_grid=param_grid,
                              cv=5, scoring='accuracy')

grid_search_dt.fit(X_train_scaled, y_train)
pruned_tree = grid_search_dt.best_estimator_
y_pred_pruned = pruned_tree.predict(X_test_scaled)

# Evaluate
print("Pruned Decision Tree Accuracy:", accuracy_score(y_test, y_pred_pruned))
print(classification_report(y_test, y_pred_pruned))

# Plot pruned tree
plt.figure(figsize=(15,10))
plot_tree(pruned_tree, feature_names=X_train.columns,
          class_names=['Low', 'High'],
          filled=True)
plt.title('Pruned Decision Tree')
plt.show()

print(f"Best Tree Size (max_leaf_nodes): {grid_search_dt.
 ↪best_params_['max_leaf_nodes']}")
```

[50]:
```
# Tuning tree size using cross-validation
param_grid = {
    'max_depth': [3, 4, 5, 6, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_leaf_nodes': [None, 5, 10, 15],
    'criterion': ['gini', 'entropy']
}

grid_search_dt = GridSearchCV(DecisionTreeClassifier(random_state=42),
                              param_grid=param_grid,
                              cv=5, scoring='accuracy')

grid_search_dt.fit(X_train_scaled, y_train)
pruned_tree = grid_search_dt.best_estimator_
y_pred_pruned = pruned_tree.predict(X_test_scaled)

# Evaluate
print("Pruned Decision Tree Accuracy:", accuracy_score(y_test, y_pred_pruned))
print(classification_report(y_test, y_pred_pruned))

# Plot pruned tree
```

```
plt.figure(figsize=(15,10))
plot_tree(pruned_tree, feature_names=X_train.columns,
          class_names=['Low', 'High'],
          filled=True)
plt.title('Pruned Decision Tree')
plt.show()
```

```
Pruned Decision Tree Accuracy: 0.6708109220233371
              precision    recall  f1-score   support

        high       0.61      0.69      0.64     14816
         low       0.73      0.66      0.69     19207

    accuracy                           0.67     34023
   macro avg       0.67      0.67      0.67     34023
weighted avg       0.68      0.67      0.67     34023
```




```
Best Tree Size (max_leaf_nodes): None
```

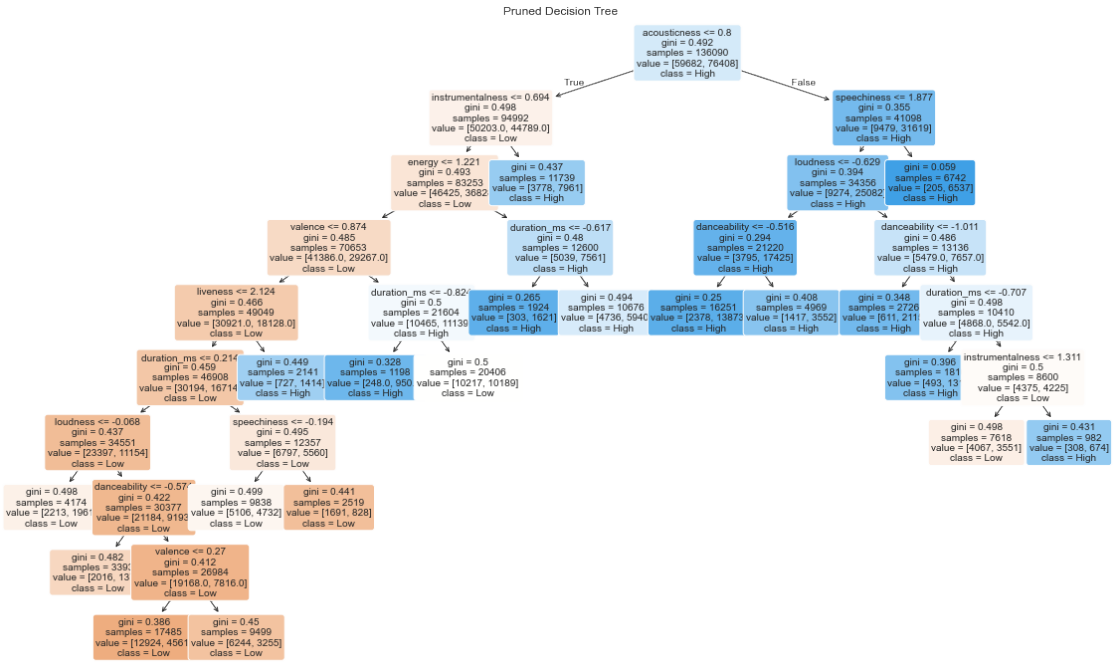
[51]:
```
print("Best Parameters Found:")
print(grid_search_dt.best_params_)
```

```
Best Parameters Found:
```

```
{'criterion': 'gini', 'max_depth': 6, 'max_leaf_nodes': None,
'min_samples_leaf': 1, 'min_samples_split': 10}
```

0.1.6 Pruned Decision Tree

[35]:
```
plt.figure(figsize=(20, 12))
plot_tree(
    pruned_tree,
    feature_names=X_train.columns,
    class_names=['Low', 'High'],
    filled=True,
    rounded=True,
    fontsize=10
)

plt.title('Pruned Decision Tree')
plt.savefig('Pruned_Decision_Tree_HighRes.png', dpi=600, bbox_inches='tight')
plt.show()
```

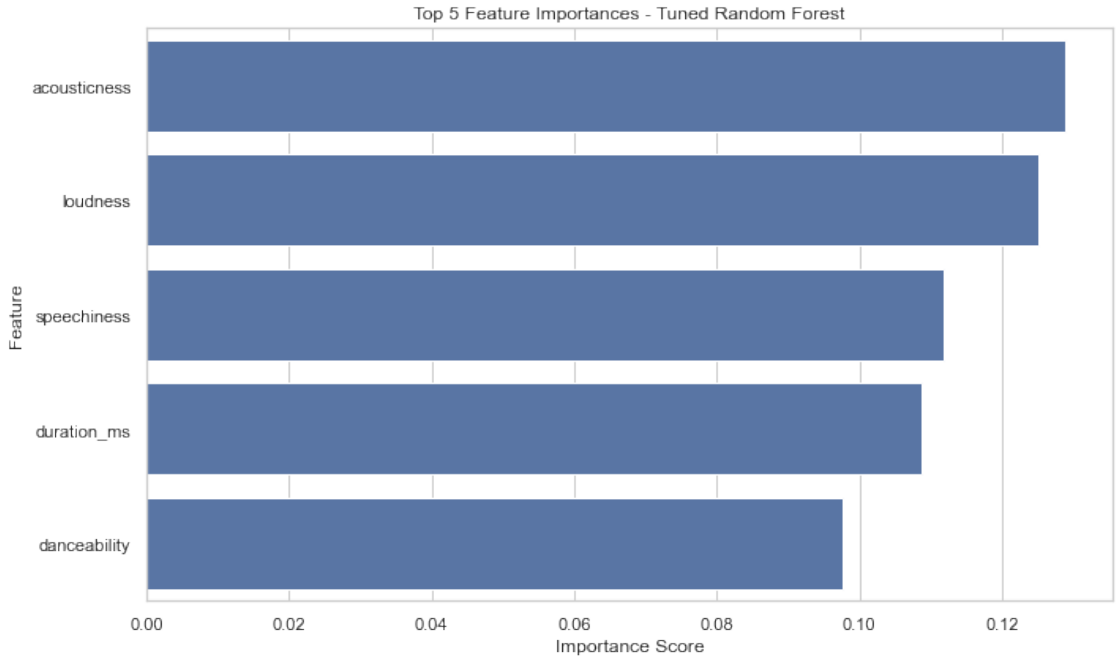


The above plot(tree) consists of the Best params for Decision tree and consists of the following: - criterion is gini - the maximum depth of the tree is 6 - the maximum number of leaf nodes in the tree are 19 - the minimum number of samples required to be at a leaf node is 1 - the minimum number of samples required to split an internal node are 10

0.1.7 Random Forest

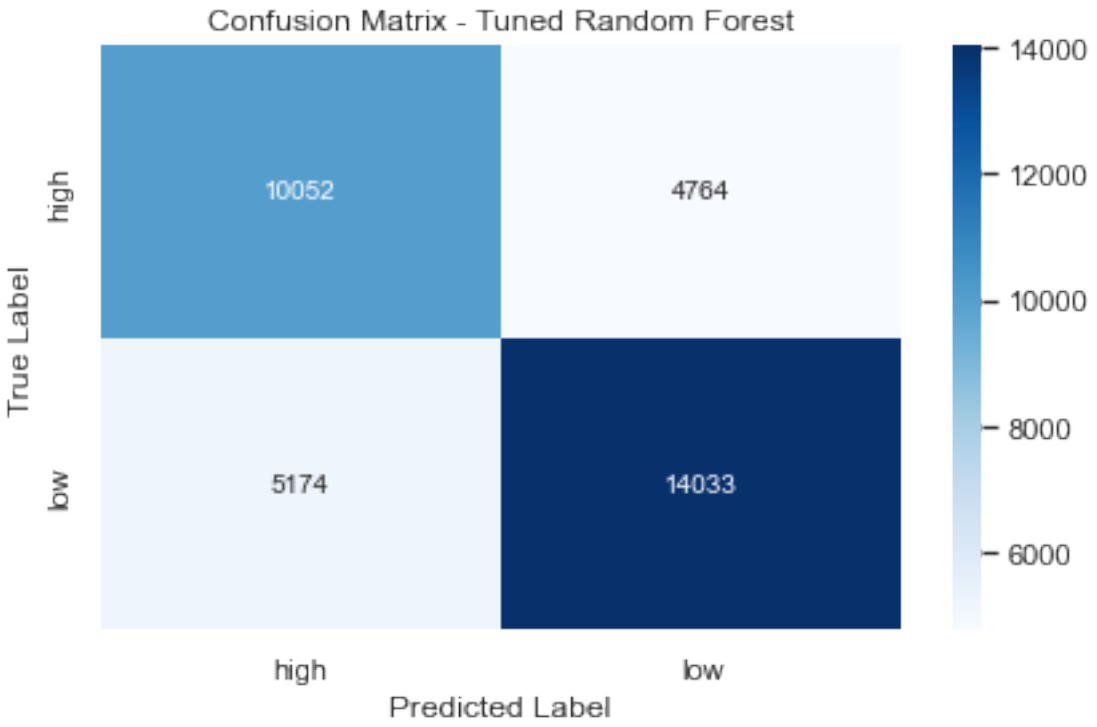
[52]:
```
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

rf_grid = GridSearchCV(RandomForestClassifier(random_state=42),
                       param_grid, cv=5, scoring='accuracy',
                       n_jobs=-1, verbose=1)

rf_grid.fit(X_train_scaled, y_train)

best_rf = rf_grid.best_estimator_
y_pred_rf = best_rf.predict(X_test_scaled)

print("Tuned Random Forest Accuracy:", accuracy_score(y_test, y_pred_rf))
print(classification_report(y_test, y_pred_rf))
print("Best Parameters:", rf_grid.best_params_)

importances = pd.Series(best_rf.feature_importances_, index=X_train.columns).
 ↪sort_values(ascending=False)
top5_importances = importances.head(5)

plt.figure(figsize=(10,6))
sns.barplot(x=top5_importances, y=top5_importances.index)
plt.title('Top 5 Feature Importances - Tuned Random Forest')
plt.xlabel('Importance Score')
plt.ylabel('Feature')
plt.tight_layout()
plt.show()
```

```
Fitting 5 folds for each of 54 candidates, totalling 270 fits
Tuned Random Forest Accuracy: 0.7079034770596361
              precision    recall  f1-score   support

        high       0.66      0.68      0.67     14816
         low       0.75      0.73      0.74     19207

    accuracy                           0.71     34023
   macro avg       0.70      0.70      0.70     34023
weighted avg       0.71      0.71      0.71     34023

Best Parameters: {'max_depth': 20, 'min_samples_leaf': 2, 'min_samples_split':
5, 'n_estimators': 200}
```



The best parameters for random forest are: - the number of trees are 200 - maximum depth of each tree is 20 - The minimum number of samples required to be at a leaf node are 2 - the minimum number of samples required to split an internal node are 5

[ ]:
```
cm = confusion_matrix(y_test, y_pred_rf, labels=['high', 'low'])

# Plot confusion matrix heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['high', 'low'], yticklabels=['high', 'low'])
plt.title('Confusion Matrix - Tuned Random Forest')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.tight_layout()
plt.show()
```



The model correctly predicted over 10,000 high popularity songs and 14,000 low popularity songs. It misclassified 4,764 high and 5,174 low popularity songs, indicating fairly balanced but slightly better performance on the low class.

[ ]:
```
print("Summary of Model Accuracies:")
print(f"- Unpruned Decision Tree Accuracy: {accuracy_score(y_test, y_pred_dt):.
  ↪4f}")
print(f"- Pruned Decision Tree Accuracy: {accuracy_score(y_test, y_pred_pruned):
  ↪.4f}")
print(f"- Random Forest Accuracy: {accuracy_score(y_test, y_pred_rf):.4f}")

if accuracy_score(y_test, y_pred_rf) > accuracy_score(y_test, y_pred_pruned):
    print("\nRandom Forest outperformed both Decision Tree versions.")
else:
    print("\nPruned Decision Tree performed competitively with Random Forest.")
```

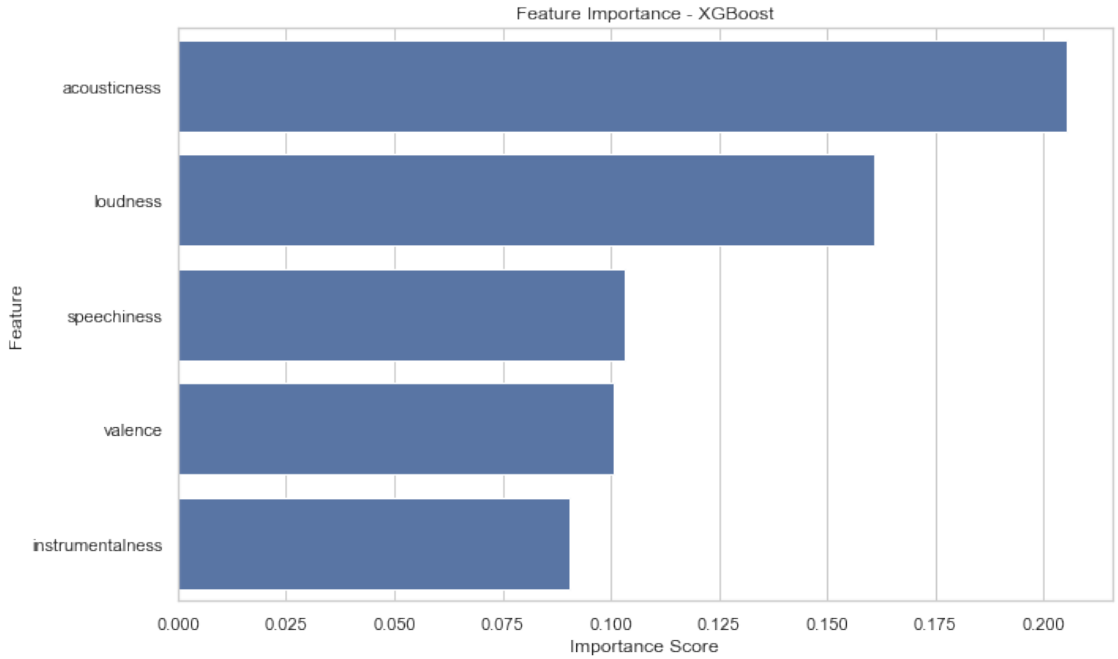
```
Summary of Model Accuracies:
- Unpruned Decision Tree Accuracy: 0.6209
- Pruned Decision Tree Accuracy: 0.6708
- Random Forest Accuracy: 0.7079

Random Forest outperformed both Decision Tree versions.
```

0.1.8 XG Boost

[ ]:
```
le = LabelEncoder()
y_train_encoded = le.fit_transform(y_train)
y_test_encoded = le.transform(y_test)


xgb_model = xgb.XGBClassifier(n_estimators=200, max_depth=4, learning_rate=0.1,␣
 ↪use_label_encoder=False, eval_metric='logloss', random_state=42)
xgb_model.fit(X_train_scaled, y_train_encoded)

y_pred_xgb = xgb_model.predict(X_test_scaled)

print("XGBoost Accuracy:", accuracy_score(y_test_encoded, y_pred_xgb))
print(classification_report(y_test_encoded, y_pred_xgb, target_names=le.
 ↪classes_))

#Feature Importance Plot
xgb_importances = pd.Series(xgb_model.feature_importances_, index=X_train.
 ↪columns).sort_values(ascending=False)
top5_xgb_importances = xgb_importances.head(5)

plt.figure(figsize=(10,6))
sns.barplot(x=top5_xgb_importances, y=top5_xgb_importances.index)
plt.title('Feature Importance - XGBoost')
plt.xlabel('Importance Score')
plt.ylabel('Feature')
plt.tight_layout()
plt.show()
```

```
C:\Users\Ankit P Bisleri\AppData\Local\Programs\Python\Python310\lib\site-
packages\xgboost\training.py:183: UserWarning: [18:29:20] WARNING: C:\actions-
runner\_work\xgboost\xgboost\src\learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
```

```
XGBoost Accuracy: 0.6979690209564119
              precision    recall  f1-score   support

        high       0.65      0.67      0.66     14816
         low       0.74      0.72      0.73     19207

    accuracy                           0.70     34023
   macro avg       0.69      0.70      0.69     34023
weighted avg       0.70      0.70      0.70     34023
```



[49]:
```
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [3, 4, 5],
    'learning_rate': [0.05, 0.1, 0.2],
    'subsample': [0.8, 1]
}

xgb_clf = xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss',␣
 ↪random_state=42)

# Grid Search
grid_search = GridSearchCV(estimator=xgb_clf, param_grid=param_grid,
                           scoring='accuracy', cv=5, verbose=1, n_jobs=-1)

grid_search.fit(X_train_scaled, y_train_encoded)

# Best model
print("Best parameters found:", grid_search.best_params_)
print("Best cross-validation score:", grid_search.best_score_)

# Evaluate best model
best_model = grid_search.best_estimator_
y_pred_best = best_model.predict(X_test_scaled)
print("Test Accuracy:", accuracy_score(y_test_encoded, y_pred_best))
```

```
print(classification_report(y_test_encoded, y_pred_best, target_names=le.
 ↪classes_))
```

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits

C:\Users\Ankit P Bisleri\AppData\Local\Programs\Python\Python310\lib\site-
packages\xgboost\training.py:183: UserWarning: [18:41:18] WARNING: C:\actions-
runner\_work\xgboost\xgboost\src\learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)

Best parameters found: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators':
200, 'subsample': 0.8}
Best cross-validation score: 0.701212432948784
Test Accuracy: 0.7021426681950446
              precision    recall  f1-score   support

        high       0.65      0.68      0.66     14816
         low       0.74      0.72      0.73     19207

    accuracy                           0.70     34023
   macro avg       0.70      0.70      0.70     34023
weighted avg       0.70      0.70      0.70     34023
```

The best parameters for XGB Boost are:

- number of trees are 200
- learning rate is 0.1
- maximum depth of each tree is 5
- subsample(fraction of the training data used for each tree) is 0.8

In binary popularity classification, model performance was influenced by both feature importance and model complexity.Random forest achieved the highest accuracy(71%) by capturing features like acousticness and loudness but required the longest training time.In contrast XG Boost provided slightly lower accuracy(69%) but trained in just 15 seconds making it a stong choice. Simpler models like decision trees performed moderately well showing imporvemnets in recall after tuning.

Multi_class_classification_code

June 13, 2025

[105]:
```
# Importing Libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier,␣
 ↪GradientBoostingClassifier, RandomForestRegressor, GradientBoostingRegressor
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier, plot_tree,␣
 ↪DecisionTreeRegressor
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier,␣
 ↪GradientBoostingClassifier, RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import accuracy_score, mean_squared_error
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,␣
 ↪precision_score, recall_score
from sklearn.metrics import classification_report, mean_squared_error, r2_score
from sklearn.model_selection import train_test_split, GridSearchCV,␣
 ↪cross_val_score, StratifiedKFold, KFold
```

[107]:
```
# Loading data from Google Drive
data_path = '/Users/sowjanyapadala/Desktop/Coursework/Q3/
 ↪DATA_5322_Statistical_Machine_Learning2/Final_Project/Spotify_project/
 ↪Dataset/SpotifyFeatures.csv'
spotify_df = pd.read_csv(data_path)
```

[109]:
```
# Clean duplicated Tracks
spotify_df= spotify_df.drop_duplicates(subset=['track_id'])
count = spotify_df['track_id'].value_counts()
num_duplicated_ids = (count > 1).sum()
print("Number of unique tracks that are duplicated after cleaning:",␣
 ↪num_duplicated_ids)
```

```
Number of unique tracks that are duplicated after cleaning: 0
```

[111]:
```
# Strip Leading/Trailing Whitespace and Fix Case
spotify_df['genre'] = spotify_df['genre'].str.replace('’', "'", regex=False)
spotify_df['genre'].unique()
```

[111]:
```
array(['Movie', 'R&B', 'A Capella', 'Alternative', 'Country', 'Dance',
       'Electronic', 'Anime', 'Folk', 'Blues', 'Opera', 'Hip-Hop',
       "Children's Music", 'Rap', 'Indie', 'Classical', 'Pop', 'Reggae',
       'Reggaeton', 'Jazz', 'Rock', 'Ska', 'Comedy', 'Soul', 'Soundtrack',
       'World'], dtype=object)
```

[113]:
```
# Mapping similar genre
genre_mapping = {
    'Pop': 'Pop/Rock',
    'Rock': 'Pop/Rock',
    'Indie': 'Pop/Rock',
    'Alternative': 'Pop/Rock',
    'Soul': 'Pop/Rock',
    'Hip-Hop': 'Hip-Hop/Rap/R&B',
    'Rap': 'Hip-Hop/Rap/R&B',
    'R&B': 'Hip-Hop/Rap/R&B',
    'Dance': 'Dance/Electronic',
    'Electronic': 'Dance/Electronic',
    'Reggaeton': 'Dance/Electronic',
    'Reggae': 'Dance/Electronic',
    'Ska': 'Dance/Electronic',
    'Jazz': 'Jazz/Blues',
    'Blues': 'Jazz/Blues',
    'Classical': 'Classical/Opera',
    'Opera': 'Classical/Opera',
    'Country': 'Country/Folk',
    'Folk': 'Country/Folk',
    'World': 'World/Soundtrack',
    'Soundtrack': 'World/Soundtrack',
    'Movie': 'Movie/Comedy',
    'Comedy': 'Movie/Comedy',
    'Children\'s Music': 'Children/Anime',
    'Anime': 'Children/Anime',
    'A Capella': 'Pop/Rock'
}
spotify_df['genre_grouped'] = spotify_df['genre'].map(genre_mapping)
```

[115]:
```
print(spotify_df['genre_grouped'].value_counts())
```

```
genre_grouped
Dance/Electronic    42384
Pop/Rock            21606
Movie/Comedy        17476
Classical/Opera     16991
```

```
Jazz/Blues          16535
World/Soundtrack    16453
Children/Anime      15676
Country/Folk        15431
Hip-Hop/Rap/R&B     14222
Name: count, dtype: int64
```

[117]:
```
# Log Transform
spotify_df['duration_ms'] = np.log1p(spotify_df['duration_ms'])
spotify_df['instrumentalness'] = np.log1p(spotify_df['instrumentalness'])
spotify_df['speechiness'] = np.log1p(spotify_df['speechiness'])

# Encode Categorical Data
for col in ['key', 'mode', 'time_signature']:
    le = LabelEncoder()
    spotify_df[col] = le.fit_transform(spotify_df[col])



numeric_features = [
    'popularity', 'acousticness', 'danceability', 'duration_ms', 'energy',
    'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo',␣
 ↪'valence'
]
```

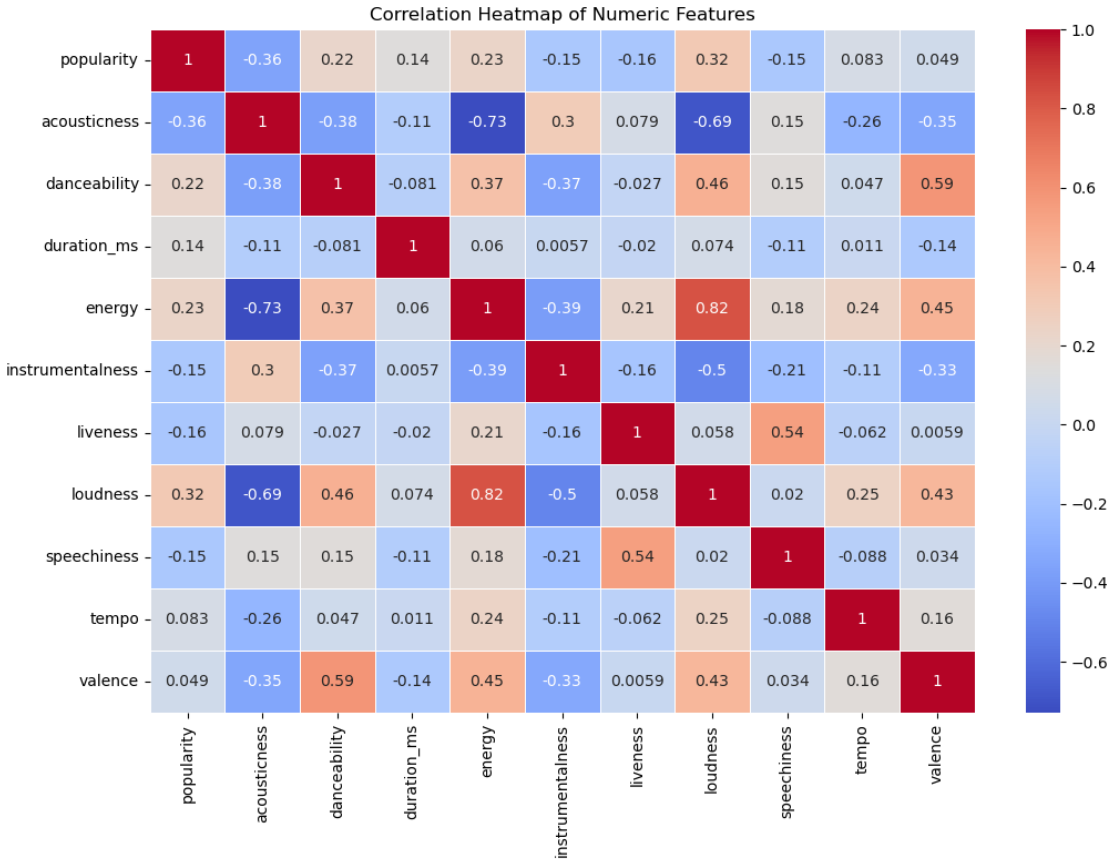
[119]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
spotify_df[numeric_features] = scaler.
 ↪fit_transform(spotify_df[numeric_features])
```

[121]:
```
spotify_df[numeric_features].head(5)
```

[121]:

	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	liveness	loudness	speechiness	tempo	valence
0	-2.085747	0.564740	-0.798733	-1.752675	1.279646	-0.547439	0.575611	1.299283	-0.384671	1.588677	1.353170
1	-2.028246	-0.431708	0.257016	-1.015026	0.652467	-0.547439	-0.348444	0.715907	-0.165891	1.813226	1.360637
2	-1.913244	1.495668	0.640447	-0.526019	-1.544471	-0.547439	-0.575904	-0.585001	-0.491152	-0.565544	-0.312134
3	-2.085747	0.815900	-1.581352	-0.778149	-0.837536	-0.547439	-0.597228	-0.319034	-0.469460	1.741558	-0.838608
4	-1.855743	1.490208	-1.103376	-2.173113	-1.203692	-0.051906	-0.106768	-1.721889	-0.429543	0.746124	-0.229989

[123]:
```
import seaborn as sns
import matplotlib.pyplot as plt
correlation_matrix = spotify_df[numeric_features].corr()
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap of Numeric Features')
plt.show()
```



[125]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
import time
```

[127]:
```
X = spotify_df[numeric_features].values
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(spotify_df['genre_grouped'])
```

```
y_categorical = to_categorical(y_encoded)
```

[129]:
```
for i, label in enumerate(label_encoder.classes_):
    print(f"{i}: {label}")
```

```
0: Children/Anime
1: Classical/Opera
2: Country/Folk
3: Dance/Electronic
4: Hip-Hop/Rap/R&B
5: Jazz/Blues
6: Movie/Comedy
7: Pop/Rock
8: World/Soundtrack
```

[131]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y_categorical, test_size=0.2, random_state=42, stratify=y_encoded
)
```

[141]:
```
# Get the number of features and classes
input_shape = X_train.shape[1]
num_classes = y_train.shape[1]

# Build the model
model1_nn = Sequential()
model1_nn.add(Dense(256, activation='relu', input_shape=(input_shape,)))
model1_nn.add(BatchNormalization())
model1_nn.add(Dropout(0.3))

model1_nn.add(Dense(128, activation='relu'))
model1_nn.add(BatchNormalization())
model1_nn.add(Dropout(0.3))

model1_nn.add(Dense(64, activation='relu'))
model1_nn.add(BatchNormalization())
model1_nn.add(Dropout(0.3))

model1_nn.add(Dense(num_classes, activation='softmax'))

# Compile the model
model1_nn.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy', 'Precision', 'Recall']
)
```

```
/opt/anaconda3/lib/python3.12/site-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
```

```
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[16]:
```
early_stopping = EarlyStopping(
    monitor='val_accuracy',
    patience=10,
    restore_best_weights=True
)


start_time = time.time()  # Record the start time

history1 = model1_nn.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=100,
    batch_size=32,
    verbose=1,
    callbacks=[early_stopping]
)

end_time = time.time()  # Record the end time

total_time = end_time - start_time
print(f"Training completed in {total_time:.2f} seconds.")
```

```
Epoch 1/100
4420/4420             24s 5ms/step -
Precision: 0.6331 - Recall: 0.3138 - accuracy: 0.4883 - loss: 1.4099 -
val_Precision: 0.7422 - val_Recall: 0.4143 - val_accuracy: 0.5962 - val_loss:
1.0854
Epoch 2/100
4420/4420             20s 4ms/step -
Precision: 0.7056 - Recall: 0.3804 - accuracy: 0.5619 - loss: 1.1847 -
val_Precision: 0.7359 - val_Recall: 0.4472 - val_accuracy: 0.6070 - val_loss:
1.0554
Epoch 3/100
4420/4420             21s 5ms/step -
Precision: 0.7125 - Recall: 0.4020 - accuracy: 0.5771 - loss: 1.1487 -
val_Precision: 0.7302 - val_Recall: 0.4505 - val_accuracy: 0.6078 - val_loss:
1.0506
Epoch 4/100
4420/4420             19s 4ms/step -
Precision: 0.7192 - Recall: 0.4118 - accuracy: 0.5814 - loss: 1.1320 -
val_Precision: 0.7451 - val_Recall: 0.4522 - val_accuracy: 0.6117 - val_loss:
1.0384
Epoch 5/100
```

```
4420/4420            23s 5ms/step -
Precision: 0.7225 - Recall: 0.4176 - accuracy: 0.5834 - loss: 1.1238 -
val_Precision: 0.7498 - val_Recall: 0.4492 - val_accuracy: 0.6136 - val_loss:
1.0322
Epoch 6/100
4420/4420            43s 5ms/step -
Precision: 0.7226 - Recall: 0.4198 - accuracy: 0.5840 - loss: 1.1196 -
val_Precision: 0.7494 - val_Recall: 0.4572 - val_accuracy: 0.6165 - val_loss:
1.0303
Epoch 7/100
4420/4420            38s 5ms/step -
Precision: 0.7234 - Recall: 0.4221 - accuracy: 0.5880 - loss: 1.1144 -
val_Precision: 0.7473 - val_Recall: 0.4606 - val_accuracy: 0.6149 - val_loss:
1.0244
Epoch 8/100
4420/4420            19s 4ms/step -
Precision: 0.7264 - Recall: 0.4241 - accuracy: 0.5892 - loss: 1.1089 -
val_Precision: 0.7426 - val_Recall: 0.4723 - val_accuracy: 0.6190 - val_loss:
1.0174
Epoch 9/100
4420/4420            23s 5ms/step -
Precision: 0.7240 - Recall: 0.4269 - accuracy: 0.5897 - loss: 1.1083 -
val_Precision: 0.7530 - val_Recall: 0.4627 - val_accuracy: 0.6197 - val_loss:
1.0163
Epoch 10/100
4420/4420            41s 5ms/step -
Precision: 0.7273 - Recall: 0.4299 - accuracy: 0.5942 - loss: 1.1024 -
val_Precision: 0.7438 - val_Recall: 0.4690 - val_accuracy: 0.6196 - val_loss:
1.0169
Epoch 11/100
4420/4420            20s 5ms/step -
Precision: 0.7259 - Recall: 0.4296 - accuracy: 0.5940 - loss: 1.1013 -
val_Precision: 0.7603 - val_Recall: 0.4545 - val_accuracy: 0.6226 - val_loss:
1.0119
Epoch 12/100
4420/4420            21s 5ms/step -
Precision: 0.7290 - Recall: 0.4321 - accuracy: 0.5941 - loss: 1.0953 -
val_Precision: 0.7568 - val_Recall: 0.4590 - val_accuracy: 0.6215 - val_loss:
1.0138
Epoch 13/100
4420/4420            43s 5ms/step -
Precision: 0.7313 - Recall: 0.4341 - accuracy: 0.5969 - loss: 1.0949 -
val_Precision: 0.7527 - val_Recall: 0.4664 - val_accuracy: 0.6200 - val_loss:
1.0105
Epoch 14/100
4420/4420            41s 5ms/step -
Precision: 0.7285 - Recall: 0.4340 - accuracy: 0.5966 - loss: 1.0926 -
val_Precision: 0.7602 - val_Recall: 0.4586 - val_accuracy: 0.6229 - val_loss:
```

```
1.0067
Epoch 15/100
4420/4420            20s 4ms/step -
Precision: 0.7296 - Recall: 0.4355 - accuracy: 0.5961 - loss: 1.0886 -
val_Precision: 0.7468 - val_Recall: 0.4794 - val_accuracy: 0.6227 - val_loss:
1.0025
Epoch 16/100
4420/4420            21s 5ms/step -
Precision: 0.7285 - Recall: 0.4357 - accuracy: 0.5954 - loss: 1.0894 -
val_Precision: 0.7572 - val_Recall: 0.4703 - val_accuracy: 0.6257 - val_loss:
1.0047
Epoch 17/100
4420/4420            41s 5ms/step -
Precision: 0.7297 - Recall: 0.4353 - accuracy: 0.5965 - loss: 1.0886 -
val_Precision: 0.7542 - val_Recall: 0.4689 - val_accuracy: 0.6225 - val_loss:
1.0053
Epoch 18/100
4420/4420            20s 4ms/step -
Precision: 0.7299 - Recall: 0.4353 - accuracy: 0.5959 - loss: 1.0881 -
val_Precision: 0.7579 - val_Recall: 0.4643 - val_accuracy: 0.6246 - val_loss:
1.0015
Epoch 19/100
4420/4420            22s 5ms/step -
Precision: 0.7310 - Recall: 0.4378 - accuracy: 0.5987 - loss: 1.0829 -
val_Precision: 0.7601 - val_Recall: 0.4681 - val_accuracy: 0.6251 - val_loss:
1.0040
Epoch 20/100
4420/4420            43s 5ms/step -
Precision: 0.7332 - Recall: 0.4420 - accuracy: 0.6010 - loss: 1.0768 -
val_Precision: 0.7544 - val_Recall: 0.4662 - val_accuracy: 0.6239 - val_loss:
1.0017
Epoch 21/100
4420/4420            37s 4ms/step -
Precision: 0.7327 - Recall: 0.4382 - accuracy: 0.5991 - loss: 1.0828 -
val_Precision: 0.7435 - val_Recall: 0.4897 - val_accuracy: 0.6257 - val_loss:
0.9985
Epoch 22/100
4420/4420            21s 5ms/step -
Precision: 0.7327 - Recall: 0.4413 - accuracy: 0.6002 - loss: 1.0806 -
val_Precision: 0.7549 - val_Recall: 0.4727 - val_accuracy: 0.6267 - val_loss:
0.9998
Epoch 23/100
4420/4420            40s 5ms/step -
Precision: 0.7330 - Recall: 0.4411 - accuracy: 0.6012 - loss: 1.0796 -
val_Precision: 0.7477 - val_Recall: 0.4770 - val_accuracy: 0.6251 - val_loss:
1.0023
Epoch 24/100
4420/4420            20s 4ms/step -
```

```
Precision: 0.7310 - Recall: 0.4428 - accuracy: 0.6019 - loss: 1.0766 -
val_Precision: 0.7505 - val_Recall: 0.4798 - val_accuracy: 0.6258 - val_loss:
0.9989
Epoch 25/100
4420/4420            21s 5ms/step -
Precision: 0.7330 - Recall: 0.4435 - accuracy: 0.6023 - loss: 1.0748 -
val_Precision: 0.7506 - val_Recall: 0.4809 - val_accuracy: 0.6261 - val_loss:
0.9963
Epoch 26/100
4420/4420            40s 4ms/step -
Precision: 0.7351 - Recall: 0.4450 - accuracy: 0.6013 - loss: 1.0767 -
val_Precision: 0.7619 - val_Recall: 0.4640 - val_accuracy: 0.6259 - val_loss:
0.9993
Epoch 27/100
4420/4420            20s 4ms/step -
Precision: 0.7322 - Recall: 0.4403 - accuracy: 0.6018 - loss: 1.0786 -
val_Precision: 0.7529 - val_Recall: 0.4816 - val_accuracy: 0.6272 - val_loss:
0.9965
Epoch 28/100
4420/4420            21s 5ms/step -
Precision: 0.7343 - Recall: 0.4472 - accuracy: 0.6041 - loss: 1.0695 -
val_Precision: 0.7568 - val_Recall: 0.4760 - val_accuracy: 0.6279 - val_loss:
0.9920
Epoch 29/100
4420/4420            21s 5ms/step -
Precision: 0.7342 - Recall: 0.4426 - accuracy: 0.6023 - loss: 1.0747 -
val_Precision: 0.7558 - val_Recall: 0.4741 - val_accuracy: 0.6275 - val_loss:
0.9961
Epoch 30/100
4420/4420            22s 5ms/step -
Precision: 0.7361 - Recall: 0.4439 - accuracy: 0.6036 - loss: 1.0709 -
val_Precision: 0.7550 - val_Recall: 0.4772 - val_accuracy: 0.6279 - val_loss:
0.9951
Epoch 31/100
4420/4420            39s 5ms/step -
Precision: 0.7334 - Recall: 0.4436 - accuracy: 0.6026 - loss: 1.0757 -
val_Precision: 0.7669 - val_Recall: 0.4666 - val_accuracy: 0.6303 - val_loss:
0.9926
Epoch 32/100
4420/4420            20s 5ms/step -
Precision: 0.7353 - Recall: 0.4450 - accuracy: 0.6039 - loss: 1.0681 -
val_Precision: 0.7587 - val_Recall: 0.4755 - val_accuracy: 0.6290 - val_loss:
0.9920
Epoch 33/100
4420/4420            20s 5ms/step -
Precision: 0.7343 - Recall: 0.4410 - accuracy: 0.5999 - loss: 1.0735 -
val_Precision: 0.7532 - val_Recall: 0.4839 - val_accuracy: 0.6298 - val_loss:
0.9906
```

```
Epoch 34/100
4420/4420            20s 4ms/step -
Precision: 0.7328 - Recall: 0.4442 - accuracy: 0.6026 - loss: 1.0748 -
val_Precision: 0.7605 - val_Recall: 0.4696 - val_accuracy: 0.6288 - val_loss:
0.9924
Epoch 35/100
4420/4420            22s 5ms/step -
Precision: 0.7340 - Recall: 0.4429 - accuracy: 0.6030 - loss: 1.0733 -
val_Precision: 0.7560 - val_Recall: 0.4811 - val_accuracy: 0.6316 - val_loss:
0.9906
Epoch 36/100
4420/4420            41s 5ms/step -
Precision: 0.7356 - Recall: 0.4495 - accuracy: 0.6058 - loss: 1.0666 -
val_Precision: 0.7615 - val_Recall: 0.4743 - val_accuracy: 0.6285 - val_loss:
0.9893
Epoch 37/100
4420/4420            19s 4ms/step -
Precision: 0.7363 - Recall: 0.4456 - accuracy: 0.6040 - loss: 1.0696 -
val_Precision: 0.7504 - val_Recall: 0.4838 - val_accuracy: 0.6257 - val_loss:
0.9944
Epoch 38/100
4420/4420            22s 5ms/step -
Precision: 0.7359 - Recall: 0.4485 - accuracy: 0.6030 - loss: 1.0690 -
val_Precision: 0.7640 - val_Recall: 0.4712 - val_accuracy: 0.6298 - val_loss:
0.9888
Epoch 39/100
4420/4420            19s 4ms/step -
Precision: 0.7343 - Recall: 0.4403 - accuracy: 0.6014 - loss: 1.0758 -
val_Precision: 0.7618 - val_Recall: 0.4736 - val_accuracy: 0.6287 - val_loss:
0.9893
Epoch 40/100
4420/4420            22s 5ms/step -
Precision: 0.7375 - Recall: 0.4462 - accuracy: 0.6042 - loss: 1.0709 -
val_Precision: 0.7523 - val_Recall: 0.4808 - val_accuracy: 0.6277 - val_loss:
0.9908
Epoch 41/100
4420/4420            41s 5ms/step -
Precision: 0.7354 - Recall: 0.4469 - accuracy: 0.6041 - loss: 1.0674 -
val_Precision: 0.7585 - val_Recall: 0.4769 - val_accuracy: 0.6302 - val_loss:
0.9901
Epoch 42/100
4420/4420            40s 5ms/step -
Precision: 0.7375 - Recall: 0.4503 - accuracy: 0.6065 - loss: 1.0643 -
val_Precision: 0.7648 - val_Recall: 0.4688 - val_accuracy: 0.6297 - val_loss:
0.9888
Epoch 43/100
4420/4420            40s 5ms/step -
Precision: 0.7372 - Recall: 0.4441 - accuracy: 0.6014 - loss: 1.0726 -
```

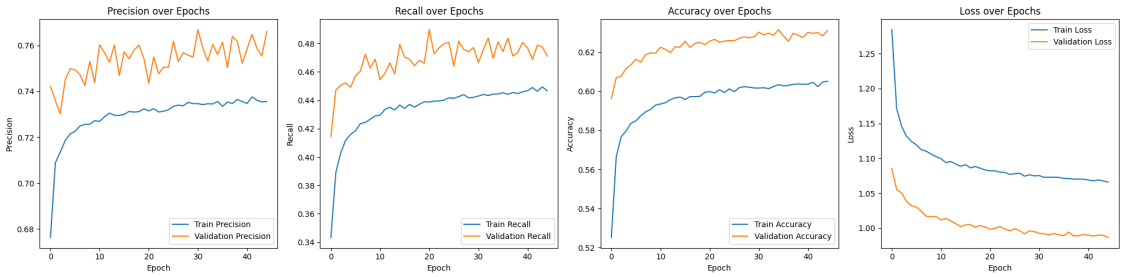
```
val_Precision: 0.7587 - val_Recall: 0.4789 - val_accuracy: 0.6302 - val_loss:
0.9901
Epoch 44/100
4420/4420             20s 4ms/step -
Precision: 0.7362 - Recall: 0.4492 - accuracy: 0.6052 - loss: 1.0663 -
val_Precision: 0.7554 - val_Recall: 0.4776 - val_accuracy: 0.6285 - val_loss:
0.9899
Epoch 45/100
4420/4420             21s 4ms/step -
Precision: 0.7377 - Recall: 0.4496 - accuracy: 0.6067 - loss: 1.0625 -
val_Precision: 0.7662 - val_Recall: 0.4711 - val_accuracy: 0.6312 - val_loss:
0.9867
Training completed in 1231.25 seconds.
```

[17]:
```python
results = model1_nn.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {results[1]:.4f}")
print(f"Test Precision: {results[2]:.4f}")
print(f"Test Recall: {results[3]:.4f}")
```

```
Test Accuracy: 0.6316
Test Precision: 0.7560
Test Recall: 0.4811
```

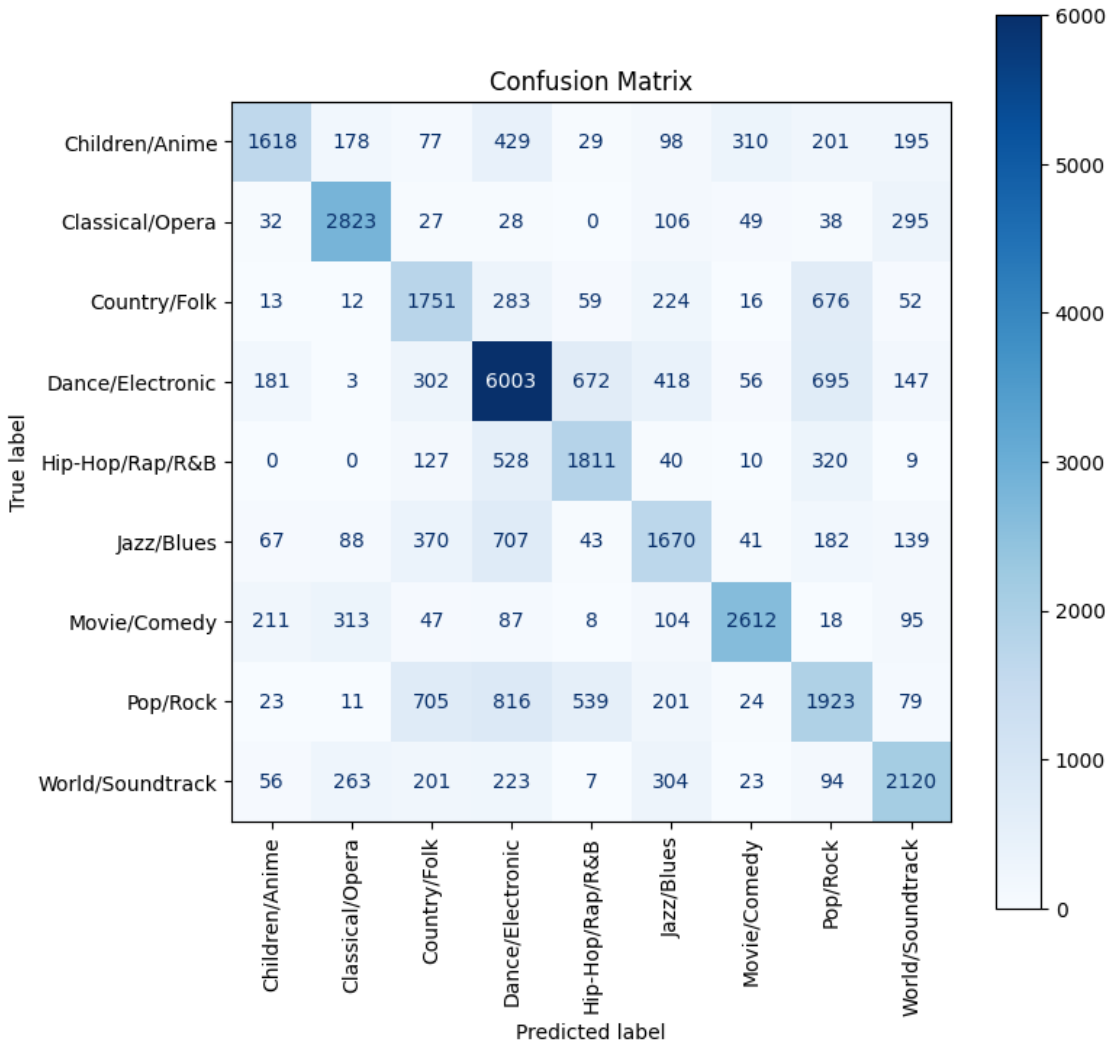
[18]:
```python
import matplotlib.pyplot as plt

plt.figure(figsize=(20, 5))

# Precision Plot
plt.subplot(1, 4, 1)
plt.plot(history1.history['Precision'], label='Train Precision')
plt.plot(history1.history['val_Precision'], label='Validation Precision')
plt.xlabel('Epoch')
plt.ylabel('Precision')
plt.title('Precision over Epochs')
plt.legend()

# Recall Plot
plt.subplot(1, 4, 2)
plt.plot(history1.history['Recall'], label='Train Recall')
plt.plot(history1.history['val_Recall'], label='Validation Recall')
plt.xlabel('Epoch')
plt.ylabel('Recall')
plt.title('Recall over Epochs')
plt.legend()

# Accuracy Plot
plt.subplot(1, 4, 3)
plt.plot(history1.history['accuracy'], label='Train Accuracy')
```

```
plt.plot(history1.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Accuracy over Epochs')
plt.legend()

# Loss Plot
plt.subplot(1, 4, 4)
plt.plot(history1.history['loss'], label='Train Loss')
plt.plot(history1.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss over Epochs')
plt.legend()

plt.tight_layout()
plt.show()
```



[19]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# Predict on the test set
y_pred_probs1 = model1_nn.predict(X_test)

# Convert one-hot predictions to class labels
y_pred1 = np.argmax(y_pred_probs1, axis=1)

# Convert one-hot true labels to class labels
y_true1 = np.argmax(y_test, axis=1)
cm1 = confusion_matrix(y_true1, y_pred1)
class_names = label_encoder.classes_
disp = ConfusionMatrixDisplay(confusion_matrix=cm1, display_labels=class_names)
fig, ax = plt.subplots(figsize=(8,8))
disp.plot(cmap='Blues', ax=ax, xticks_rotation=90)
plt.title('Confusion Matrix')
plt.show()
```

```
1105/1105              2s 1ms/step
```



Performance Metrics (Test Set):
Accuracy: 63.16%
Precision: 75.60%
Recall: 48.11%
These results indicate that the model is effective in predicting genre classes with high precision which means that out of all the tracks the model predicted that the track belongs to particular genre approximately 75.6% were actually correct. However, the lower recall suggests that it misses several true positive cases.
The training/validation accuracy and precision curves shows a consistent upward trend, with validation outperforming training slightly which indicates good generalization.

[20]:
```
# Build the model with 2 hidden layers
model2_nn = Sequential()
```

```
# First hidden layer
model2_nn.add(Dense(256, activation='relu', input_shape=(input_shape,)))
model2_nn.add(BatchNormalization())
model2_nn.add(Dropout(0.3))

# Second hidden layer
model2_nn.add(Dense(128, activation='relu'))
model2_nn.add(BatchNormalization())
model2_nn.add(Dropout(0.3))

# Output layer
model2_nn.add(Dense(num_classes, activation='softmax'))

# Compile the model
model2_nn.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy', 'Precision', 'Recall']
)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[21]:
```
start_time = time.time()
history2 = model2_nn.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=45,
    batch_size=32,
    verbose=1,
    callbacks=[early_stopping]
)
end_time = time.time()  # Record the end time

total_time = end_time - start_time
print(f"Training completed in {total_time:.2f} seconds.")
```

```
Epoch 1/45
4420/4420             21s 4ms/step -
Precision: 0.6358 - Recall: 0.3529 - accuracy: 0.5106 - loss: 1.3504 -
val_Precision: 0.7355 - val_Recall: 0.4145 - val_accuracy: 0.5959 - val_loss:
1.0827
Epoch 2/45
4420/4420             18s 4ms/step -
```

```
Precision: 0.7095 - Recall: 0.4040 - accuracy: 0.5719 - loss: 1.1471 -
val_Precision: 0.7391 - val_Recall: 0.4484 - val_accuracy: 0.6100 - val_loss:
1.0467
Epoch 3/45
4420/4420            18s 4ms/step -
Precision: 0.7153 - Recall: 0.4179 - accuracy: 0.5831 - loss: 1.1179 -
val_Precision: 0.7455 - val_Recall: 0.4520 - val_accuracy: 0.6140 - val_loss:
1.0371
Epoch 4/45
4420/4420            19s 4ms/step -
Precision: 0.7208 - Recall: 0.4210 - accuracy: 0.5841 - loss: 1.1116 -
val_Precision: 0.7498 - val_Recall: 0.4546 - val_accuracy: 0.6182 - val_loss:
1.0263
Epoch 5/45
4420/4420            22s 4ms/step -
Precision: 0.7215 - Recall: 0.4302 - accuracy: 0.5922 - loss: 1.0939 -
val_Precision: 0.7499 - val_Recall: 0.4538 - val_accuracy: 0.6164 - val_loss:
1.0262
Epoch 6/45
4420/4420            21s 4ms/step -
Precision: 0.7240 - Recall: 0.4331 - accuracy: 0.5913 - loss: 1.0917 -
val_Precision: 0.7505 - val_Recall: 0.4467 - val_accuracy: 0.6158 - val_loss:
1.0236
Epoch 7/45
4420/4420            20s 4ms/step -
Precision: 0.7286 - Recall: 0.4335 - accuracy: 0.5952 - loss: 1.0853 -
val_Precision: 0.7394 - val_Recall: 0.4686 - val_accuracy: 0.6162 - val_loss:
1.0200
Epoch 8/45
4420/4420            18s 4ms/step -
Precision: 0.7299 - Recall: 0.4386 - accuracy: 0.5974 - loss: 1.0790 -
val_Precision: 0.7491 - val_Recall: 0.4686 - val_accuracy: 0.6201 - val_loss:
1.0132
Epoch 9/45
4420/4420            19s 4ms/step -
Precision: 0.7291 - Recall: 0.4372 - accuracy: 0.5973 - loss: 1.0776 -
val_Precision: 0.7496 - val_Recall: 0.4668 - val_accuracy: 0.6223 - val_loss:
1.0114
Epoch 10/45
4420/4420            22s 4ms/step -
Precision: 0.7295 - Recall: 0.4399 - accuracy: 0.5990 - loss: 1.0777 -
val_Precision: 0.7503 - val_Recall: 0.4685 - val_accuracy: 0.6241 - val_loss:
1.0074
Epoch 11/45
4420/4420            20s 4ms/step -
Precision: 0.7324 - Recall: 0.4481 - accuracy: 0.6030 - loss: 1.0665 -
val_Precision: 0.7545 - val_Recall: 0.4632 - val_accuracy: 0.6212 - val_loss:
1.0086
```

```
Epoch 12/45
4420/4420            18s 4ms/step -
Precision: 0.7312 - Recall: 0.4417 - accuracy: 0.5995 - loss: 1.0686 -
val_Precision: 0.7439 - val_Recall: 0.4778 - val_accuracy: 0.6226 - val_loss:
1.0074
Epoch 13/45
4420/4420            21s 5ms/step -
Precision: 0.7335 - Recall: 0.4477 - accuracy: 0.6030 - loss: 1.0644 -
val_Precision: 0.7584 - val_Recall: 0.4695 - val_accuracy: 0.6280 - val_loss:
0.9980
Epoch 14/45
4420/4420            42s 5ms/step -
Precision: 0.7309 - Recall: 0.4444 - accuracy: 0.6007 - loss: 1.0663 -
val_Precision: 0.7522 - val_Recall: 0.4723 - val_accuracy: 0.6259 - val_loss:
1.0003
Epoch 15/45
4420/4420            18s 4ms/step -
Precision: 0.7333 - Recall: 0.4482 - accuracy: 0.6025 - loss: 1.0613 -
val_Precision: 0.7589 - val_Recall: 0.4685 - val_accuracy: 0.6272 - val_loss:
0.9981
Epoch 16/45
4420/4420            18s 4ms/step -
Precision: 0.7362 - Recall: 0.4491 - accuracy: 0.6042 - loss: 1.0572 -
val_Precision: 0.7550 - val_Recall: 0.4750 - val_accuracy: 0.6282 - val_loss:
0.9950
Epoch 17/45
4420/4420            20s 5ms/step -
Precision: 0.7342 - Recall: 0.4483 - accuracy: 0.6040 - loss: 1.0596 -
val_Precision: 0.7522 - val_Recall: 0.4712 - val_accuracy: 0.6250 - val_loss:
0.9980
Epoch 18/45
4420/4420            19s 4ms/step -
Precision: 0.7326 - Recall: 0.4479 - accuracy: 0.6050 - loss: 1.0600 -
val_Precision: 0.7513 - val_Recall: 0.4728 - val_accuracy: 0.6271 - val_loss:
1.0008
Epoch 19/45
4420/4420            18s 4ms/step -
Precision: 0.7372 - Recall: 0.4517 - accuracy: 0.6064 - loss: 1.0580 -
val_Precision: 0.7483 - val_Recall: 0.4831 - val_accuracy: 0.6265 - val_loss:
0.9973
Epoch 20/45
4420/4420            21s 4ms/step -
Precision: 0.7343 - Recall: 0.4520 - accuracy: 0.6074 - loss: 1.0580 -
val_Precision: 0.7575 - val_Recall: 0.4711 - val_accuracy: 0.6281 - val_loss:
0.9928
Epoch 21/45
4420/4420            18s 4ms/step -
Precision: 0.7370 - Recall: 0.4501 - accuracy: 0.6062 - loss: 1.0549 -
```

```
val_Precision: 0.7530 - val_Recall: 0.4793 - val_accuracy: 0.6287 - val_loss:
0.9931
Epoch 22/45
4420/4420            18s 4ms/step -
Precision: 0.7363 - Recall: 0.4492 - accuracy: 0.6048 - loss: 1.0573 -
val_Precision: 0.7524 - val_Recall: 0.4798 - val_accuracy: 0.6279 - val_loss:
0.9926
Epoch 23/45
4420/4420            22s 4ms/step -
Precision: 0.7373 - Recall: 0.4516 - accuracy: 0.6060 - loss: 1.0517 -
val_Precision: 0.7629 - val_Recall: 0.4654 - val_accuracy: 0.6289 - val_loss:
0.9937
Epoch 24/45
4420/4420            18s 4ms/step -
Precision: 0.7370 - Recall: 0.4481 - accuracy: 0.6055 - loss: 1.0533 -
val_Precision: 0.7596 - val_Recall: 0.4727 - val_accuracy: 0.6307 - val_loss:
0.9900
Epoch 25/45
4420/4420            23s 5ms/step -
Precision: 0.7360 - Recall: 0.4526 - accuracy: 0.6069 - loss: 1.0506 -
val_Precision: 0.7585 - val_Recall: 0.4755 - val_accuracy: 0.6292 - val_loss:
0.9896
Epoch 26/45
4420/4420            18s 4ms/step -
Precision: 0.7364 - Recall: 0.4527 - accuracy: 0.6070 - loss: 1.0499 -
val_Precision: 0.7639 - val_Recall: 0.4716 - val_accuracy: 0.6306 - val_loss:
0.9900
Epoch 27/45
4420/4420            21s 4ms/step -
Precision: 0.7359 - Recall: 0.4532 - accuracy: 0.6058 - loss: 1.0477 -
val_Precision: 0.7585 - val_Recall: 0.4746 - val_accuracy: 0.6285 - val_loss:
0.9926
Epoch 28/45
4420/4420            21s 4ms/step -
Precision: 0.7377 - Recall: 0.4533 - accuracy: 0.6086 - loss: 1.0504 -
val_Precision: 0.7614 - val_Recall: 0.4701 - val_accuracy: 0.6324 - val_loss:
0.9880
Epoch 29/45
4420/4420            20s 4ms/step -
Precision: 0.7381 - Recall: 0.4551 - accuracy: 0.6093 - loss: 1.0482 -
val_Precision: 0.7598 - val_Recall: 0.4768 - val_accuracy: 0.6337 - val_loss:
0.9851
Epoch 30/45
4420/4420            21s 4ms/step -
Precision: 0.7373 - Recall: 0.4578 - accuracy: 0.6113 - loss: 1.0428 -
val_Precision: 0.7664 - val_Recall: 0.4643 - val_accuracy: 0.6289 - val_loss:
0.9931
Epoch 31/45
```

```
4420/4420            22s 4ms/step -
Precision: 0.7384 - Recall: 0.4570 - accuracy: 0.6087 - loss: 1.0458 -
val_Precision: 0.7604 - val_Recall: 0.4691 - val_accuracy: 0.6314 - val_loss:
0.9890
Epoch 32/45
4420/4420            18s 4ms/step -
Precision: 0.7397 - Recall: 0.4573 - accuracy: 0.6102 - loss: 1.0447 -
val_Precision: 0.7583 - val_Recall: 0.4751 - val_accuracy: 0.6310 - val_loss:
0.9879
Epoch 33/45
4420/4420            18s 4ms/step -
Precision: 0.7395 - Recall: 0.4536 - accuracy: 0.6077 - loss: 1.0498 -
val_Precision: 0.7583 - val_Recall: 0.4776 - val_accuracy: 0.6289 - val_loss:
0.9890
Epoch 34/45
4420/4420            20s 4ms/step -
Precision: 0.7358 - Recall: 0.4524 - accuracy: 0.6067 - loss: 1.0514 -
val_Precision: 0.7623 - val_Recall: 0.4739 - val_accuracy: 0.6307 - val_loss:
0.9874
Epoch 35/45
4420/4420            18s 4ms/step -
Precision: 0.7387 - Recall: 0.4569 - accuracy: 0.6078 - loss: 1.0476 -
val_Precision: 0.7607 - val_Recall: 0.4752 - val_accuracy: 0.6342 - val_loss:
0.9851
Epoch 36/45
4420/4420            20s 4ms/step -
Precision: 0.7370 - Recall: 0.4553 - accuracy: 0.6070 - loss: 1.0464 -
val_Precision: 0.7579 - val_Recall: 0.4741 - val_accuracy: 0.6314 - val_loss:
0.9888
Epoch 37/45
4420/4420            18s 4ms/step -
Precision: 0.7359 - Recall: 0.4540 - accuracy: 0.6084 - loss: 1.0479 -
val_Precision: 0.7585 - val_Recall: 0.4819 - val_accuracy: 0.6328 - val_loss:
0.9848
Epoch 38/45
4420/4420            21s 4ms/step -
Precision: 0.7374 - Recall: 0.4561 - accuracy: 0.6091 - loss: 1.0466 -
val_Precision: 0.7546 - val_Recall: 0.4885 - val_accuracy: 0.6323 - val_loss:
0.9835
Epoch 39/45
4420/4420            18s 4ms/step -
Precision: 0.7355 - Recall: 0.4569 - accuracy: 0.6091 - loss: 1.0465 -
val_Precision: 0.7641 - val_Recall: 0.4739 - val_accuracy: 0.6326 - val_loss:
0.9851
Epoch 40/45
4420/4420            21s 5ms/step -
Precision: 0.7398 - Recall: 0.4582 - accuracy: 0.6097 - loss: 1.0469 -
val_Precision: 0.7576 - val_Recall: 0.4757 - val_accuracy: 0.6301 - val_loss:
```

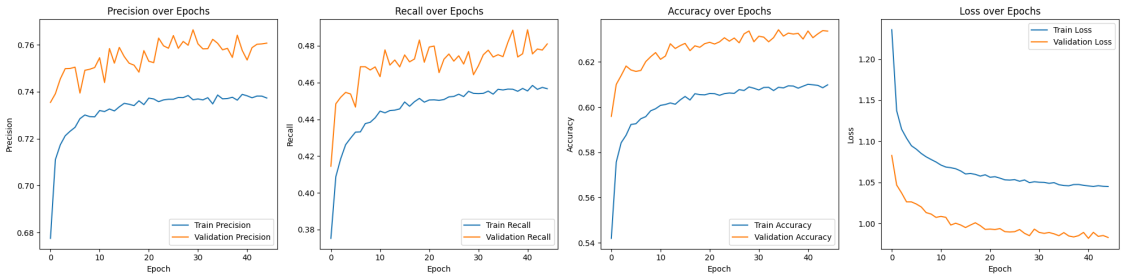
```
0.9891
Epoch 41/45
4420/4420             41s 5ms/step -
Precision: 0.7373 - Recall: 0.4541 - accuracy: 0.6094 - loss: 1.0451 -
val_Precision: 0.7535 - val_Recall: 0.4887 - val_accuracy: 0.6337 - val_loss:
0.9818
Epoch 42/45
4420/4420             40s 5ms/step -
Precision: 0.7392 - Recall: 0.4618 - accuracy: 0.6119 - loss: 1.0390 -
val_Precision: 0.7588 - val_Recall: 0.4756 - val_accuracy: 0.6307 - val_loss:
0.9892
Epoch 43/45
4420/4420             18s 4ms/step -
Precision: 0.7398 - Recall: 0.4569 - accuracy: 0.6107 - loss: 1.0437 -
val_Precision: 0.7602 - val_Recall: 0.4782 - val_accuracy: 0.6324 - val_loss:
0.9843
Epoch 44/45
4420/4420             18s 4ms/step -
Precision: 0.7407 - Recall: 0.4583 - accuracy: 0.6101 - loss: 1.0452 -
val_Precision: 0.7604 - val_Recall: 0.4777 - val_accuracy: 0.6339 - val_loss:
0.9853
Epoch 45/45
4420/4420             18s 4ms/step -
Precision: 0.7374 - Recall: 0.4549 - accuracy: 0.6092 - loss: 1.0463 -
val_Precision: 0.7607 - val_Recall: 0.4810 - val_accuracy: 0.6336 - val_loss:
0.9829
Training completed in 942.36 seconds.
```

[22]:
```python
results = model2_nn.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {results[1]:.4f}")
print(f"Test Precision: {results[2]:.4f}")
print(f"Test Recall: {results[3]:.4f}")
```

```
Test Accuracy: 0.6342
Test Precision: 0.7607
Test Recall: 0.4752
```

[23]:
```python
import matplotlib.pyplot as plt

plt.figure(figsize=(20, 5))

# Precision Plot
plt.subplot(1, 4, 1)
plt.plot(history2.history['Precision'], label='Train Precision')
plt.plot(history2.history['val_Precision'], label='Validation Precision')
plt.xlabel('Epoch')
plt.ylabel('Precision')
plt.title('Precision over Epochs')
```

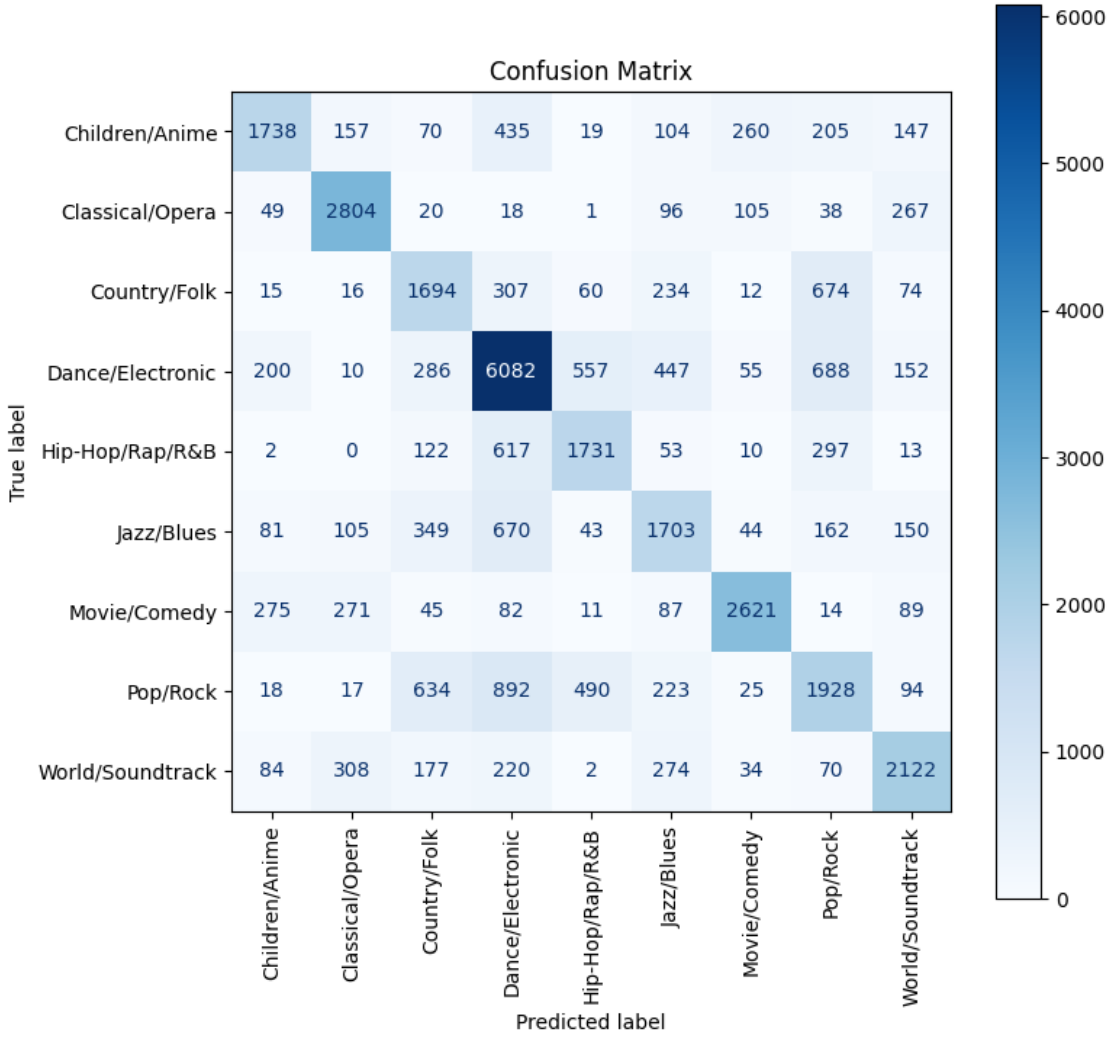
```
plt.legend()

# Recall Plot
plt.subplot(1, 4, 2)
plt.plot(history2.history['Recall'], label='Train Recall')
plt.plot(history2.history['val_Recall'], label='Validation Recall')
plt.xlabel('Epoch')
plt.ylabel('Recall')
plt.title('Recall over Epochs')
plt.legend()

# Accuracy Plot
plt.subplot(1, 4, 3)
plt.plot(history2.history['accuracy'], label='Train Accuracy')
plt.plot(history2.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Accuracy over Epochs')
plt.legend()

# Loss Plot
plt.subplot(1, 4, 4)
plt.plot(history2.history['loss'], label='Train Loss')
plt.plot(history2.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss over Epochs')
plt.legend()

plt.tight_layout()
plt.show()
```



[24]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# Predict on the test set
```

```
y_pred_probs2 = model2_nn.predict(X_test)

# Convert one-hot predictions to class labels
y_pred2 = np.argmax(y_pred_probs2, axis=1)

# Convert one-hot true labels to class labels
y_true2= np.argmax(y_test, axis=1)
cm2 = confusion_matrix(y_true2, y_pred2)
class_names = label_encoder.classes_
disp = ConfusionMatrixDisplay(confusion_matrix=cm2, display_labels=class_names)
fig, ax = plt.subplots(figsize=(8,8))
disp.plot(cmap='Blues', ax=ax, xticks_rotation=90)
plt.title('Confusion Matrix')
plt.show()
```

```
1105/1105              2s 1ms/step
```



Performance Summary (Test Set):
Accuracy: 63.42%
Precision: 76.07%
Recall: 47.52%
The values when compared to model1, shows not much noticeable improvement in the accuracy, precision and recall is marginally lower compared to model1. This indicates that reducing the number of hidden layers from three to two in Model 2 did not lead to a significant performance gain. The minimal difference suggests that the third hidden layer in Model 1 may not have contributed substantial complexity or learning capacity to justify the additional computational cost. Model 2 thus achieves comparable performance with a simpler architecture, making it a more efficient alternative.

[33]:
```
import pandas as pd
from sklearn.utils import shuffle

# Combine X and y into a DataFrame
data_combined = pd.concat([pd.DataFrame(X), pd.Series(y_encoded,␣
 ↪name='label')], axis=1)

# Find the minimum class count
min_count = data_combined['label'].value_counts().min()

# Downsample each class
downsampled = data_combined.groupby('label').apply(lambda x: x.
 ↪sample(min_count, random_state=42)).reset_index(drop=True)

# Separate features and labels
X_downsampled = downsampled.drop('label', axis=1).values
y_downsampled = downsampled['label'].values

# Encode labels to categorical
y_downsampled_categorical = to_categorical(y_downsampled)

# Split into train/test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X_downsampled, y_downsampled_categorical,
    test_size=0.2,
    random_state=42,
    stratify=y_downsampled
)
```

```
<ipython-input-33-761a065d90de>:11: DeprecationWarning: DataFrameGroupBy.apply
operated on the grouping columns. This behavior is deprecated, and in a future
version of pandas the grouping columns will be excluded from the operation.
Either pass `include_groups=False` to exclude the groupings or explicitly select
```

```
the grouping columns after groupby to silence this warning.
  downsampled = data_combined.groupby('label').apply(lambda x:
x.sample(min_count, random_state=42)).reset_index(drop=True)
```

[39]:
```python
# Get the number of features and classes
input_shape = X_train.shape[1]
num_classes = y_train.shape[1]
# Build the model with 2 hidden layers
model4_nn = Sequential()

# First hidden layer
model4_nn.add(Dense(256, activation='relu', input_shape=(input_shape,)))
model4_nn.add(BatchNormalization())
model4_nn.add(Dropout(0.2))

# Second hidden layer
model4_nn.add(Dense(128, activation='relu'))
model4_nn.add(BatchNormalization())
model4_nn.add(Dropout(0.2))

# Output layer
model4_nn.add(Dense(num_classes, activation='softmax'))

# Compile the model
model4_nn.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy', 'Precision', 'Recall']
)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

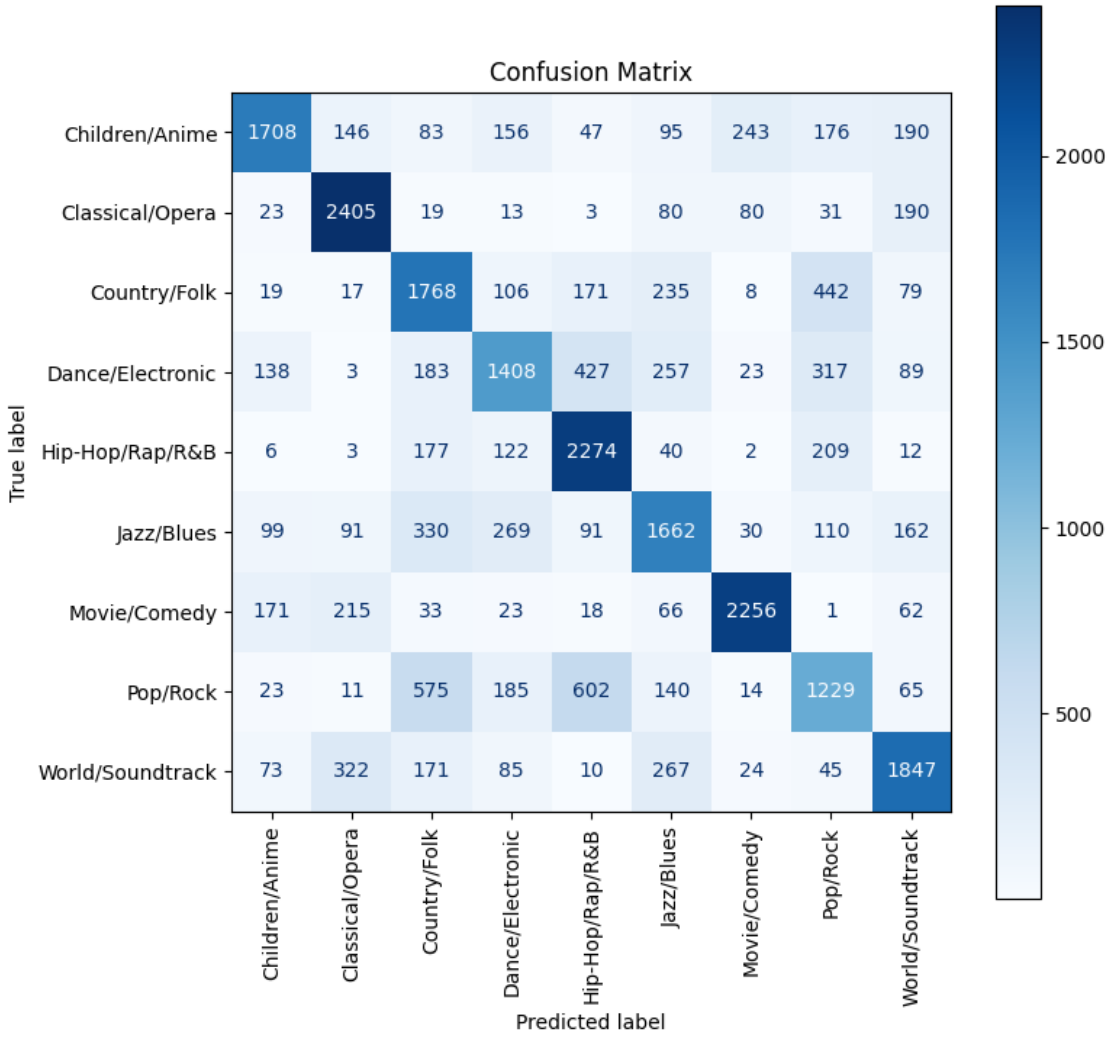
[41]:
```python
results = model4_nn.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {results[1]:.4f}")
print(f"Test Precision: {results[2]:.4f}")
print(f"Test Recall: {results[3]:.4f}")
```

```
Test Accuracy: 0.6468
Test Precision: 0.7621
Test Recall: 0.5098
```

[43]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

```
# Predict on the test set
y_pred_probs4 = model4_nn.predict(X_test)

# Convert one-hot predictions to class labels
y_pred4 = np.argmax(y_pred_probs4, axis=1)

# Convert one-hot true labels to class labels
y_true4= np.argmax(y_test, axis=1)
cm4 = confusion_matrix(y_true4, y_pred4)
class_names = label_encoder.classes_
disp = ConfusionMatrixDisplay(confusion_matrix=cm4, display_labels=class_names)
fig, ax = plt.subplots(figsize=(8,8))
disp.plot(cmap='Blues', ax=ax, xticks_rotation=90)
plt.title('Confusion Matrix')
plt.show()
```

```
800/800                    2s 2ms/step
```



[48]:
```
# Build the model with 2 hidden layers
model5_nn = Sequential()

# First hidden layer
model5_nn.add(Dense(64, activation='relu', input_shape=(input_shape,)))
model5_nn.add(BatchNormalization())
model5_nn.add(Dropout(0.2))

# Second hidden layer
model5_nn.add(Dense(128, activation='relu'))
model5_nn.add(BatchNormalization())
model5_nn.add(Dropout(0.2))

# Output layer
model5_nn.add(Dense(num_classes, activation='softmax'))

# Compile the model
model5_nn.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy', 'Precision', 'Recall']
)
```

[49]:
```
import time
start_time = time.time()
history5 = model5_nn.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=50,
    batch_size=128,
    verbose=1,
    callbacks=[early_stopping],

)
end_time = time.time()  # Record the end time

total_time = end_time - start_time
print(f"Training completed in {total_time:.2f} seconds.")
```

```
Epoch 1/50
800/800             27s 6ms/step -
Precision: 0.6216 - Recall: 0.3058 - accuracy: 0.4706 - loss: 1.4942 -
val_Precision: 0.7343 - val_Recall: 0.4148 - val_accuracy: 0.5963 - val_loss:
1.1150
Epoch 2/50
```

```
800/800             3s 4ms/step -
Precision: 0.7062 - Recall: 0.3935 - accuracy: 0.5677 - loss: 1.1876 -
val_Precision: 0.7353 - val_Recall: 0.4387 - val_accuracy: 0.6056 - val_loss:
1.0889
Epoch 3/50
800/800             3s 4ms/step -
Precision: 0.7132 - Recall: 0.4090 - accuracy: 0.5790 - loss: 1.1580 -
val_Precision: 0.7356 - val_Recall: 0.4479 - val_accuracy: 0.6092 - val_loss:
1.0726
Epoch 4/50
800/800             5s 6ms/step -
Precision: 0.7204 - Recall: 0.4263 - accuracy: 0.5897 - loss: 1.1294 -
val_Precision: 0.7428 - val_Recall: 0.4513 - val_accuracy: 0.6135 - val_loss:
1.0586
Epoch 5/50
800/800             4s 4ms/step -
Precision: 0.7252 - Recall: 0.4347 - accuracy: 0.5944 - loss: 1.1144 -
val_Precision: 0.7410 - val_Recall: 0.4614 - val_accuracy: 0.6152 - val_loss:
1.0484
Epoch 6/50
800/800             5s 4ms/step -
Precision: 0.7244 - Recall: 0.4404 - accuracy: 0.5969 - loss: 1.1076 -
val_Precision: 0.7486 - val_Recall: 0.4576 - val_accuracy: 0.6179 - val_loss:
1.0386
Epoch 7/50
800/800             5s 6ms/step -
Precision: 0.7252 - Recall: 0.4435 - accuracy: 0.5986 - loss: 1.1010 -
val_Precision: 0.7469 - val_Recall: 0.4702 - val_accuracy: 0.6182 - val_loss:
1.0325
Epoch 8/50
800/800             4s 4ms/step -
Precision: 0.7297 - Recall: 0.4534 - accuracy: 0.6050 - loss: 1.0832 -
val_Precision: 0.7425 - val_Recall: 0.4805 - val_accuracy: 0.6218 - val_loss:
1.0263
Epoch 9/50
800/800             5s 6ms/step -
Precision: 0.7273 - Recall: 0.4508 - accuracy: 0.6037 - loss: 1.0911 -
val_Precision: 0.7504 - val_Recall: 0.4662 - val_accuracy: 0.6224 - val_loss:
1.0252
Epoch 10/50
800/800             8s 10ms/step -
Precision: 0.7318 - Recall: 0.4539 - accuracy: 0.6064 - loss: 1.0815 -
val_Precision: 0.7458 - val_Recall: 0.4734 - val_accuracy: 0.6221 - val_loss:
1.0230
Epoch 11/50
800/800             4s 5ms/step -
Precision: 0.7281 - Recall: 0.4509 - accuracy: 0.6019 - loss: 1.0864 -
val_Precision: 0.7483 - val_Recall: 0.4766 - val_accuracy: 0.6261 - val_loss:
```

```
1.0171
Epoch 12/50
800/800             4s 4ms/step -
Precision: 0.7315 - Recall: 0.4528 - accuracy: 0.6046 - loss: 1.0794 -
val_Precision: 0.7506 - val_Recall: 0.4770 - val_accuracy: 0.6273 - val_loss:
1.0147
Epoch 13/50
800/800             5s 6ms/step -
Precision: 0.7316 - Recall: 0.4570 - accuracy: 0.6072 - loss: 1.0730 -
val_Precision: 0.7498 - val_Recall: 0.4758 - val_accuracy: 0.6255 - val_loss:
1.0150
Epoch 14/50
800/800             4s 4ms/step -
Precision: 0.7333 - Recall: 0.4583 - accuracy: 0.6075 - loss: 1.0705 -
val_Precision: 0.7423 - val_Recall: 0.4868 - val_accuracy: 0.6233 - val_loss:
1.0146
Epoch 15/50
800/800             4s 4ms/step -
Precision: 0.7327 - Recall: 0.4636 - accuracy: 0.6108 - loss: 1.0649 -
val_Precision: 0.7478 - val_Recall: 0.4816 - val_accuracy: 0.6273 - val_loss:
1.0115
Epoch 16/50
800/800             4s 5ms/step -
Precision: 0.7341 - Recall: 0.4627 - accuracy: 0.6121 - loss: 1.0643 -
val_Precision: 0.7512 - val_Recall: 0.4791 - val_accuracy: 0.6279 - val_loss:
1.0049
Epoch 17/50
800/800             4s 5ms/step -
Precision: 0.7340 - Recall: 0.4653 - accuracy: 0.6093 - loss: 1.0647 -
val_Precision: 0.7465 - val_Recall: 0.4917 - val_accuracy: 0.6275 - val_loss:
1.0051
Epoch 18/50
800/800             4s 4ms/step -
Precision: 0.7341 - Recall: 0.4655 - accuracy: 0.6141 - loss: 1.0644 -
val_Precision: 0.7454 - val_Recall: 0.4947 - val_accuracy: 0.6295 - val_loss:
1.0029
Epoch 19/50
800/800             4s 5ms/step -
Precision: 0.7326 - Recall: 0.4659 - accuracy: 0.6115 - loss: 1.0632 -
val_Precision: 0.7471 - val_Recall: 0.4893 - val_accuracy: 0.6296 - val_loss:
1.0028
Epoch 20/50
800/800             5s 4ms/step -
Precision: 0.7341 - Recall: 0.4696 - accuracy: 0.6149 - loss: 1.0560 -
val_Precision: 0.7503 - val_Recall: 0.4868 - val_accuracy: 0.6286 - val_loss:
1.0030
Epoch 21/50
800/800             3s 4ms/step -
```

```
Precision: 0.7310 - Recall: 0.4649 - accuracy: 0.6132 - loss: 1.0623 -
val_Precision: 0.7540 - val_Recall: 0.4817 - val_accuracy: 0.6297 - val_loss:
1.0011
Epoch 22/50
800/800             6s 5ms/step -
Precision: 0.7366 - Recall: 0.4706 - accuracy: 0.6163 - loss: 1.0497 -
val_Precision: 0.7542 - val_Recall: 0.4833 - val_accuracy: 0.6291 - val_loss:
0.9997
Epoch 23/50
800/800             4s 5ms/step -
Precision: 0.7345 - Recall: 0.4673 - accuracy: 0.6146 - loss: 1.0574 -
val_Precision: 0.7547 - val_Recall: 0.4816 - val_accuracy: 0.6305 - val_loss:
0.9994
Epoch 24/50
800/800             5s 4ms/step -
Precision: 0.7376 - Recall: 0.4675 - accuracy: 0.6128 - loss: 1.0524 -
val_Precision: 0.7468 - val_Recall: 0.4920 - val_accuracy: 0.6275 - val_loss:
0.9991
Epoch 25/50
800/800             3s 4ms/step -
Precision: 0.7354 - Recall: 0.4681 - accuracy: 0.6118 - loss: 1.0557 -
val_Precision: 0.7531 - val_Recall: 0.4822 - val_accuracy: 0.6290 - val_loss:
0.9998
Epoch 26/50
800/800             6s 5ms/step -
Precision: 0.7365 - Recall: 0.4711 - accuracy: 0.6142 - loss: 1.0504 -
val_Precision: 0.7544 - val_Recall: 0.4850 - val_accuracy: 0.6299 - val_loss:
0.9985
Epoch 27/50
800/800             3s 4ms/step -
Precision: 0.7389 - Recall: 0.4717 - accuracy: 0.6145 - loss: 1.0509 -
val_Precision: 0.7458 - val_Recall: 0.4946 - val_accuracy: 0.6314 - val_loss:
0.9966
Epoch 28/50
800/800             6s 5ms/step -
Precision: 0.7355 - Recall: 0.4715 - accuracy: 0.6162 - loss: 1.0507 -
val_Precision: 0.7522 - val_Recall: 0.4889 - val_accuracy: 0.6326 - val_loss:
0.9941
Epoch 29/50
800/800             4s 5ms/step -
Precision: 0.7334 - Recall: 0.4678 - accuracy: 0.6122 - loss: 1.0591 -
val_Precision: 0.7526 - val_Recall: 0.4894 - val_accuracy: 0.6314 - val_loss:
0.9930
Epoch 30/50
800/800             4s 4ms/step -
Precision: 0.7370 - Recall: 0.4708 - accuracy: 0.6141 - loss: 1.0499 -
val_Precision: 0.7562 - val_Recall: 0.4843 - val_accuracy: 0.6328 - val_loss:
0.9943
```

```
Epoch 31/50
800/800             8s 8ms/step -
Precision: 0.7367 - Recall: 0.4713 - accuracy: 0.6152 - loss: 1.0460 -
val_Precision: 0.7548 - val_Recall: 0.4825 - val_accuracy: 0.6309 - val_loss:
0.9993
Epoch 32/50
800/800             4s 4ms/step -
Precision: 0.7349 - Recall: 0.4707 - accuracy: 0.6167 - loss: 1.0460 -
val_Precision: 0.7493 - val_Recall: 0.4924 - val_accuracy: 0.6310 - val_loss:
0.9928
Epoch 33/50
800/800             5s 4ms/step -
Precision: 0.7400 - Recall: 0.4733 - accuracy: 0.6175 - loss: 1.0463 -
val_Precision: 0.7506 - val_Recall: 0.4860 - val_accuracy: 0.6316 - val_loss:
0.9939
Epoch 34/50
800/800             8s 10ms/step -
Precision: 0.7386 - Recall: 0.4749 - accuracy: 0.6176 - loss: 1.0436 -
val_Precision: 0.7534 - val_Recall: 0.4864 - val_accuracy: 0.6308 - val_loss:
0.9941
Epoch 35/50
800/800             9s 11ms/step -
Precision: 0.7376 - Recall: 0.4730 - accuracy: 0.6166 - loss: 1.0476 -
val_Precision: 0.7560 - val_Recall: 0.4833 - val_accuracy: 0.6299 - val_loss:
0.9963
Epoch 36/50
800/800             5s 6ms/step -
Precision: 0.7396 - Recall: 0.4735 - accuracy: 0.6168 - loss: 1.0457 -
val_Precision: 0.7447 - val_Recall: 0.4989 - val_accuracy: 0.6314 - val_loss:
0.9935
Epoch 37/50
800/800             3s 4ms/step -
Precision: 0.7372 - Recall: 0.4747 - accuracy: 0.6169 - loss: 1.0430 -
val_Precision: 0.7560 - val_Recall: 0.4839 - val_accuracy: 0.6328 - val_loss:
0.9910
Epoch 38/50
800/800             9s 11ms/step -
Precision: 0.7392 - Recall: 0.4744 - accuracy: 0.6181 - loss: 1.0434 -
val_Precision: 0.7551 - val_Recall: 0.4824 - val_accuracy: 0.6340 - val_loss:
0.9940
Epoch 39/50
800/800             8s 9ms/step -
Precision: 0.7357 - Recall: 0.4710 - accuracy: 0.6154 - loss: 1.0438 -
val_Precision: 0.7536 - val_Recall: 0.4878 - val_accuracy: 0.6322 - val_loss:
0.9901
Epoch 40/50
800/800             11s 10ms/step -
Precision: 0.7380 - Recall: 0.4783 - accuracy: 0.6199 - loss: 1.0401 -
```

```
val_Precision: 0.7512 - val_Recall: 0.4888 - val_accuracy: 0.6310 - val_loss:
0.9946
Epoch 41/50
800/800            8s 10ms/step -
Precision: 0.7367 - Recall: 0.4688 - accuracy: 0.6153 - loss: 1.0484 -
val_Precision: 0.7514 - val_Recall: 0.4920 - val_accuracy: 0.6325 - val_loss:
0.9920
Epoch 42/50
800/800            5s 6ms/step -
Precision: 0.7373 - Recall: 0.4715 - accuracy: 0.6182 - loss: 1.0408 -
val_Precision: 0.7526 - val_Recall: 0.4904 - val_accuracy: 0.6329 - val_loss:
0.9927
Epoch 43/50
800/800            3s 4ms/step -
Precision: 0.7378 - Recall: 0.4736 - accuracy: 0.6166 - loss: 1.0449 -
val_Precision: 0.7500 - val_Recall: 0.4960 - val_accuracy: 0.6325 - val_loss:
0.9875
Epoch 44/50
800/800            3s 4ms/step -
Precision: 0.7375 - Recall: 0.4733 - accuracy: 0.6145 - loss: 1.0417 -
val_Precision: 0.7560 - val_Recall: 0.4893 - val_accuracy: 0.6343 - val_loss:
0.9901
Epoch 45/50
800/800            8s 10ms/step -
Precision: 0.7410 - Recall: 0.4758 - accuracy: 0.6193 - loss: 1.0356 -
val_Precision: 0.7511 - val_Recall: 0.4964 - val_accuracy: 0.6331 - val_loss:
0.9880
Epoch 46/50
800/800            8s 6ms/step -
Precision: 0.7390 - Recall: 0.4777 - accuracy: 0.6211 - loss: 1.0444 -
val_Precision: 0.7537 - val_Recall: 0.4934 - val_accuracy: 0.6330 - val_loss:
0.9911
Epoch 47/50
800/800            11s 8ms/step -
Precision: 0.7412 - Recall: 0.4787 - accuracy: 0.6192 - loss: 1.0378 -
val_Precision: 0.7586 - val_Recall: 0.4893 - val_accuracy: 0.6351 - val_loss:
0.9865
Epoch 48/50
800/800            11s 9ms/step -
Precision: 0.7401 - Recall: 0.4760 - accuracy: 0.6196 - loss: 1.0434 -
val_Precision: 0.7539 - val_Recall: 0.4896 - val_accuracy: 0.6339 - val_loss:
0.9906
Epoch 49/50
800/800            6s 4ms/step -
Precision: 0.7428 - Recall: 0.4782 - accuracy: 0.6211 - loss: 1.0363 -
val_Precision: 0.7536 - val_Recall: 0.4950 - val_accuracy: 0.6347 - val_loss:
0.9868
Epoch 50/50
```

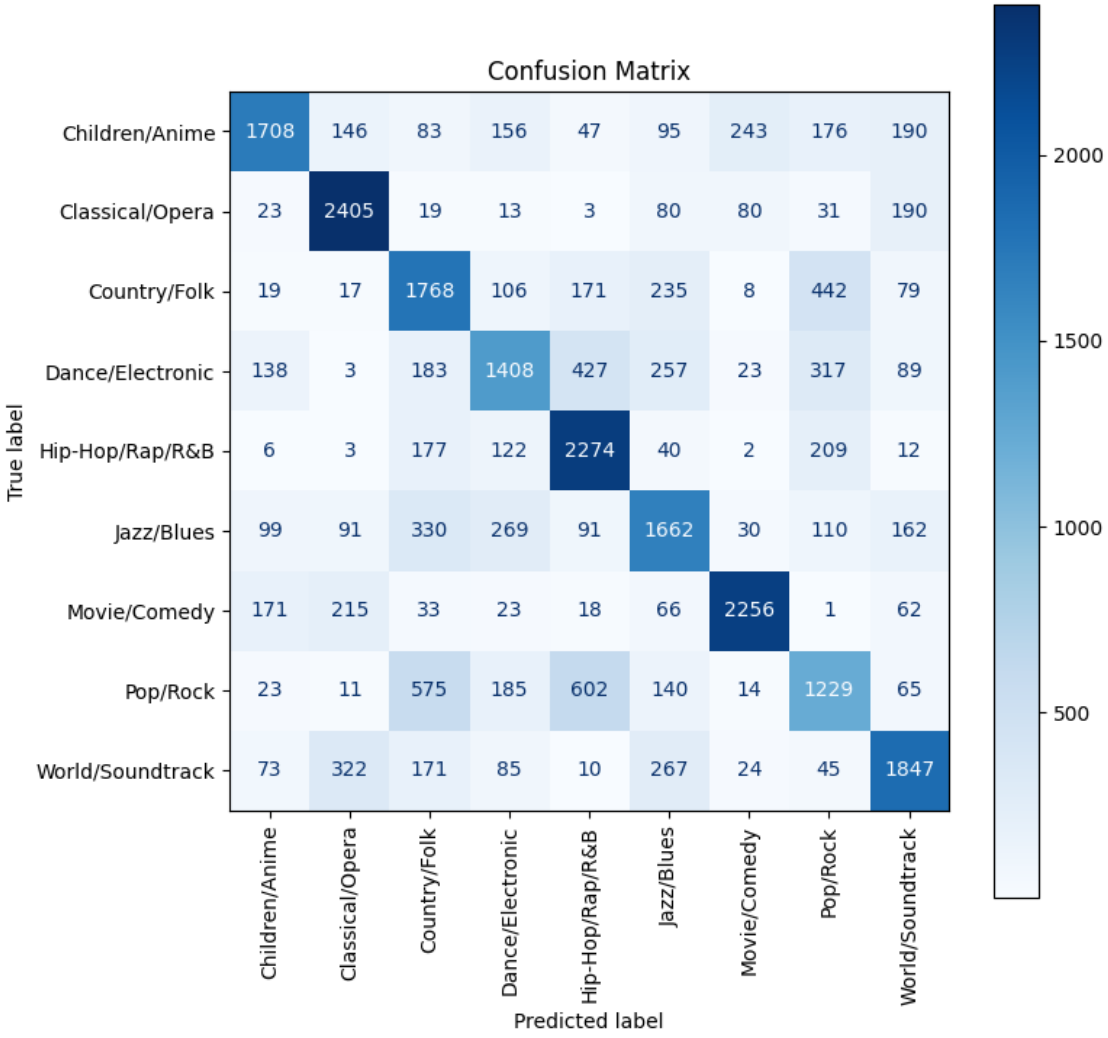
```
800/800             6s 7ms/step -
Precision: 0.7398 - Recall: 0.4778 - accuracy: 0.6214 - loss: 1.0395 -
val_Precision: 0.7546 - val_Recall: 0.4918 - val_accuracy: 0.6343 - val_loss:
0.9879
Training completed in 300.60 seconds.
```

[50]:
```python
results = model5_nn.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {results[1]:.4f}")
print(f"Test Precision: {results[2]:.4f}")
print(f"Test Recall: {results[3]:.4f}")
```

```
Test Accuracy: 0.6351
Test Precision: 0.7586
Test Recall: 0.4893
```

[51]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# Predict on the test set
y_pred_probs5 = model4_nn.predict(X_test)

# Convert one-hot predictions to class labels
y_pred5 = np.argmax(y_pred_probs4, axis=1)

# Convert one-hot true labels to class labels
y_true5= np.argmax(y_test, axis=1)
cm5 = confusion_matrix(y_true5, y_pred5)
class_names = label_encoder.classes_
disp = ConfusionMatrixDisplay(confusion_matrix=cm5, display_labels=class_names)
fig, ax = plt.subplots(figsize=(8,8))
disp.plot(cmap='Blues', ax=ax, xticks_rotation=90)
plt.title('Confusion Matrix')
plt.show()
```

```
800/800             1s 1ms/step
```



[53]:
```
from sklearn.metrics import classification_report


# For Model A
report_a = classification_report(y_true2, y_pred2, target_names=label_encoder.
 ↪classes_)
print("Model A Classification Report:\n", report_a)

# For Model B
report_b = classification_report(y_true4, y_pred4, target_names=label_encoder.
 ↪classes_)
print("Model B Classification Report with Undersampling:\n", report_b)
```

```
Model A Classification Report:
                    precision    recall  f1-score   support
```

```
  Children/Anime       0.71      0.55      0.62      3135
 Classical/Opera       0.76      0.83      0.79      3398
    Country/Folk       0.50      0.55      0.52      3086
Dance/Electronic       0.65      0.72      0.68      8477
 Hip-Hop/Rap/R&B       0.59      0.61      0.60      2845
      Jazz/Blues       0.53      0.51      0.52      3307
    Movie/Comedy       0.83      0.75      0.79      3495
        Pop/Rock       0.47      0.45      0.46      4321
World/Soundtrack       0.68      0.64      0.66      3291

        accuracy                           0.63     35355
       macro avg       0.64      0.62      0.63     35355
    weighted avg       0.64      0.63      0.63     35355

Model B Classification Report with Undersampling:
                   precision    recall  f1-score   support

  Children/Anime       0.76      0.60      0.67      2844
 Classical/Opera       0.75      0.85      0.79      2844
    Country/Folk       0.53      0.62      0.57      2845
Dance/Electronic       0.59      0.49      0.54      2845
 Hip-Hop/Rap/R&B       0.62      0.80      0.70      2845
      Jazz/Blues       0.58      0.58      0.58      2844
    Movie/Comedy       0.84      0.79      0.82      2845
        Pop/Rock       0.48      0.43      0.45      2844
World/Soundtrack       0.69      0.65      0.67      2844

        accuracy                           0.65     25600
       macro avg       0.65      0.65      0.64     25600
    weighted avg       0.65      0.65      0.64     25600
```

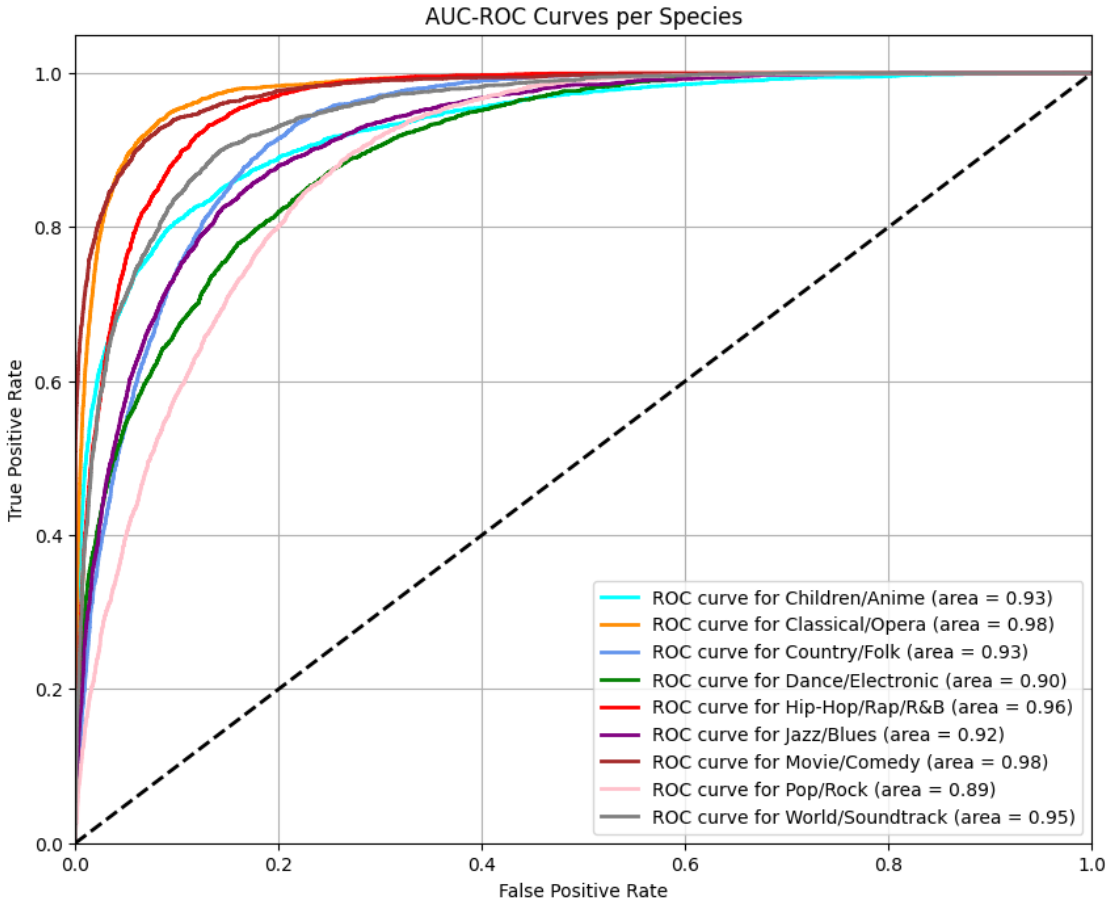
Performance Summary (Test Set) for model 4(2 hidden layers with 256 and 128 units):
Accuracy: 64.68%
Precision: 76.21%
Recall: 50.98%
These results mark a notable improvement in recall, while accuracy and precision slightly exceeded when compared to previous models. The higher recall indicates the model became more sensitive to identifying correct genre labels across all classes, likely due to the balanced class distribution during training. Undersampling the data led to better generalization across all genre classes, especially for those with the fewer samples for the class.
Even the confusion matrix shows predictions are more evenly distributed across genre classes, with less dominance by majority class Dance/Electronic.

Performance Summary (Test Set) for model 5(2 hidden layers with 64 and 128 units):
Accuracy: 63.52%
Precision: 75.86%
Recall: 48.93%

Compared to Model 4, Model 5 demonstrated a slight decline in all performance metrics. This suggests that reducing the capacity of the first hidden layer to 64 units may have limited the model's ability to effectively learn complex feature patterns, resulting in lower generalization performance.

Among all models evaluated, the neural network trained on balanced (undersampled) data with two hidden layers of sizes 256 and 128 (Model 4) achieved the best overall results. This makes it the most effective architecture for multiclass genre classification in this project.

[57]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
y_true = y_test
n_classes = y_true.shape[1]

fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_true[:, i], y_pred_probs4[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
plt.figure(figsize=(10, 8))
colors = ['aqua', 'darkorange', 'cornflowerblue', 'green', 'red',
          'purple', 'brown', 'pink', 'grey', 'olive', 'gold', 'black']

for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=2,
             label=f'ROC curve for {label_encoder.classes_[i]} (area =␣
 ↪{roc_auc[i]:0.2f})')

plt.plot([0, 1], [0, 1], 'k--', lw=2)  # Diagonal line

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('AUC-ROC Curves per Species')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```



From the above plot, the AUC (Area Under Curve) values range from 0.89 to 0.98, which indicates that the model performs very well across all genre classes.

Classical/Opera and Movie/Comedy achieved the highest AUC values of 0.98, suggesting that the model is highly effective in distinguishing these genres from the others. The lowest AUC was for Pop/Rock (0.89), indicating that this genre is the most challenging to classify.

0.1 Conventional Methods

[135]:
```
df_multi = X.copy()
y_multi = spotify_df['genre_grouped']
X_train, X_test, y_train, y_test = train_test_split(
    df_multi, y_multi, test_size=0.2, random_state=42, stratify=y_multi
)
```

0.2 Decision Trees

[138]:
```
cv = KFold(n_splits = 5, shuffle = True, random_state = 5322)
param_grid = {
    'max_depth': [3, 5, 6, X_train.shape[1]],
    'min_samples_split': [2, 3, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'class_weight': [None, 'balanced']
}

multi_dt = DecisionTreeClassifier(random_state = 5322)
grid_search = GridSearchCV(
    estimator = multi_dt,
    param_grid = param_grid,
    cv = cv,
    scoring = 'f1_weighted',
    n_jobs = -1,
    verbose = 1
)

start_time = time.time()
grid_search.fit(X_train, y_train)
end_time = time.time()
training_duration = end_time - start_time
print(f"Training time (Decision Tree with GridSearchCV): {training_duration:.
 ↪2f} seconds")
```

```
Fitting 5 folds for each of 96 candidates, totalling 480 fits
Training time (Decision Tree with GridSearchCV): 56.89 seconds
```

[140]:
```
grid_search.best_params_
```

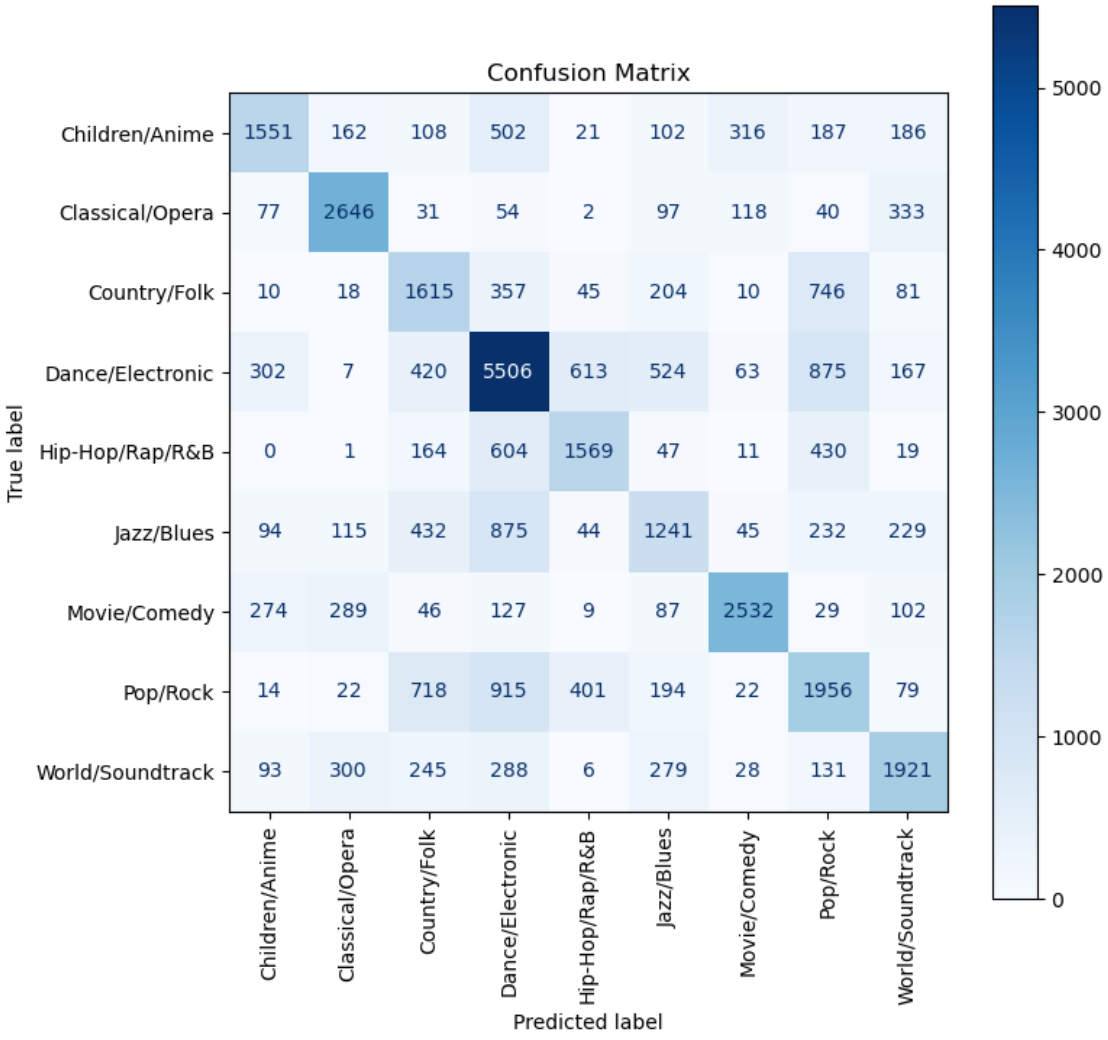
[140]:
```
{'class_weight': None,
 'max_depth': 11,
 'min_samples_leaf': 4,
 'min_samples_split': 2}
```

[142]:
```
best_multi_dt = grid_search.best_estimator_

binary_multi_pred = best_multi_dt.predict(X_test)
print(classification_report(y_test, binary_multi_pred))
```

```
                precision    recall  f1-score   support

  Children/Anime       0.64      0.49      0.56      3135
 Classical/Opera       0.74      0.78      0.76      3398
    Country/Folk       0.43      0.52      0.47      3086
Dance/Electronic       0.60      0.65      0.62      8477
 Hip-Hop/Rap/R&B       0.58      0.55      0.56      2845
```

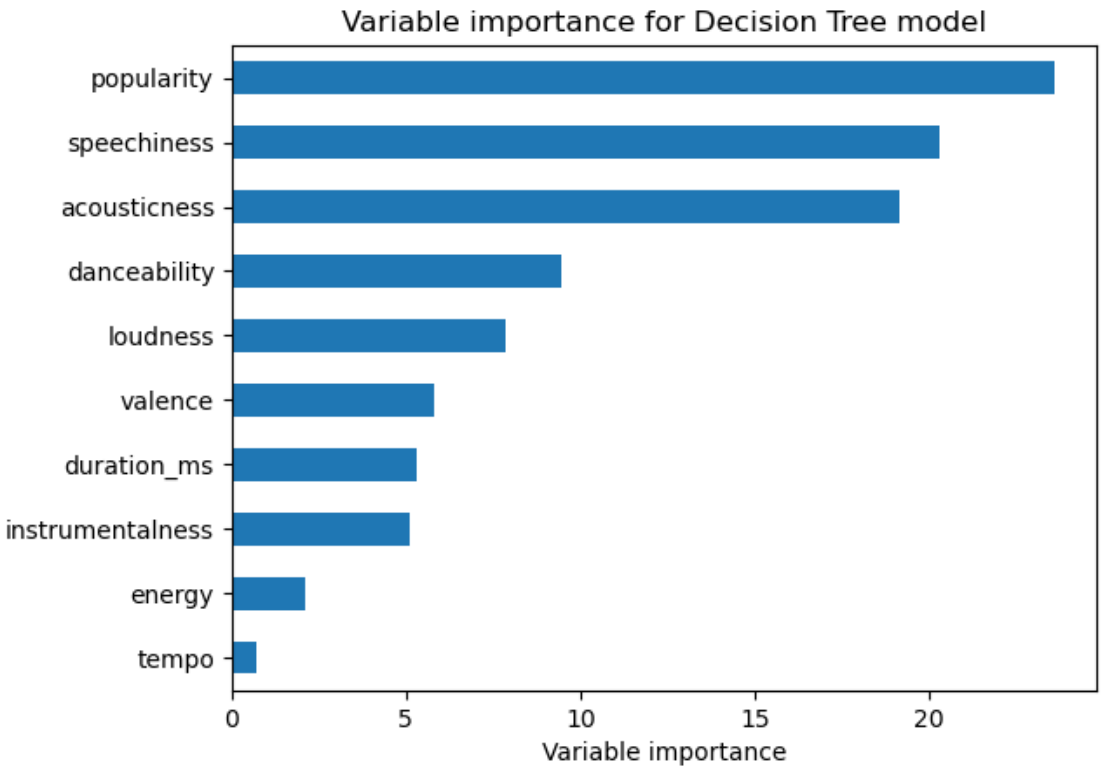
```
          Jazz/Blues       0.45      0.38      0.41      3307
       Movie/Comedy       0.81      0.72      0.76      3495
           Pop/Rock       0.42      0.45      0.44      4321
  World/Soundtrack       0.62      0.58      0.60      3291

           accuracy                           0.58     35355
          macro avg       0.59      0.57      0.58     35355
       weighted avg       0.59      0.58      0.58     35355
```

[144]:
```
class_names = [
    "Children/Anime",
    "Classical/Opera",
    "Country/Folk",
    "Dance/Electronic",
    "Hip-Hop/Rap/R&B",
    "Jazz/Blues",
    "Movie/Comedy",
    "Pop/Rock",
    "World/Soundtrack"
]
cm_dt = confusion_matrix(y_test, binary_multi_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm_dt,␣
 ↪display_labels=class_names)
fig, ax = plt.subplots(figsize=(8,8))
disp.plot(cmap='Blues', ax=ax, xticks_rotation=90)
plt.title('Confusion Matrix')
plt.show()
```

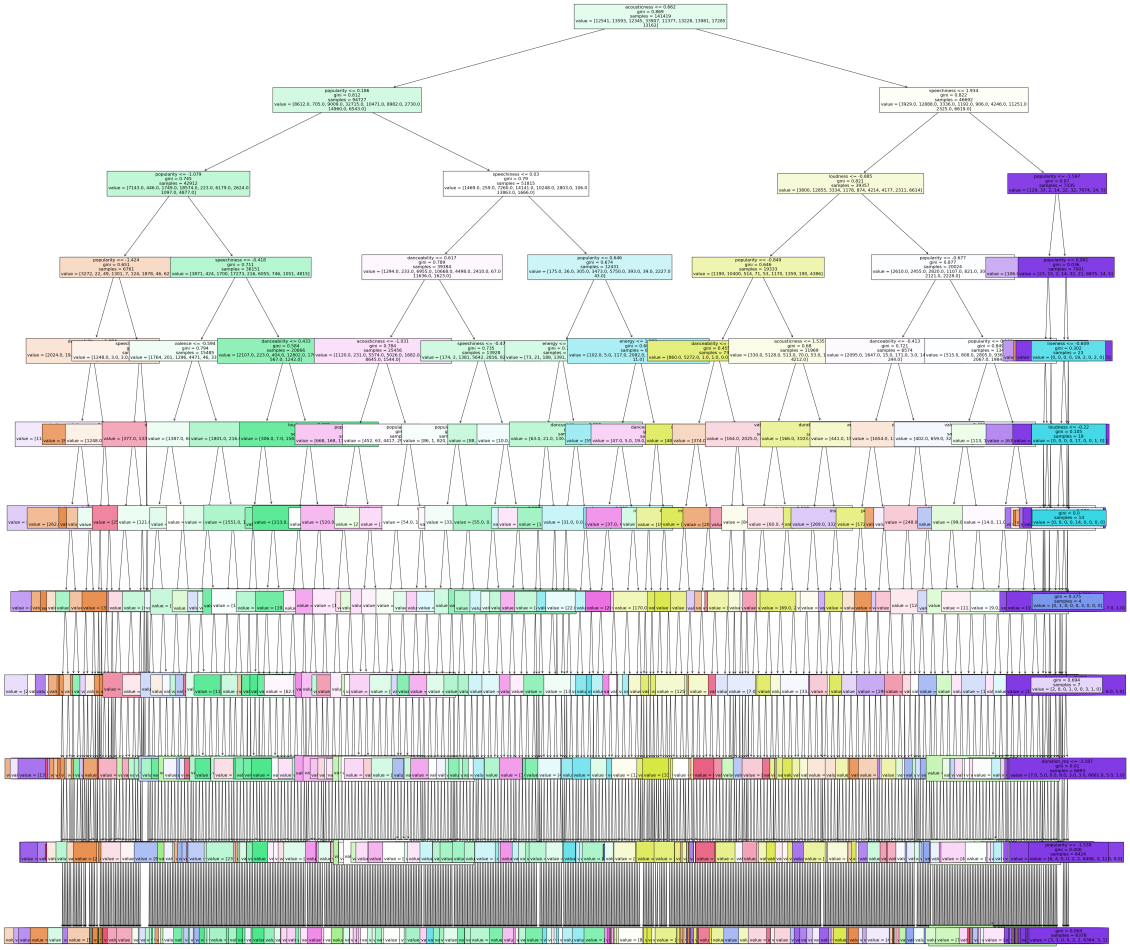


[146]:
```
# Plotting the variable importance for boosting model
feature_importance = best_multi_dt.feature_importances_*100
rel_imp = pd.Series(feature_importance, index=numeric_features).
 ↪sort_values(ascending = True,inplace=False)
rel_imp.tail(10).T.plot(kind='barh')
plt.xlabel('Variable importance')
plt.title('Variable importance for Decision Tree model')
```

[146]: Text(0.5, 1.0, 'Variable importance for Decision Tree model')



[148]:
```
plt.figure(figsize=(50,50))
plot_tree(best_multi_dt
          , filled=True
          , feature_names=numeric_features
          , label='all'
          , fontsize=12)
plt.show()
```



The Decision Tree model was trained to classify tracks into one of nine grouped genres using musical features such as tempo, loudness, acousticness, and danceability. Hyperparameters 'max_depth', 'min_samples_split', 'min_samples_leaf' were optimized using a grid search with 5-fold cross-validation.

Best Model Parameters: max_depth: 11 min_samples_split: 2 min_samples_leaf: 4 class_weight: None

Overall Performance: Accuracy: 58%

Children/Anime - Of all the tracks the model labeled as Children/Anime, 64% were actually from this genre. However, it only managed to correctly identify 49% of all actual Children/Anime tracks. So it's fairly precise but misses many true ones.

Classical/Opera - Of all the tracks labeled Classical/Opera, 74% were correct. And it captured 78% of all true Classical/Opera tracks. This is one of the most reliable genres in terms of prediction.

Country/Folk - Only 43% of tracks predicted as Country/Folk were correct meaning it's often confused with other genres. It found 52% of actual Country/Folk tracks. Both precision and recall

are low, indicating confusion with similar genres.

Dance/Electronic - When predicting Dance/Electronic, 60% of those predictions were correct, and it captured 65% of true instances. This is decent, showing good model understanding for this genre.

Hip-Hop/Rap/R&B - Of all the tracks predicted as Hip-Hop/Rap/R&B, 58% were correct, and it found 55% of the actual ones. Performance here is average, with moderate false positives and false negatives.

Jazz/Blues - Only 45% of the predicted Jazz/Blues tracks were truly from this genre, and just 38% of actual Jazz/Blues tracks were identified. This is one of the weakest genres, indicating heavy misclassification.

Movie/Comedy - 81% of the tracks labeled as Movie/Comedy were correct showing the model is very confident when it predicts this genre. It also identified 72% of all actual Movie/Comedy tracks. This is a very strong performing class.

Pop/Rock - Only 42% of the tracks predicted as Pop/Rock were correct, and the model found 45% of the true Pop/Rock tracks. Performance is weak, possibly due to similarity with Country or Dance genres.

World/Soundtrack - Of all tracks labeled World/Soundtrack, 62% were correct, and it captured 58% of actual ones. This is a moderately well-performing class, with some confusion likely with Classical or Movie genres.

0.3 Random Forest

[154]:
```
# Random Forest

# Parameter grid for tuning
#param_grid = {
    #'n_estimators': [150, 200, 250],
    #'max_depth': [4,5,6],
    #'min_samples_split': [2, 5],
    #'min_samples_leaf': [1, 2],
    #'max_features': [0.5]
#}

#grid_search = GridSearchCV(
    #estimator=rf,
    #param_grid=param_grid,
    #cv=5,
    #scoring='f1_weighted',
    #n_jobs=-1,
   #verbose=1
#)
rf = RandomForestClassifier(n_estimators = 200, max_depth = 6, max_features=6,␣
 ↪random_state=42)
start_time = time.time()
rf.fit(X_train, y_train)
```

```
end_time = time.time()
training_duration = end_time - start_time
print(f"Training time (RandomForest): {training_duration:.2f} seconds")
```

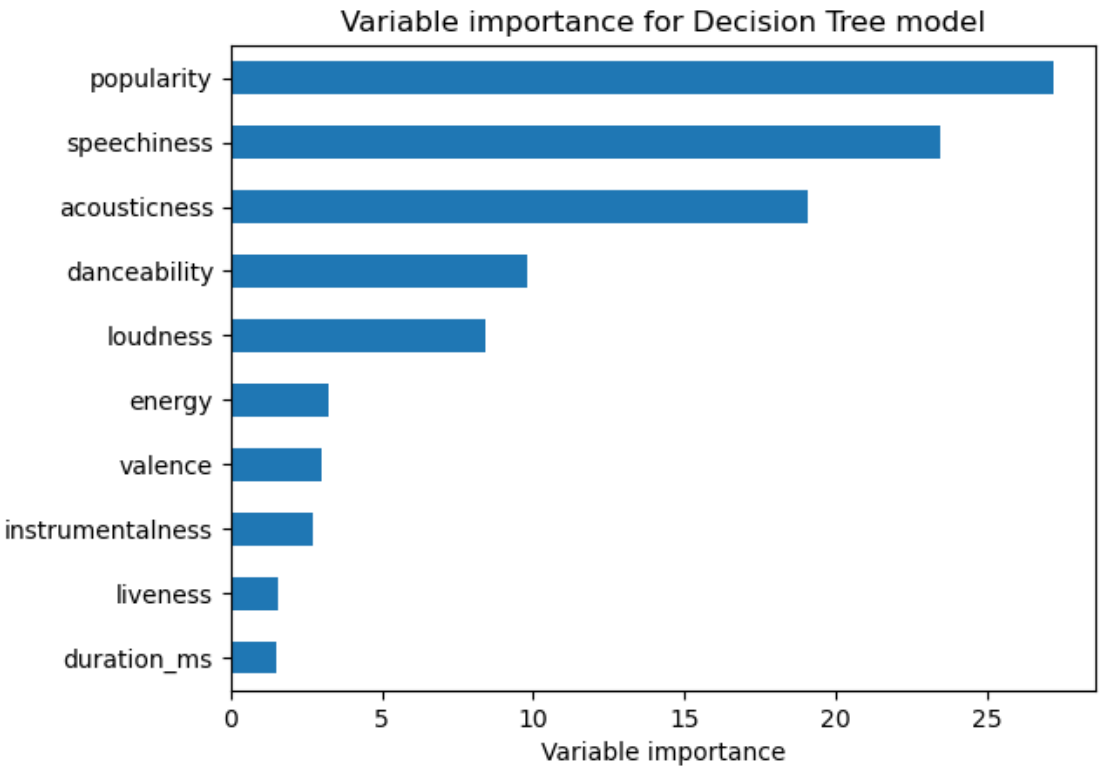
```
Training time (RandomForest): 38.42 seconds
```

[170]:
```
y_pred_best = rf.predict(X_test)
print("Classification Report:")
print(classification_report(y_test, y_pred_best))
```

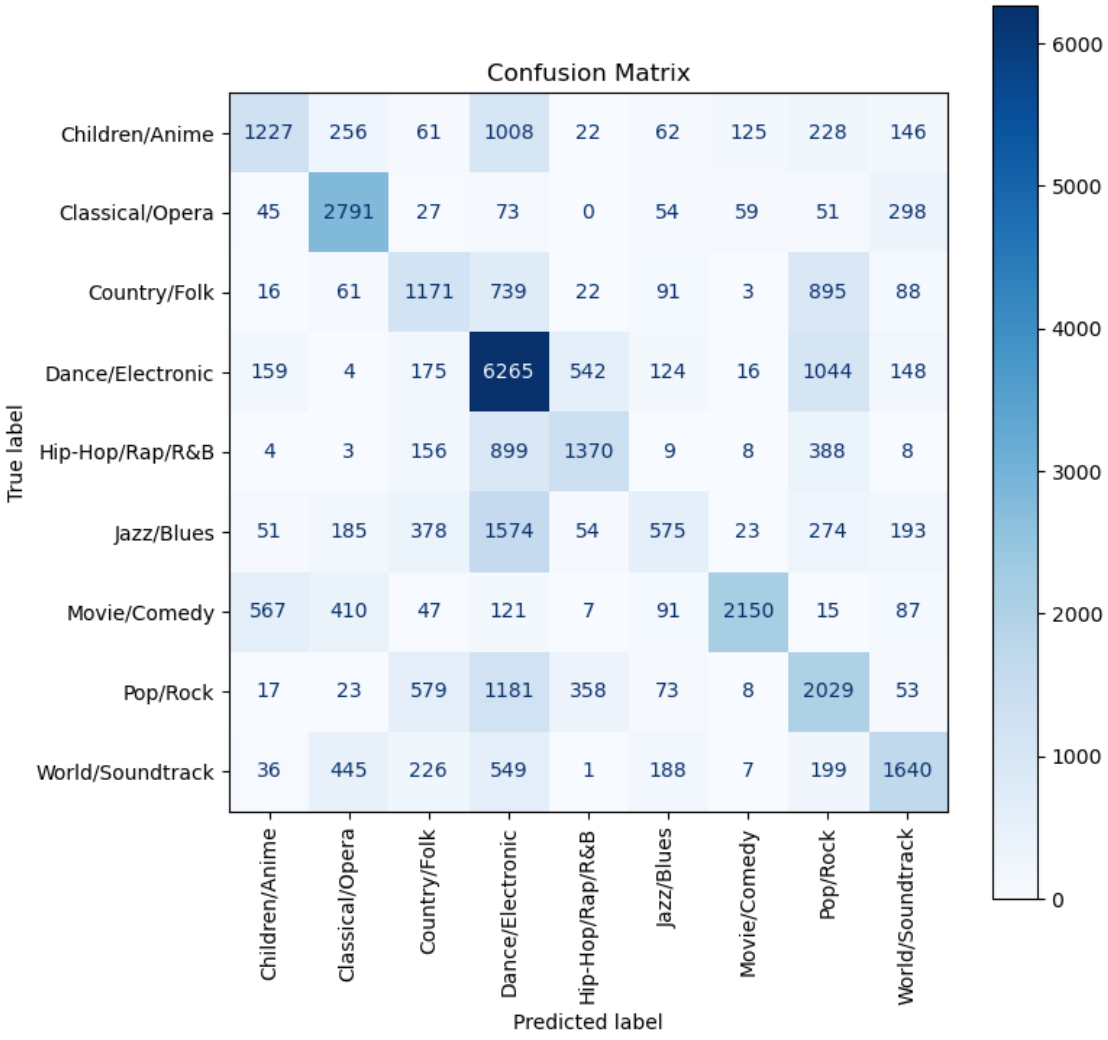
```
Classification Report:
                  precision    recall  f1-score   support

  Children/Anime       0.58      0.39      0.47      3135
 Classical/Opera       0.67      0.82      0.74      3398
    Country/Folk       0.42      0.38      0.40      3086
Dance/Electronic       0.50      0.74      0.60      8477
 Hip-Hop/Rap/R&B       0.58      0.48      0.52      2845
      Jazz/Blues       0.45      0.17      0.25      3307
    Movie/Comedy       0.90      0.62      0.73      3495
       Pop/Rock       0.40      0.47      0.43      4321
World/Soundtrack       0.62      0.50      0.55      3291

        accuracy                           0.54     35355
       macro avg       0.57      0.51      0.52     35355
    weighted avg       0.56      0.54      0.53     35355
```

[172]:
```
feature_importance = rf.feature_importances_*100
rel_imp = pd.Series(feature_importance, index=numeric_features).
 ↪sort_values(ascending = True,inplace=False)
rel_imp.tail(10).T.plot(kind='barh')
plt.xlabel('Variable importance')
plt.title('Variable importance for Decision Tree model')
```

[172]:
```
Text(0.5, 1.0, 'Variable importance for Decision Tree model')
```

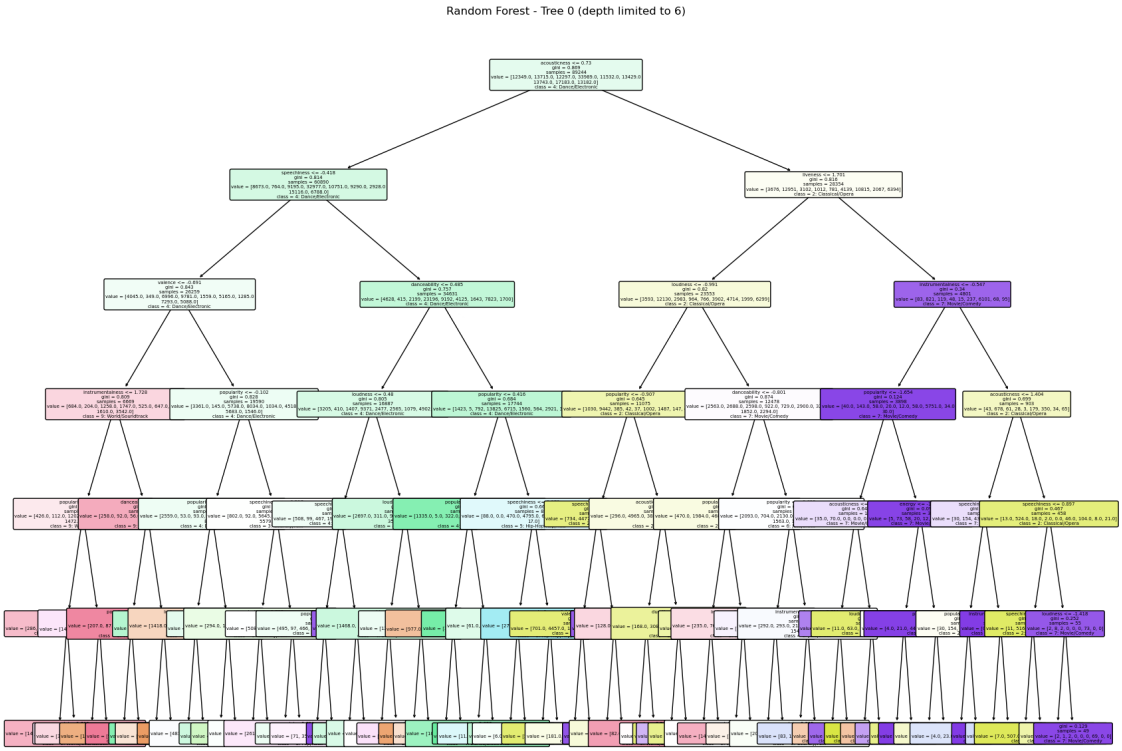


[174]:
```
cm_rf = confusion_matrix(y_test, y_pred_best)
disp = ConfusionMatrixDisplay(confusion_matrix=cm_rf,␣
 ↪display_labels=class_names)
fig, ax = plt.subplots(figsize=(8,8))
disp.plot(cmap='Blues', ax=ax, xticks_rotation=90)
plt.title('Confusion Matrix')
plt.show()
```



[176]:
```
class_names1 = [
    "1: Children/Anime",
    "2: Classical/Opera",
    "3: Country/Folk",
    "4: Dance/Electronic",
    "5: Hip-Hop/Rap/R&B",
    "6: Jazz/Blues",
    "7: Movie/Comedy",
    "8: Pop/Rock",
    "9: World/Soundtrack"
]
fig = plt.figure(figsize=(20, 15))
plot_tree(rf.estimators_[0], feature_names=numeric_features,␣
 ↪class_names=class_names1, filled=True, rounded=True, fontsize = 5)
```

```
plt.title("Random Forest - Tree 0 (depth limited to 6)")
plt.show()
fig.savefig("rf_tree_depth4.pdf", bbox_inches='tight')
plt.close(fig)
```



Best Model Parameters: n_estimators = 200, max_depth = 6, max_features=6

Model Performance: Accuracy - 54%

Higher precision in Movie/Comedy (90%) and recall in Dance/Electronic (74%) and Classical/Opera (82%). This implies that when the model predicts a track as Movie/Comedy 9 out 10 times eventhough the dataset contains Dance/Electronic as the majority class. Similarly the model successfully retriueves most of the actual tracks belonging to the genre Dance/Electronic and Classical/Opera.

But overall, recall drops sharply in many genres especially Jazz/Blues.

0.4 Gradient Boosting

[180]:
```
# Parameter grid for tuning
#param_grid = {
    #'n_estimators': [150, 200, 250],
    #'max_depth': [4,5,6],
    #'min_samples_split': [2, 5],
```

```
    #'min_samples_leaf': [1, 2],
    #'learning_rate': [0.01, 0.05, 0.1]
#}

#grid_search = GridSearchCV(
    #estimator=Tree_Bst_reg,
    #param_grid=param_grid,
    #cv=5,
    #scoring='f1_weighted',
    #n_jobs=-1,
   #verbose=1
#)
```

[182]:
```
Tree_Bst_reg = GradientBoostingClassifier(n_estimators=250, learning_rate=0.1,␣
 ↪max_depth=4, random_state=1)
start_time = time.time()
Tree_Bst_reg.fit(X_train,y_train)
end_time = time.time()
training_duration = end_time - start_time
print(f"Training time (GradientBoosting ): {training_duration:.2f} seconds")
```

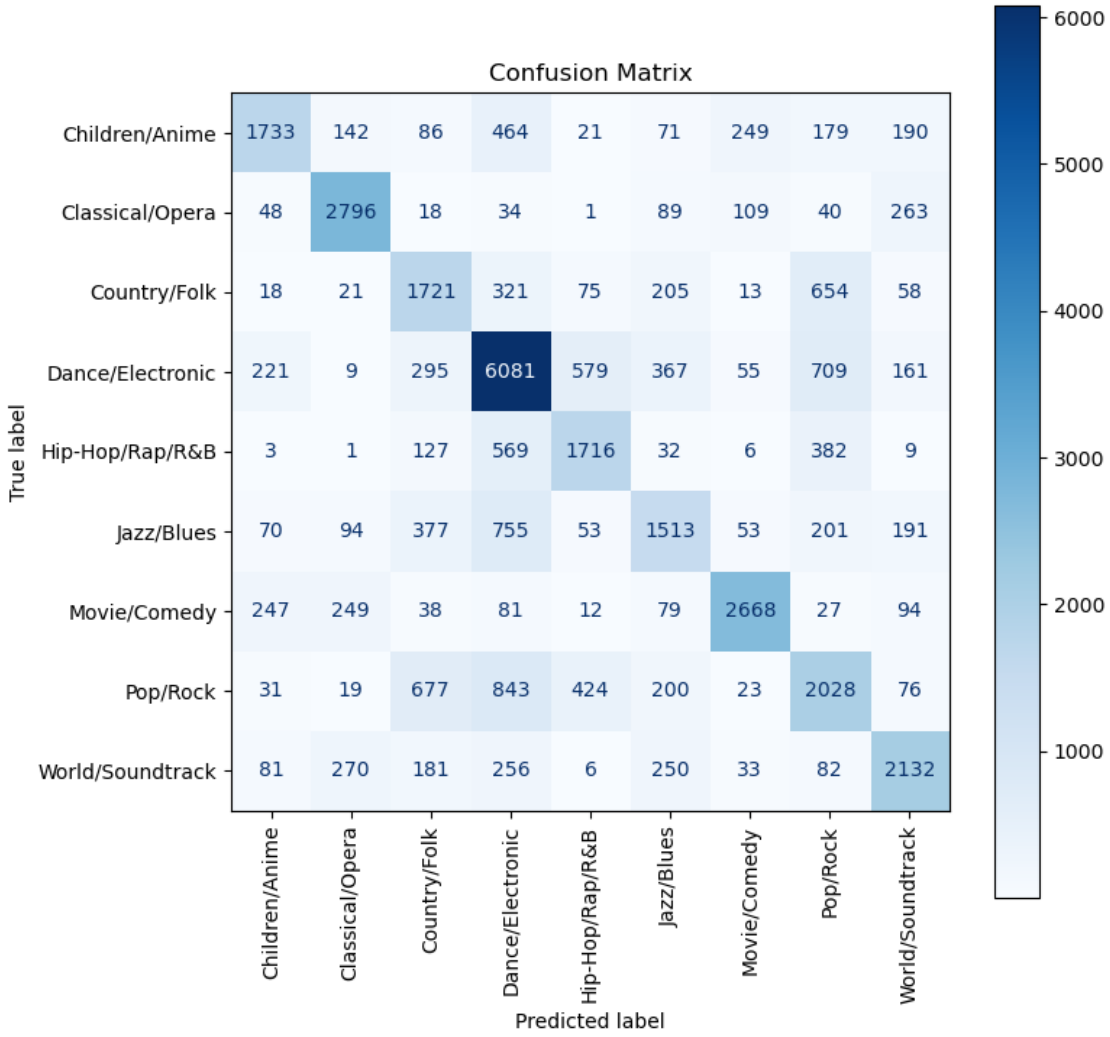
```
Training time (GradientBoosting ): 881.50 seconds
```

[186]:
```
y_pred_bag_reg = Tree_Bst_reg.predict(X_test)
print("Classification Report:")
print(classification_report(y_test, y_pred_bag_reg))
```

```
Classification Report:
                  precision    recall  f1-score   support

  Children/Anime       0.71      0.55      0.62      3135
 Classical/Opera       0.78      0.82      0.80      3398
    Country/Folk       0.49      0.56      0.52      3086
Dance/Electronic       0.65      0.72      0.68      8477
 Hip-Hop/Rap/R&B       0.59      0.60      0.60      2845
      Jazz/Blues       0.54      0.46      0.50      3307
    Movie/Comedy       0.83      0.76      0.80      3495
       Pop/Rock       0.47      0.47      0.47      4321
World/Soundtrack       0.67      0.65      0.66      3291

        accuracy                           0.63     35355
       macro avg       0.64      0.62      0.63     35355
    weighted avg       0.64      0.63      0.63     35355
```

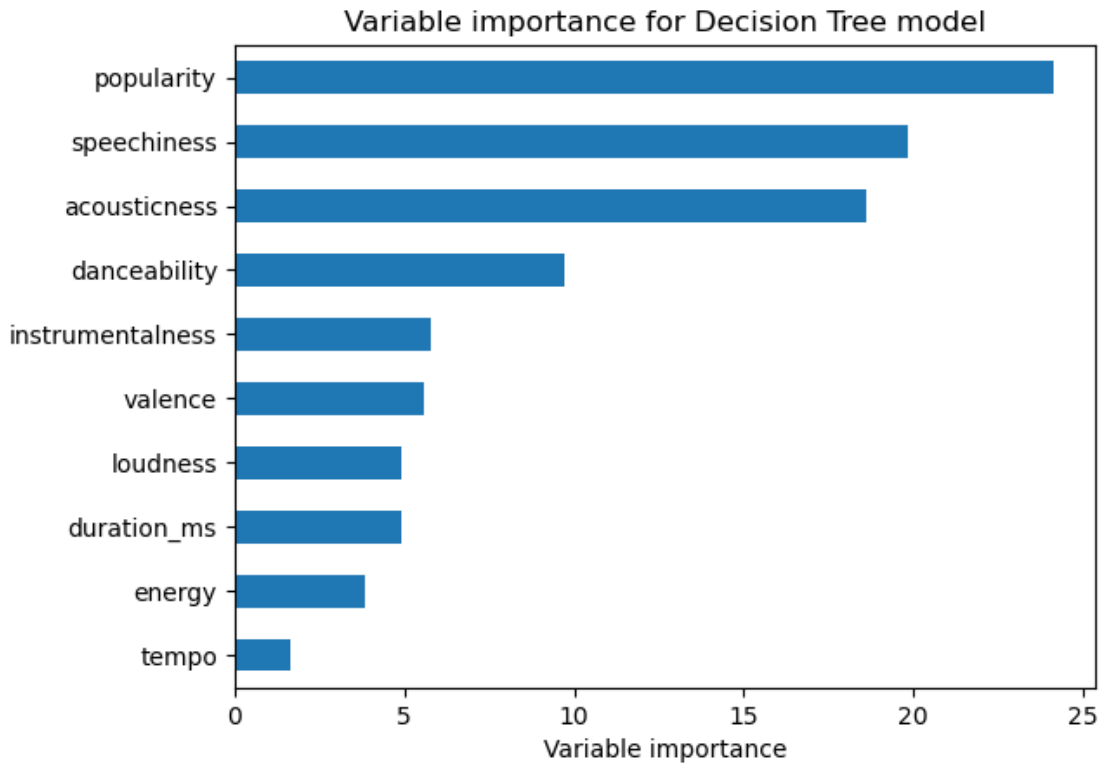
[188]:
```
cm_gb = confusion_matrix(y_test, y_pred_bag_reg)
disp = ConfusionMatrixDisplay(confusion_matrix=cm_gb,␣
 ↪display_labels=class_names)
```

```
fig, ax = plt.subplots(figsize=(8,8))
disp.plot(cmap='Blues', ax=ax, xticks_rotation=90)
plt.title('Confusion Matrix')
plt.show()
```

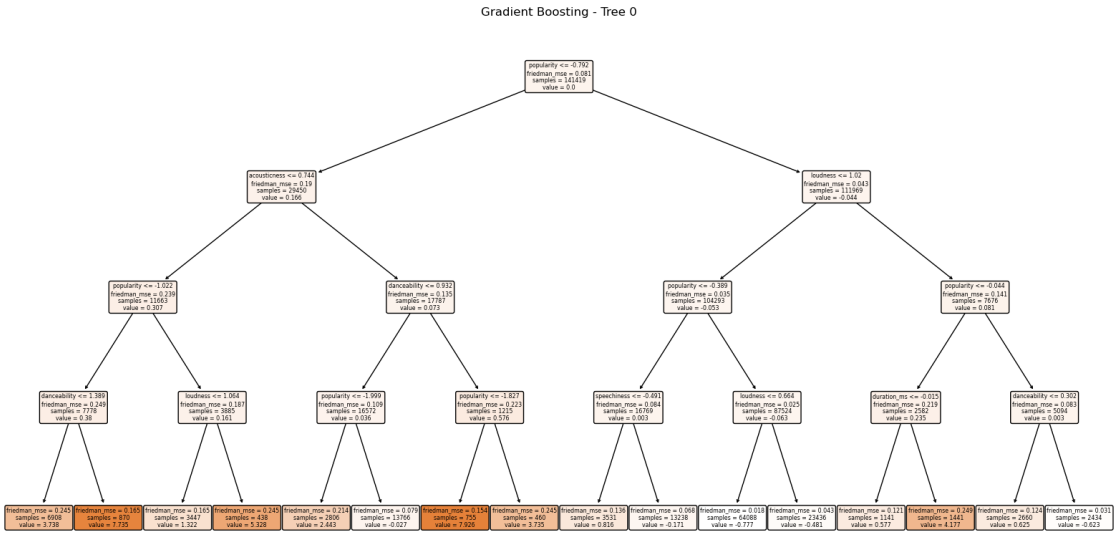


[190]:
```
feature_importance = Tree_Bst_reg.feature_importances_*100
rel_imp = pd.Series(feature_importance, index=numeric_features).
  ↪sort_values(ascending = True,inplace=False)
rel_imp.tail(10).T.plot(kind='barh')
plt.xlabel('Variable importance')
plt.title('Variable importance for Decision Tree model')
```

[190]: Text(0.5, 1.0, 'Variable importance for Decision Tree model')



[192]:
```
class_names = [
    "1: Children/Anime",
    "2: Classical/Opera",
    "3: Country/Folk",
    "4: Dance/Electronic",
    "5: Hip-Hop/Rap/R&B",
    "6: Jazz/Blues",
    "7: Movie/Comedy",
    "8: Pop/Rock",
    "9: World/Soundtrack"
]

plt.figure(figsize=(20, 10))
plot_tree(Tree_Bst_reg.estimators_[0,0], feature_names=numeric_features,␣
 ↪class_names=class_names, filled=True, rounded=True, max_depth=4)
plt.title("Gradient Boosting - Tree 0")
plt.show()
```



Model Parameters: n_estimators=200 learning_rate=0.1 max_depth=4

Model Performance: Accuracy = 63%

Classical/Opera and Movie/Comedy which has a perfect balance between precision and recall, model is both accurate and consistent in identifying this genre.

Dance/Electronic: The interesting thing here is even though the model is exposed to a significantly higher number of Dance/Electronic tracks during training, it does not overfit by assigning this label to ambiguous tracks. Instead, it tries to balance class prediction, maintaining reasonable precision

This indicates the model is not biased towards the majority class.

Pop/Rock has both low precision (47%) and recall (47%) indicating this genre remains hard for the model to classify.

Jazz/Blues has one of the lowest recall scores (46%) many actual Jazz/Blues tracks are being misclassified.

Indicates this genre is still not well captured by the model.

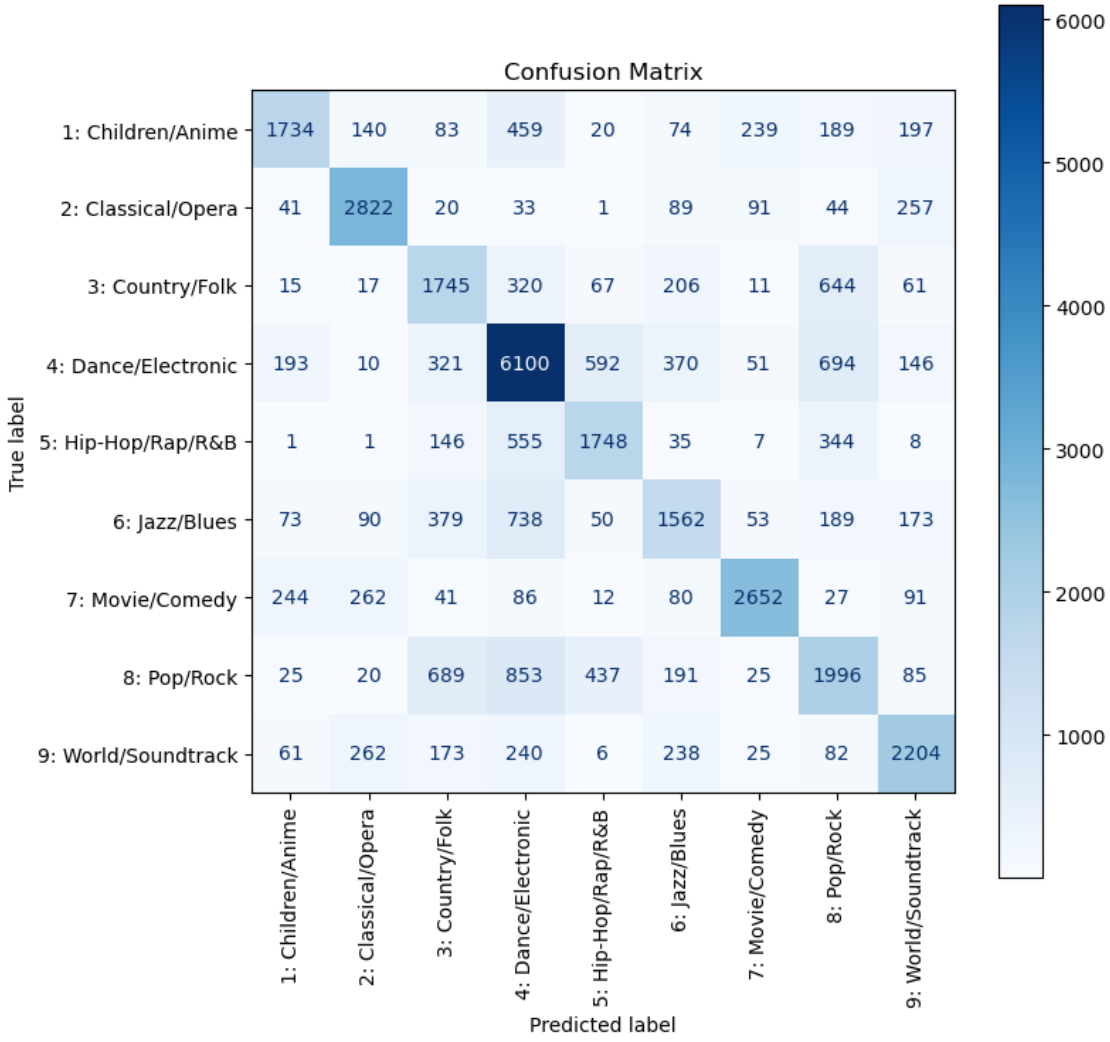
Gradient Boosting achieves the best balance between precision and recall across all most all genres when compared to decision tree and random forest.

[195]:
```
y_multibst = spotify_df['genre_grouped']
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y_multibst)
y_train_ds1 = label_encoder.fit_transform(y_train)
y_test_ds1 = label_encoder.fit_transform(y_test)
X_train, X_test, y_train, y_test = train_test_split(
    df_multi, y_encoded, test_size=0.2, random_state=42, stratify=y_multi
)
```

[207]:
```
import xgboost as xgb

# Create the XGBoost classifier
xgb_clf = xgb.XGBClassifier(
    objective='multi:softmax',
    num_class=9,
    eval_metric='mlogloss',
    use_label_encoder=False,
    n_estimators=250,
    max_depth=5,
    learning_rate=0.1,
    random_state=42
)

# Train the model
start_time = time.time()
xgb_clf.fit(X_train, y_train)
y_pred_xgb = xgb_clf.predict(X_test)
end_time = time.time()
training_duration = end_time - start_time
print(f"Training time (XGBoost): {training_duration:.2f} seconds")

cm_xgb = confusion_matrix(y_test, y_pred_xgb)
disp = ConfusionMatrixDisplay(confusion_matrix=cm_xgb,␣
 ↪display_labels=class_names)
fig, ax = plt.subplots(figsize=(8,8))
disp.plot(cmap='Blues', ax=ax, xticks_rotation=90)
plt.title('Confusion Matrix')
plt.show()

print("\nClassification Report:")
print(classification_report(y_test, y_pred_xgb))
```

```
/opt/anaconda3/lib/python3.12/site-packages/xgboost/training.py:183:
UserWarning: [11:28:39] WARNING:
/Users/runner/work/xgboost/xgboost/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
```

```
Training time (XGBoost): 3.95 seconds
```



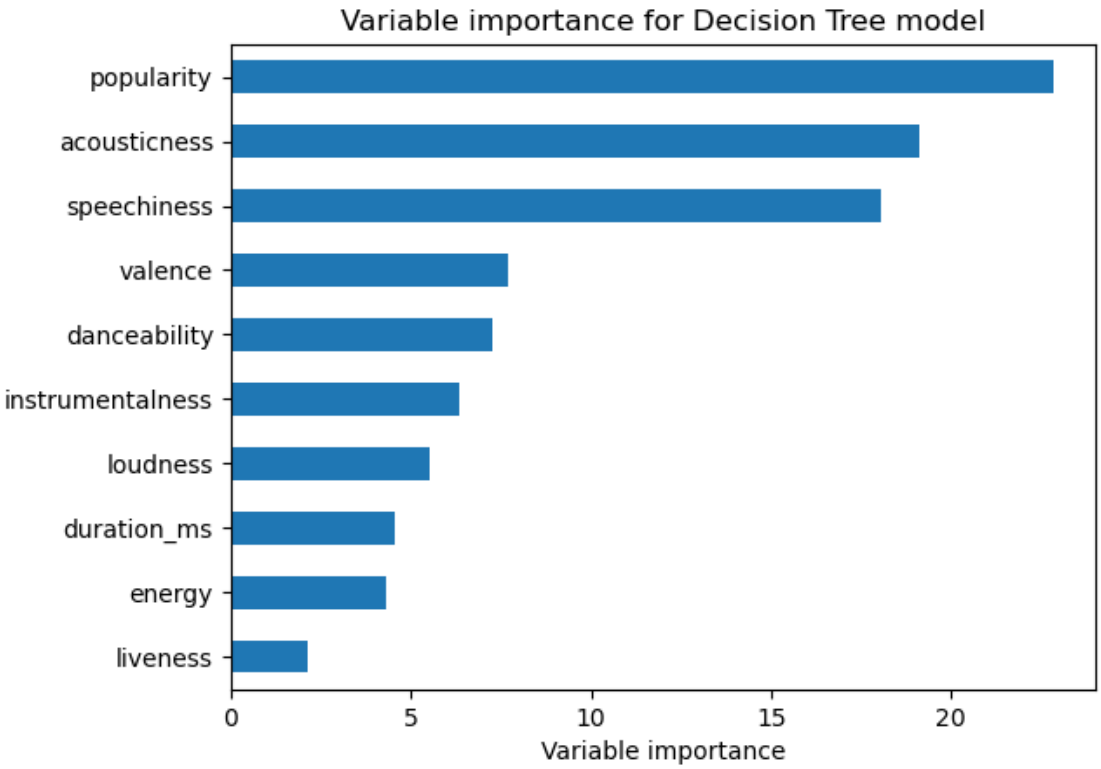
Classification Report:

	precision	recall	f1-score	support
0	0.73	0.55	0.63	3135
1	0.78	0.83	0.80	3398
2	0.49	0.57	0.52	3086
3	0.65	0.72	0.68	8477
4	0.60	0.61	0.61	2845
5	0.55	0.47	0.51	3307
6	0.84	0.76	0.80	3495
7	0.47	0.46	0.47	4321
8	0.68	0.67	0.68	3291
accuracy			0.64	35355

```
   macro avg       0.64      0.63      0.63     35355
weighted avg       0.64      0.64      0.64     35355
```

[209]:
```
feature_importance = xgb_clf.feature_importances_*100
rel_imp = pd.Series(feature_importance, index=numeric_features).
  ↪sort_values(ascending = True,inplace=False)
rel_imp.tail(10).T.plot(kind='barh')
plt.xlabel('Variable importance')
plt.title('Variable importance for Decision Tree model')
```

[209]:
```
Text(0.5, 1.0, 'Variable importance for Decision Tree model')
```



0.5 KNN

[85]:
```
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(
    df_multi, y_multi, test_size=0.2, random_state=42, stratify=y_multi
)
```

[87]:
```
from sklearn.neighbors import KNeighborsClassifier
```

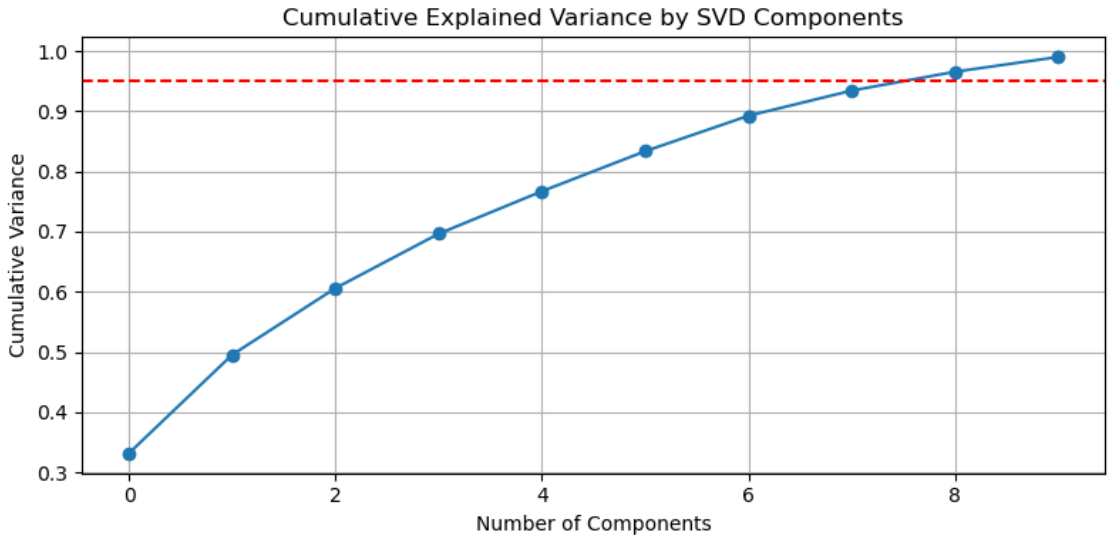
```
knn = KNeighborsClassifier(n_neighbors=5, metric="euclidean", weights="uniform", p=2)  # Euclidean distance
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
print(classification_report(y_test, y_pred_knn))
```

```
              precision    recall  f1-score   support

           0       0.55      0.56      0.56      3135
           1       0.69      0.79      0.74      3398
           2       0.37      0.51      0.43      3086
           3       0.60      0.69      0.64      8477
           4       0.55      0.49      0.52      2845
           5       0.48      0.38      0.43      3307
           6       0.83      0.71      0.76      3495
           7       0.41      0.31      0.36      4321
           8       0.61      0.56      0.58      3291

    accuracy                           0.57     35355
   macro avg       0.57      0.56      0.56     35355
weighted avg       0.57      0.57      0.57     35355
```

[89]:
```
from sklearn.decomposition import PCA, TruncatedSVD

svd = TruncatedSVD(n_components=df_multi.shape[1] - 1, random_state=42)
svd.fit(df_multi)
```

[89]:
```
TruncatedSVD(n_components=10, random_state=42)
```

[91]:
```
explained_var = np.cumsum(svd.explained_variance_ratio_)
plt.figure(figsize=(8, 4))
plt.plot(explained_var, marker='o')
plt.axhline(y=0.95, color='r', linestyle='--')
plt.title("Cumulative Explained Variance by SVD Components")
plt.xlabel("Number of Components")
plt.ylabel("Cumulative Variance")
plt.grid(True)
plt.tight_layout()
plt.show()
```



[93]:
```
pca = PCA(n_components=7, random_state=42)
X_train_pca = pca.fit_transform(df_multi)
```

[95]:
```
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(
    X_train_pca, y_multi, test_size=0.2, random_state=42, stratify=y_multi
)
```

[97]:
```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5, metric="euclidean", weights=␣
 ↪"uniform", p=2)  # Euclidean distance
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
print(classification_report(y_test, y_pred_knn))
```

```
              precision    recall  f1-score   support

           0       0.55      0.56      0.56      3135
           1       0.69      0.79      0.74      3398
           2       0.37      0.51      0.43      3086
           3       0.60      0.69      0.64      8477
           4       0.55      0.49      0.52      2845
           5       0.48      0.38      0.43      3307
           6       0.83      0.71      0.76      3495
           7       0.41      0.31      0.36      4321
           8       0.61      0.56      0.58      3291

    accuracy                           0.57     35355
   macro avg       0.57      0.56      0.56     35355
```

```
weighted avg       0.57      0.57      0.57     35355
```

After applying PCA with 7 components training is done using KNN classifier. The accuracy dropped from 57% to 53% compared to the original KNN model (without PCA). This indicates the reduced feature space didn't preserve enough information for genre classification.

[ ]: