COGS 118A- Project Proposal

Team

- Xiaoxuan Zhang
- Yunxiang Chi
- Xiaoyan He
- Jiayi Dong
- Elaine (Yilin) Ge

Abstract

The project is designed to be a **Music Genre Classification Tool**.

Our study aims to develop an advanced music genre classification system with the objective of enhancing personalized recommendation mechanisms and user experience. The comprehensive classification model is capable of accommodating a wide spectrum of music genres, such as rock, pop, jazz, hip-hop, and electronic, among others. The dataset comprises audio recordings of songs, evaluated through an array of attributes such as spectral information, rhythmic patterns, and harmonic characteristics, along with mean and variance of multiple features in CSV files. These extracted features serve as input for our machine learning models. Employing algorithms like K-Nearest Neighbors (KNN), Support Vector Machines (SVM), and Neural Networks, the research exploits a heterogeneous dataset of labeled music samples spanning various genres. The efficacy of the proposed music genre classification system is gauged through numerous evaluation metrics, including accuracy, precision, recall, F1 score, and confusion matrix. These quantitative assessments shed light on the models' classification performance and their capability to accurately predict music genres, enabling the selection of the most efficient model. By demonstrating high accuracy and robustness, our tool exhibits its effectiveness and dependability, promising significant improvements in music recommendation systems.

Background

Music genres are categories that have arisen through a complex interplay of cultures, artists, and market forces to characterize differences between compositions and performances. They can reflect the cultural origins of the music, the style, the instrumentation used, the mood it conveys, or even the time period when it was produced. For example, the genre of classical music, the more ancient style, is known for its sophisticated orchestral composition and large musical scales. On the other hand, genres

like rock and pop, which emerged more recently, have their distinct characteristics: rock typically features a strong backbeat along with electric guitars, while pop music often has catchy and repetitive melodies and a verse-chorus structure.

Our project is focused on developing a state-of-the-art music genre classification system that optimizes the way music platforms and music companies design their personalized recommendation systems to improve overall music listening experience and user experience through a more precise way of avigating and discovering music.

To achieve this, we employ advanced machine learning algorithms and data analysis techniques, leveraging a diverse dataset of labeled music samples spanning different genres with various features. We will also compare among different algoritms' performances to ensure the system's accuracy, robustness, and ability to generalize across a wide range of music styles.

Problem Statement

The problem our project seeks to address is the accurate categorization of music tracks into their respective genres. Precise music genre classification is a complex and multifaceted challenge due to the inherent subjectivity involved in music perception and the overlapping elements present across various genres. Yet, accurate genre classification is paramount to improving user experience on music platforms, as it plays a crucial role in personalized music recommendation systems.

Our project tackles this problem through the lens of supervised machine learning. With a dataset of songs with labeled genres and an array of song features, we aim to train a model that can accurately predict the genre of a song, even with unseen data. Our goal is to devise a machine learning model that optimizes precision and generalization while minimizing overfitting.

Data

The dataset we are using is GTZAN Dataset - Music Genre Classification [1]

- https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification
- Content:
  - original soundfile: Collection of 10 genres with 100 30-seconds audio files each.
  - images: Power Spectra of each audio files in image formats.
  - 2 csv: Containing multiple features of all the songs.

For our model, we design to use both csv files, which have the following 10 variables (with both mean and variance) as well as 1000 observations:

- pitch (chroma in the dataset)
- RMS in audio signals
- spectral centroid
- spectral bandwidth
- roll-off frequency
- zero crossing rate
- harmony
- perceptron
- tempo
- 20 groups of Mel Frequency Cepstral Coefficients

Special note about our dataset

Worth mentioning point on our dataset, because there are two csv files in the GTZAN Dataset - Music Genre Classification dataset[[1]], we have the chance to choose from a dataset, which contains `30-seconds song features` with 1000 rows and `3-seconds song features` with 9990 rows, equivalent to cutting every song into 10 pieces from the 30-seconds song features dataset. Since the sequence and time features in the songs are crucial components, we started with the `30-seconds song features` dataset. However, at the very beginning of the algorithm implementation, our group jumped back to the topic and re-evaluated which dataset may lead to better performance in the end. Due to the fact that the dataset contains 20 sets of Mel Frequency Cepstral Coefficients (MFCCs) features (*20*2* columns with mean and variation for each time snippet), which contains the information about time, we decided to switch to the `3-seconds song features` dataset, trading 1000 rows of data to 9990 rows, in order to achieve better overall performance.

Library setups

In [1]:
```
# Please uncomment the following lines if libraries are missing
# # %pip install --upgrade pip
# %pip install pandas
# %pip install numpy
# %pip install matplotlib
# %pip install seaborn
# %pip install scipy
# %pip install scikit-learn
# %pip install imblearn
```

In [2]:
```
# EDA
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from mpl_toolkits.mplot3d import Axes3D
```

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# knn
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import BaggingClassifier

# grid search
from sklearn.model_selection import StratifiedKFold, GridSearchCV, cross_val_s
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error, classification_report, confusi
from math import sqrt

# svm
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import BaggingClassifier, AdaBoostClassifier
from sklearn.feature_selection import RFE
from imblearn.over_sampling import SMOTE

# rnn
import torch
from torch import nn
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader

# cnn
from sklearn.ensemble import BaggingRegressor
from tqdm.auto import tqdm
```

Exploratory Data Analysis

Loading Data

In [3]:
```
df_raw = pd.read_csv("Data/features_3_sec.csv", sep=",")
df_raw = df_raw.drop(columns='filename')
df_raw = df_raw.drop(columns='length')
df_raw.head()
```

Out[3]:

	chroma_stft_mean	chroma_stft_var	rms_mean	rms_var	spectral_centroid_mean	s
0	0.335406	0.091048	0.130405	0.003521	1773.065032	
1	0.343065	0.086147	0.112699	0.001450	1816.693777	
2	0.346815	0.092243	0.132003	0.004620	1788.539719	
3	0.363639	0.086856	0.132565	0.002448	1655.289045	
4	0.335579	0.088129	0.143289	0.001701	1630.656199	

5 rows × 58 columns

Statistics

In [4]:
```
df_raw.describe()
```

Out[4]:

	chroma_stft_mean	chroma_stft_var	rms_mean	rms_var	spectral_centro
count	9990.000000	9990.000000	9990.000000	9.990000e+03	999(
mean	0.379534	0.084876	0.130859	2.676388e-03	219
std	0.090466	0.009637	0.068545	3.585628e-03	75
min	0.107108	0.015345	0.000953	4.379535e-08	47
25%	0.315698	0.079833	0.083782	6.145900e-04	163(
50%	0.384741	0.085108	0.121253	1.491318e-03	220{
75%	0.442443	0.091092	0.176328	3.130862e-03	271:
max	0.749481	0.120964	0.442567	3.261522e-02	5432

8 rows × 57 columns

Missing Values

In [5]:
```
null_value_df = df_raw.isnull().sum()
print(null_value_df.unique().item(), null_value_df.sum())
```

```
0 0
```

There is no missing value found in the raw dataset.

Outliers

In some cases, cleaning up some outliers is part of the EDA process. However, after our group discussion and related knowledge on the topic of music classification, we decided to keep the outliers in the dataset, which may help the future models to be more robust.

In [6]:
```
# numeric_cols = df_raw.select_dtypes(include=[np.number]).columns

# z_scores = np.abs(stats.zscore(df_raw[numeric_cols]))
# threshold = 3

# # Remove rows with outliers
# df = df_raw[(z_scores < threshold).all(axis=1)]

df = df_raw
df_label = df.filter(regex=r'label')
```

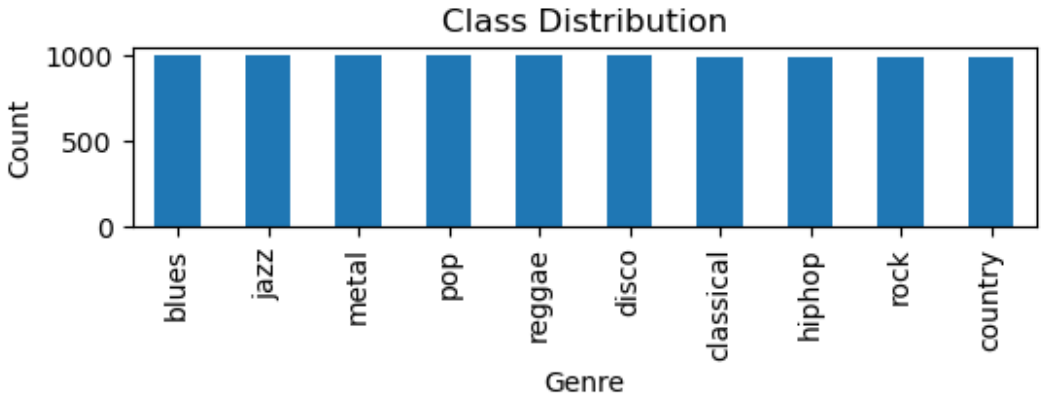
In [7]:
```
df_label['label'].unique()
```

Out[7]:
```
array(['blues', 'classical', 'country', 'disco', 'hiphop', 'jazz',
       'metal', 'pop', 'reggae', 'rock'], dtype=object)
```

Class Distribution

Check if the raw dataset contains balanced amount of all the genres that we are going to train.

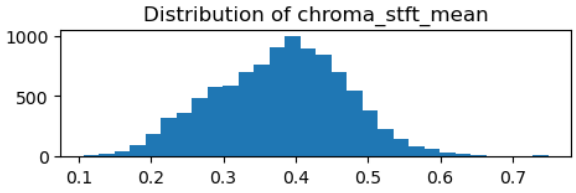
In [8]:
```
plt.figure(figsize=(6,1.2))
df['label'].value_counts().plot(kind='bar')
plt.title('Class Distribution')
plt.xlabel('Genre')
plt.ylabel('Count')
plt.show()
```

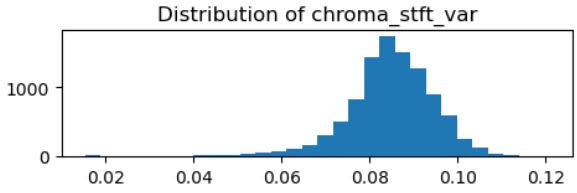


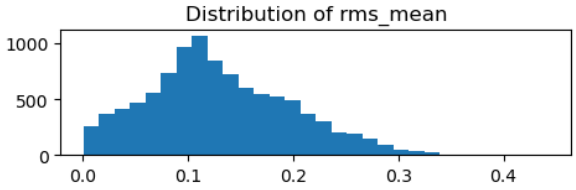
Each class has exactly the same number of datapoints.

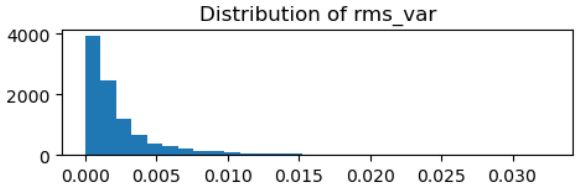
Data Visualization

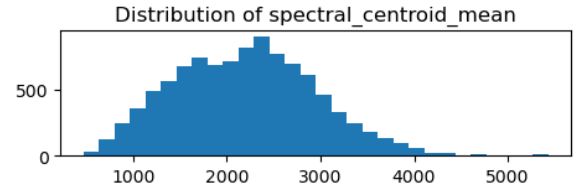
In [9]:
```
# Grouped histograms for each feature mean and variance
mean_cols = [col for col in df.columns if 'mean' in col]
var_cols = [col for col in df.columns if 'var' in col]

fig, axs = plt.subplots(len(mean_cols), 2, figsize=(10, 50))

for i, col in enumerate(mean_cols):
    axs[i, 0].hist(df[col], bins=30)
    axs[i, 0].set_title(f'Distribution of {col}')

for i, col in enumerate(var_cols):
    axs[i, 1].hist(df[col], bins=30)
    axs[i, 1].set_title(f'Distribution of {col}')

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

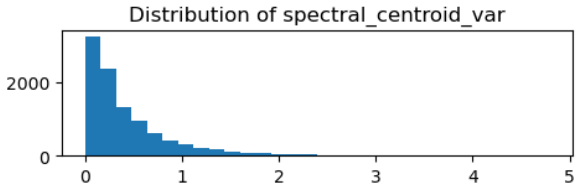

Distribution of chroma_stft_mean
1000
500
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7

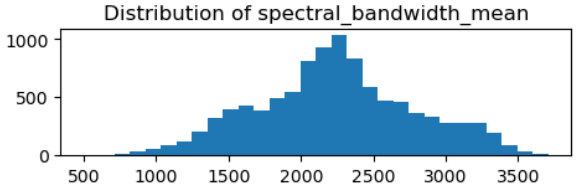

Distribution of chroma_stft_var
1000
0
0.02
0.04
0.06
0.08
0.10
0.12

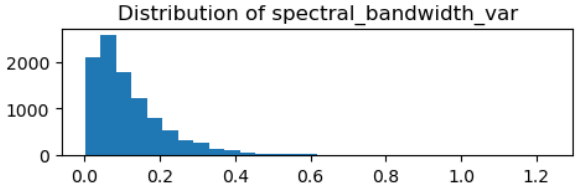

Distribution of rms_mean
1000
500
0
0.0
0.1
0.2
0.3
0.4

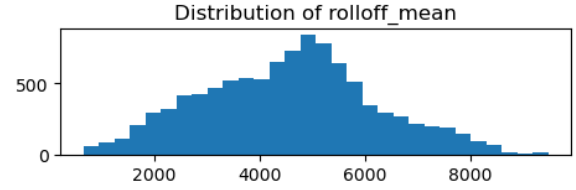

Distribution of rms_var
4000
2000
0
0.000
0.005
0.010
0.015
0.020
0.025
0.030

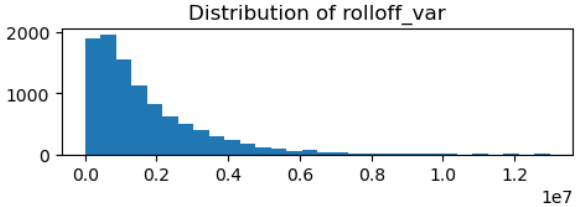

Distribution of spectral_centroid_mean
500
0
1000
2000
3000
4000
5000

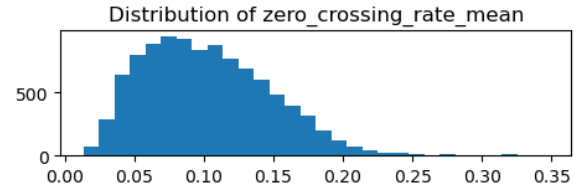

Distribution of spectral_centroid_var
2000
0
0
1
2
3
4
5
1e6

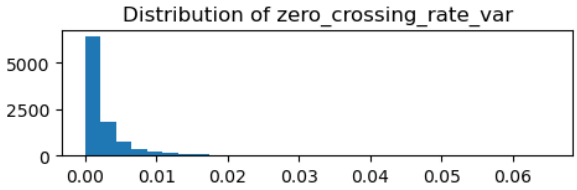

Distribution of spectral_bandwidth_mean
1000
500
0
500
1000
1500
2000
2500
3000
3500

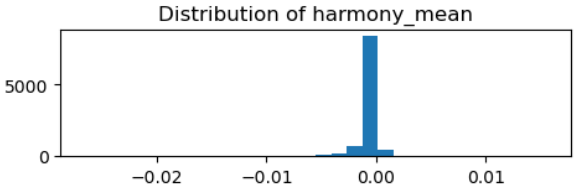

Distribution of spectral_bandwidth_var
2000
1000
0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1e6

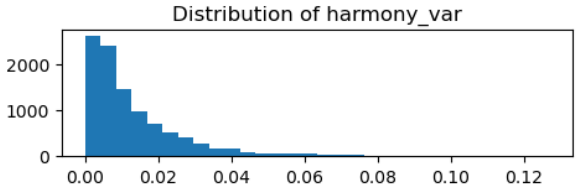

Distribution of rolloff_mean
500
0
2000
4000
6000
8000

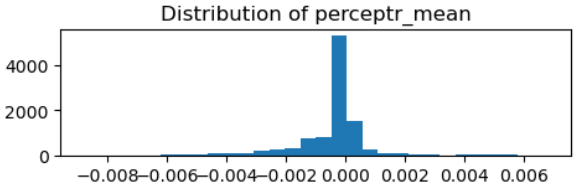

Distribution of rolloff_var
2000
1000
0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1e7

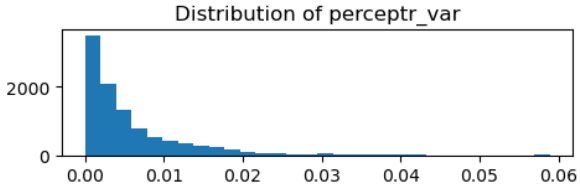

Distribution of zero_crossing_rate_mean
500
0
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35

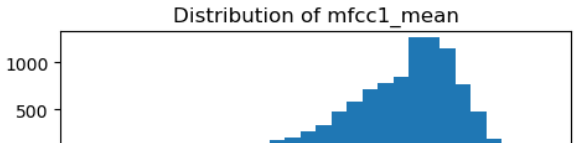

Distribution of zero_crossing_rate_var
5000
2500
0
0.00
0.01
0.02
0.03
0.04
0.05
0.06

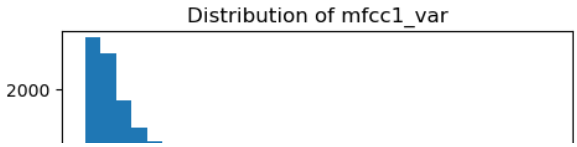

Distribution of harmony_mean
5000
0
−0.02
−0.01
0.00
0.01

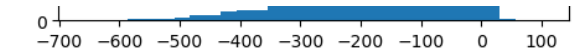

Distribution of harmony_var
2000
1000
0
0.00
0.02
0.04
0.06
0.08
0.10
0.12

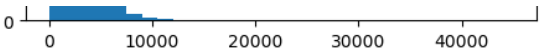

Distribution of perceptr_mean
4000
2000
0
−0.008
−0.006
−0.004
−0.002
0.000
0.002
0.004
0.006

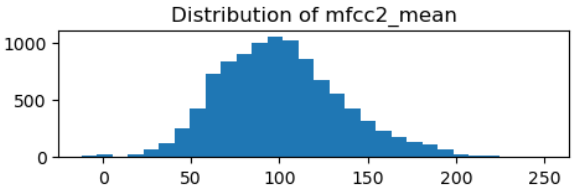

Distribution of perceptr_var
2000
0
0.00
0.01
0.02
0.03
0.04
0.05
0.06

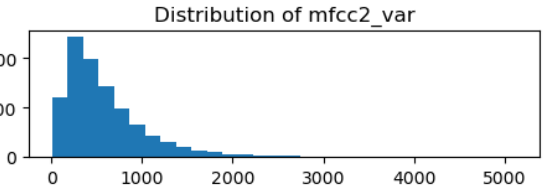

Distribution of mfcc1_mean
1000
500

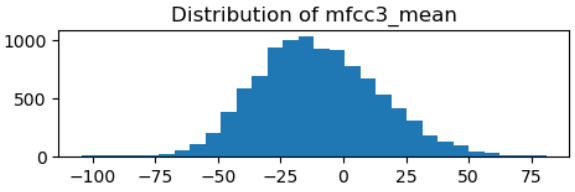

Distribution of mfcc1_var
2000

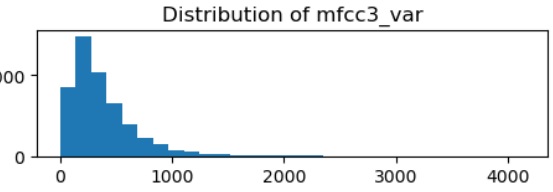

0
−700 −600 −500 −400 −300 −200 −100 0 100

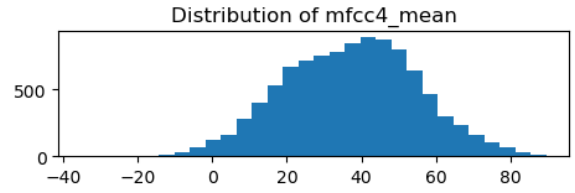

0
0 10000 20000 30000 40000

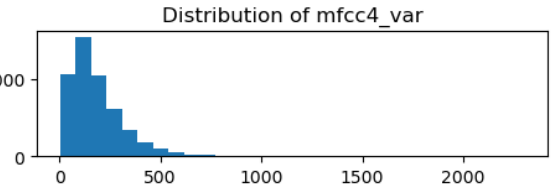

Distribution of mfcc2_mean
1000
500
0
0 50 100 150 200 250

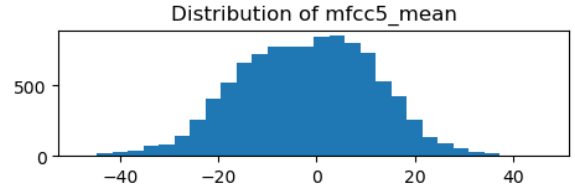

Distribution of mfcc2_var
2000
1000
0
0 1000 2000 3000 4000 5000

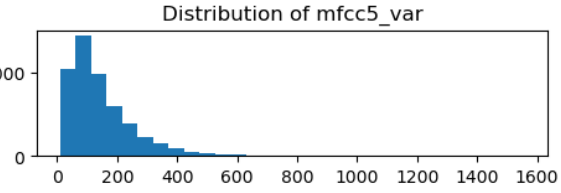

Distribution of mfcc3_mean
1000
500
0
−100 −75 −50 −25 0 25 50 75

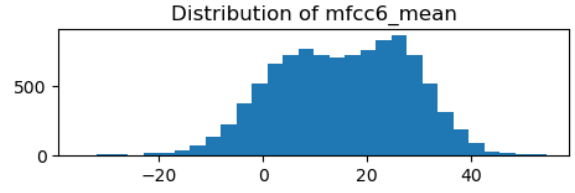

Distribution of mfcc3_var
2000
0
0 1000 2000 3000 4000

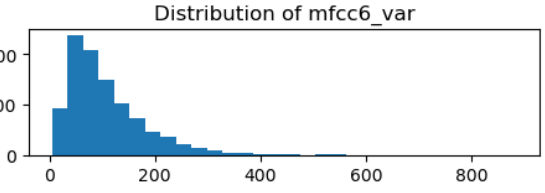

Distribution of mfcc4_mean
500
0
−40 −20 0 20 40 60 80

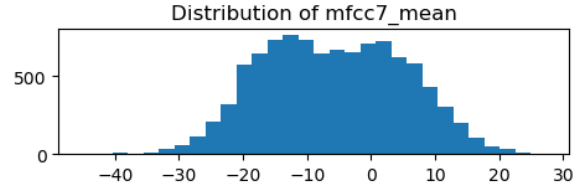

Distribution of mfcc4_var
2000
0
0 500 1000 1500 2000

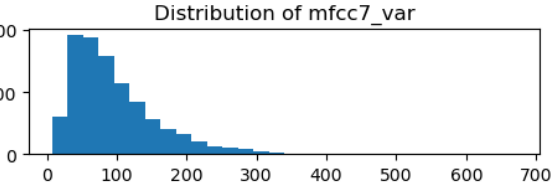

Distribution of mfcc5_mean
500
0
−40 −20 0 20 40

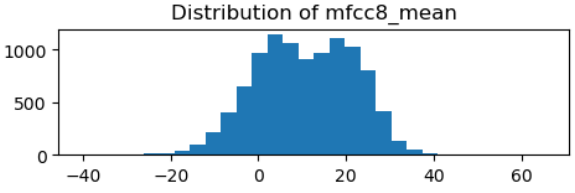

Distribution of mfcc5_var
2000
0
0 200 400 600 800 1000 1200 1400 1600

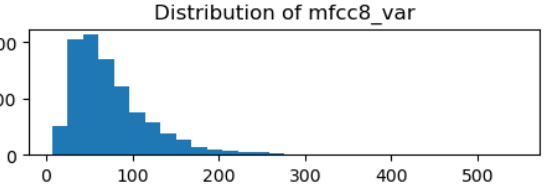

Distribution of mfcc6_mean
500
0
−20 0 20 40

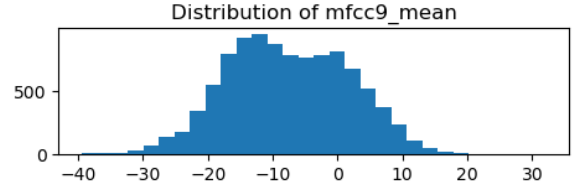

Distribution of mfcc6_var
2000
1000
0
0 200 400 600 800

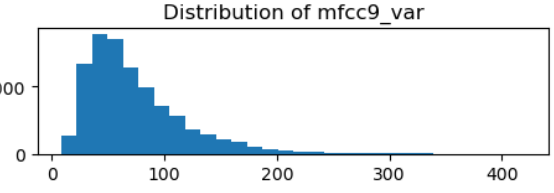

Distribution of mfcc7_mean
500
0
−40 −30 −20 −10 0 10 20 30


Distribution of mfcc7_var
2000
1000
0
0 100 200 300 400 500 600 700

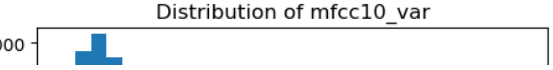

Distribution of mfcc8_mean
1000
500
0
−40 −20 0 20 40 60

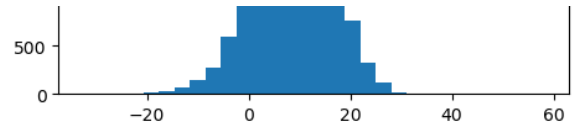

Distribution of mfcc8_var
2000
1000
0
0 100 200 300 400 500

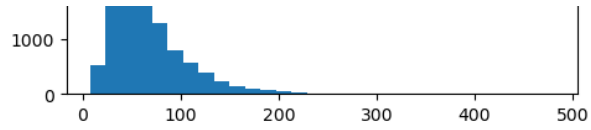

Distribution of mfcc9_mean
500
0
−40 −30 −20 −10 0 10 20 30

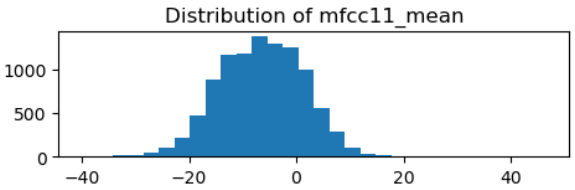

Distribution of mfcc9_var
1000
0
0 100 200 300 400

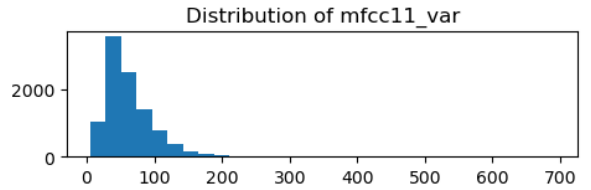

Distribution of mfcc10_mean
1000

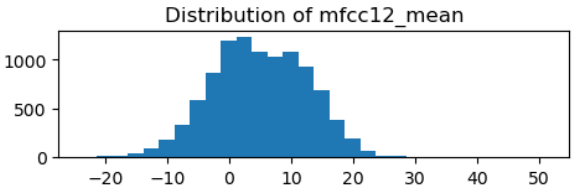

Distribution of mfcc10_var
2000

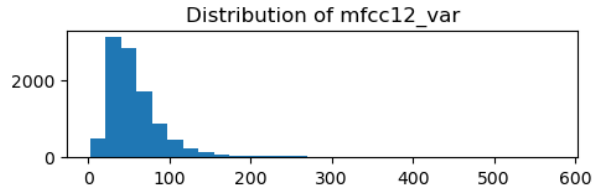

500
0
−20
0
20
40
60

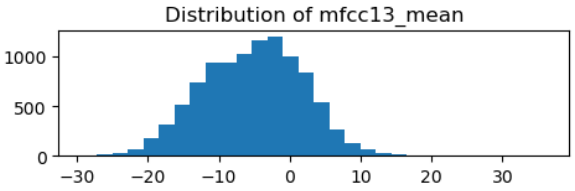

1000
0
0
100
200
300
400
500

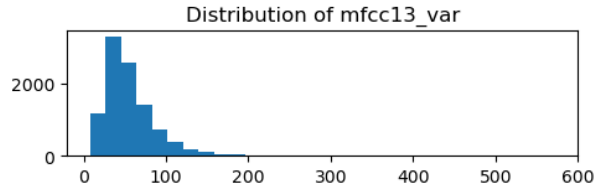

Distribution of mfcc11_mean
1000
500
0
−40
−20
0
20
40

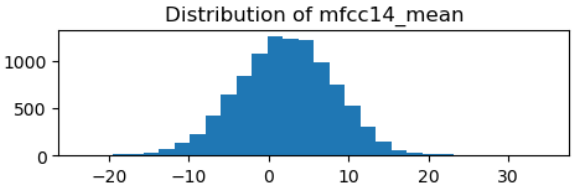

Distribution of mfcc11_var
2000
0
0
100
200
300
400
500
600
700

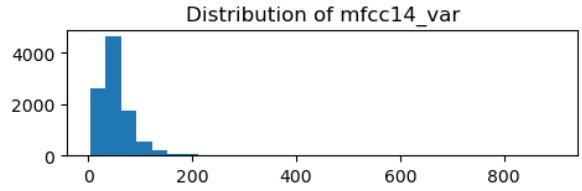

Distribution of mfcc12_mean
1000
500
0
−20
−10
0
10
20
30
40
50

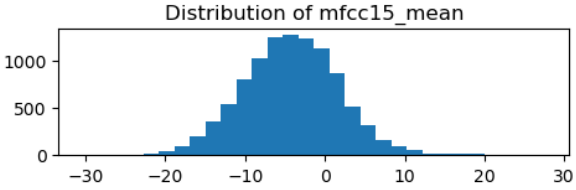

Distribution of mfcc12_var
2000
0
0
100
200
300
400
500
600

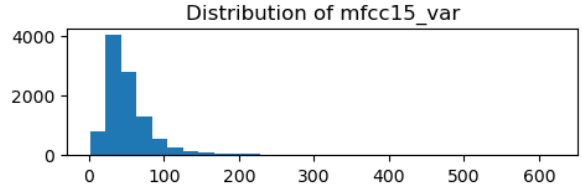

Distribution of mfcc13_mean
1000
500
0
−30
−20
−10
0
10
20
30

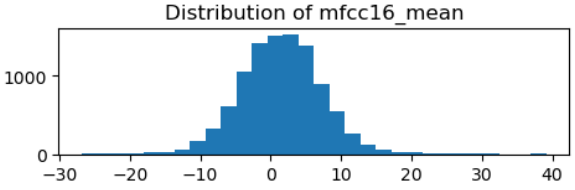

Distribution of mfcc13_var
2000
0
0
100
200
300
400
500
600

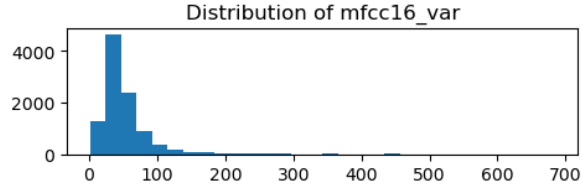

Distribution of mfcc14_mean
1000
500
0
−20
−10
0
10
20
30

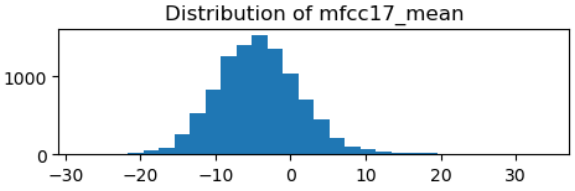

Distribution of mfcc14_var
4000
2000
0
0
200
400
600
800

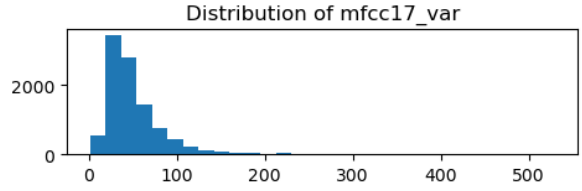

Distribution of mfcc15_mean
1000
500
0
−30
−20
−10
0
10
20
30

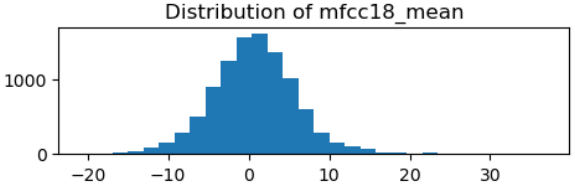

Distribution of mfcc15_var
4000
2000
0
0
100
200
300
400
500
600

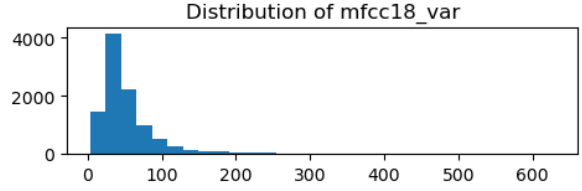

Distribution of mfcc16_mean
1000
0
−30
−20
−10
0
10
20
30
40

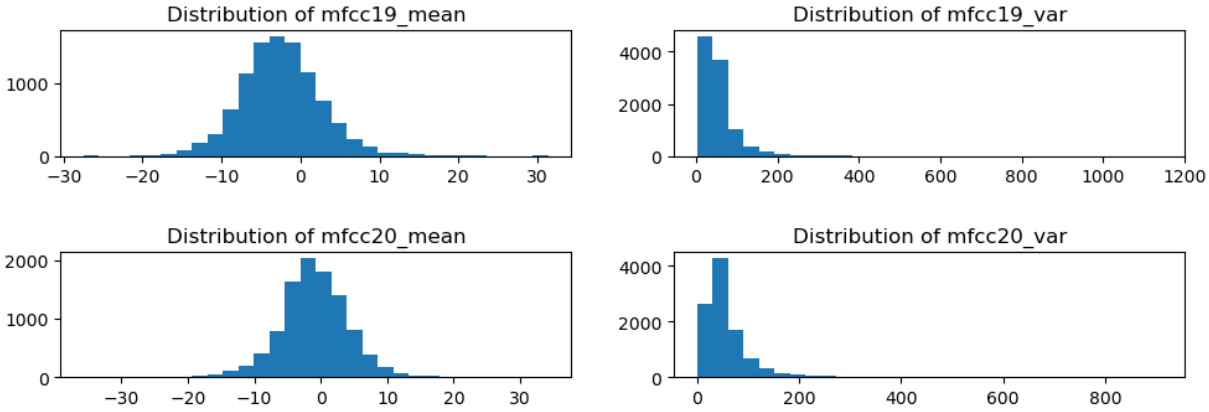

Distribution of mfcc16_var
4000
2000
0
0
100
200
300
400
500
600
700


Distribution of mfcc17_mean
1000
0
−30
−20
−10
0
10
20
30


Distribution of mfcc17_var
2000
0
0
100
200
300
400
500


Distribution of mfcc18_mean
1000
0
−20
−10
0
10
20
30


Distribution of mfcc18_var
4000
2000
0
0
100
200
300
400
500
600



After the visualization of the dataframe, we can see that harmony and perceptr have very centralized mean. This indicates that they might not be very impactful to our model, we decided to drop these two columns.

In [10]:
```
drop_four = ["harmony_mean", "harmony_var", "perceptr_mean", "perceptr_var"]
df = df.drop(labels=drop_four, axis=1)
df.head()
```

Out[10]:

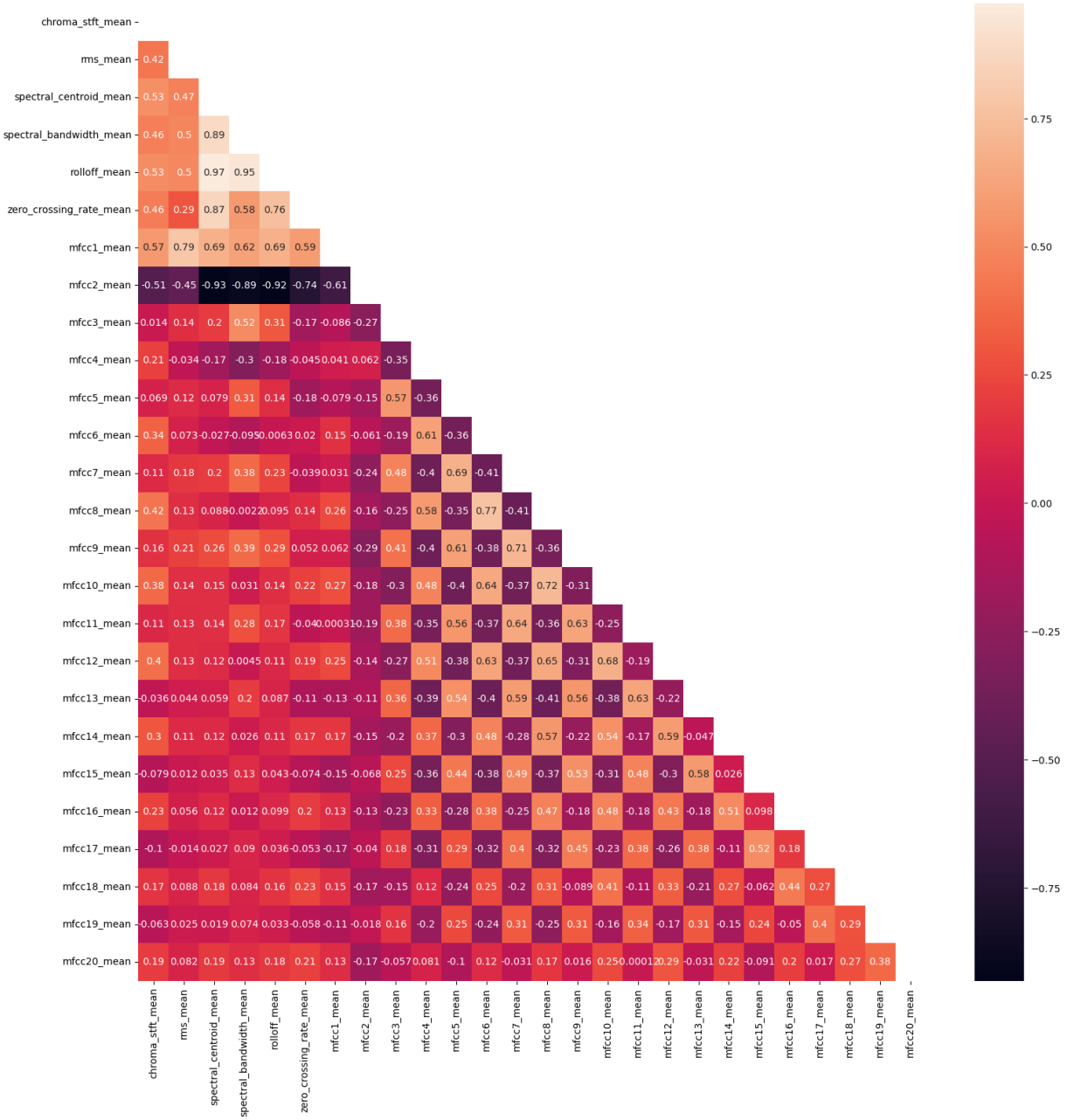
	chroma_stft_mean	chroma_stft_var	rms_mean	rms_var	spectral_centroid_mean	s
0	0.335406	0.091048	0.130405	0.003521	1773.065032	
1	0.343065	0.086147	0.112699	0.001450	1816.693777	
2	0.346815	0.092243	0.132003	0.004620	1788.539719	
3	0.363639	0.086856	0.132565	0.002448	1655.289045	
4	0.335579	0.088129	0.143289	0.001701	1630.656199	

5 rows × 54 columns

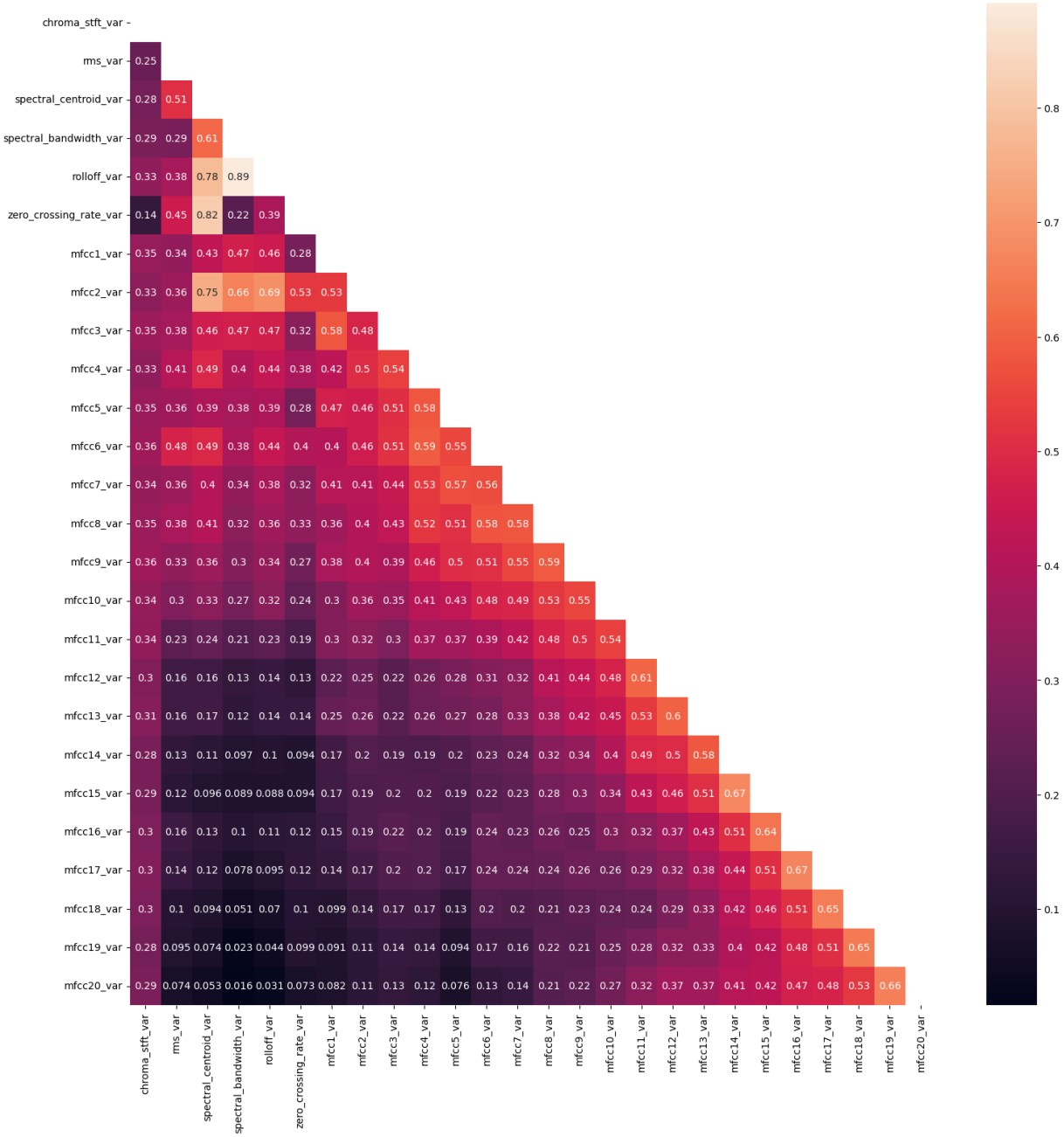
Correlation Analysis

- Plotting out the overall correlation matrix

In [11]:
```
# Calculate correlation matrix
corr_matrix_means = df[[col for col in df.columns if 'mean' in col]].corr()
corr_matrix_vars = df[[col for col in df.columns if 'var' in col]].corr()

# Plot heatmap of correlation matrix
mask = np.triu(np.ones_like(corr_matrix_means, dtype=bool))
plt.figure(figsize=(18,18))
sns.heatmap(corr_matrix_means, mask=mask, annot=True)
plt.show()
```



In [12]:
```
plt.figure(figsize=(18,18))
mask = np.triu(np.ones_like(corr_matrix_vars, dtype=bool))
sns.heatmap(corr_matrix_vars, mask=mask, annot=True)
plt.show()
```



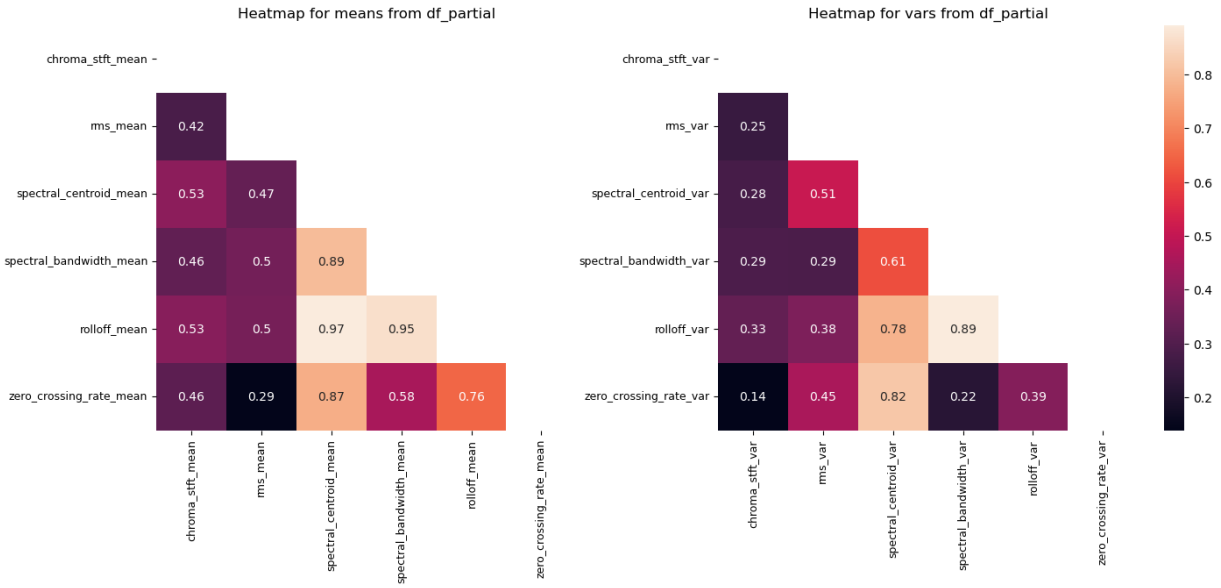
Because Mel Frequency Cepstral Coefficients (MFCCs) got seperated into 20 parts in the raw dataset, while contributing to the same feature, we decided to draw two correlation matrices by seperating mfcc with other features for *mean* and *var* respectively.

In [13]:
```
# Prepare the partial dataframe with MFCCs
df_mfcc = df.filter(regex=r'(mfcc*|label)')

# Prepare the partial dataframe without MFCCs
df_partial = df.drop(labels=df.filter(regex=r'mfcc*').columns, axis=1)

df_mfcc.shape, df_partial.shape
```

Out[13]:
```
((9990, 41), (9990, 14))
```

In [14]:
```
# Calculate correlation matrix
corr_matrix_means_partial = df_partial.filter(regex=r'_mean').corr()
corr_matrix_vars_partial = df_partial.filter(regex=r'_var').corr()
```

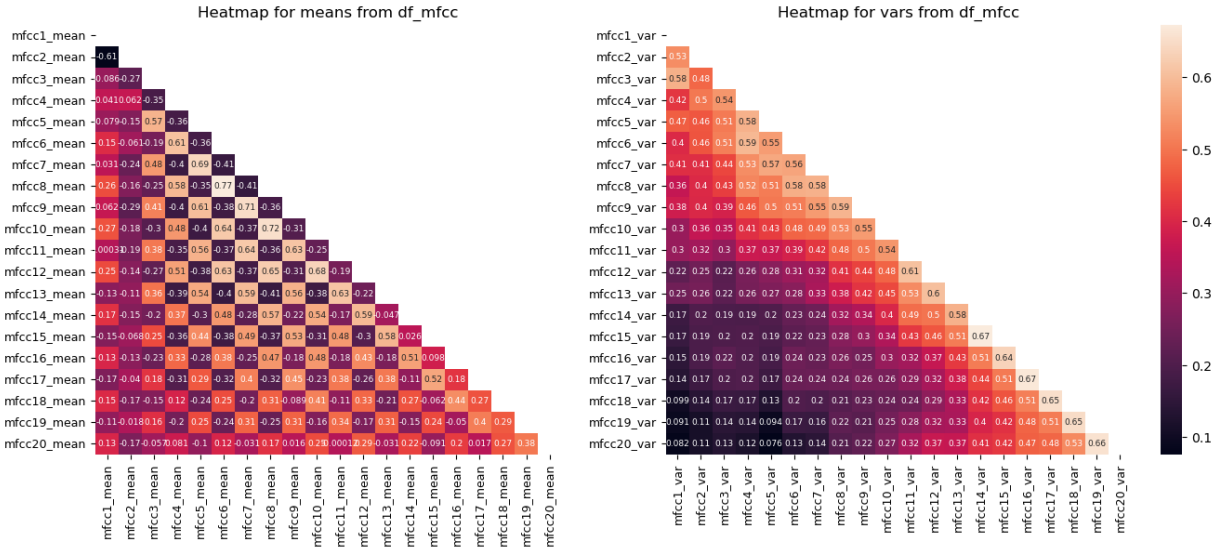
```
# Plot heatmap of correlation matrix
fig, axs = plt.subplots(ncols=2, gridspec_kw={'width_ratios': [0.8, 1]})
fig = plt.gcf()
fig.set_size_inches(16, 6)
plt.subplots_adjust(wspace=0.3)

# Plot the first heatmap on the left subplot
mask1 = np.triu(np.ones_like(corr_matrix_means_partial, dtype=bool))
fig1 = sns.heatmap(corr_matrix_means_partial, ax=axs[0], cbar=False, annot=True
fig1.set_xticklabels(fig1.get_yticklabels(), fontsize=9)
fig1.set_yticklabels(fig1.get_yticklabels(), fontsize=9)
fig1.set_title('Heatmap for means from df_partial')

# Plot the second heatmap on the right subplot
mask2 = np.triu(np.ones_like(corr_matrix_vars_partial, dtype=bool))
fig2 = sns.heatmap(corr_matrix_vars_partial, ax=axs[1], cbar=True, annot=True,
fig2.set_xticklabels(fig2.get_yticklabels(), fontsize=9)
fig2.set_yticklabels(fig2.get_yticklabels(), fontsize=9) # rotation=-30
fig2.set_title('Heatmap for vars from df_partial')

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



Based on the data we have, since spectral centroid and spectral bandwidth are related, it's not suprised to see their high correlation. But we also found the there's relatively high correlation in between `spectral centroid` & `zero crossing rate`, `rolloff` & `spectral centroid`, `rolloff` & `spectral bandwidth`, `rolloff` & `zero crossing rate`, which are worth discovering more.

In [15]:
```
# Calculate correlation matrix
corr_matrix_means_mfcc = df_mfcc.filter(regex=r'_mean').corr()
corr_matrix_vars_mfcc = df_mfcc.filter(regex=r'_var').corr()


# Plot heatmap of correlation matrix
```

```
fig, axs = plt.subplots(ncols=2, gridspec_kw={'width_ratios': [0.8, 1]})
fig = plt.gcf()
fig.set_size_inches(16, 6)
plt.subplots_adjust(wspace=0.2)

# Plot the first heatmap on the left subplot
mask1 = np.triu(np.ones_like(corr_matrix_means_mfcc, dtype=bool))
fig1 = sns.heatmap(corr_matrix_means_mfcc, ax=axs[0], cbar=False, annot=True, a
fig1.set_xticklabels(fig1.get_yticklabels(), fontsize=9)
fig1.set_yticklabels(fig1.get_yticklabels(), fontsize=9)
fig1.set_title('Heatmap for means from df_mfcc')

# Plot the second heatmap on the right subplot
mask2 = np.triu(np.ones_like(corr_matrix_vars_mfcc, dtype=bool))
fig2 = sns.heatmap(corr_matrix_vars_mfcc, ax=axs[1], cbar=True, annot=True, ann
fig2.set_xticklabels(fig2.get_yticklabels(), fontsize=9)
fig2.set_yticklabels(fig2.get_yticklabels(), fontsize=9) # rotation=-30
fig2.set_title('Heatmap for vars from df_mfcc')

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



Could have more explaination of the pattern here. Don't really know what mfcc is.

- Plotting out the correlation by label

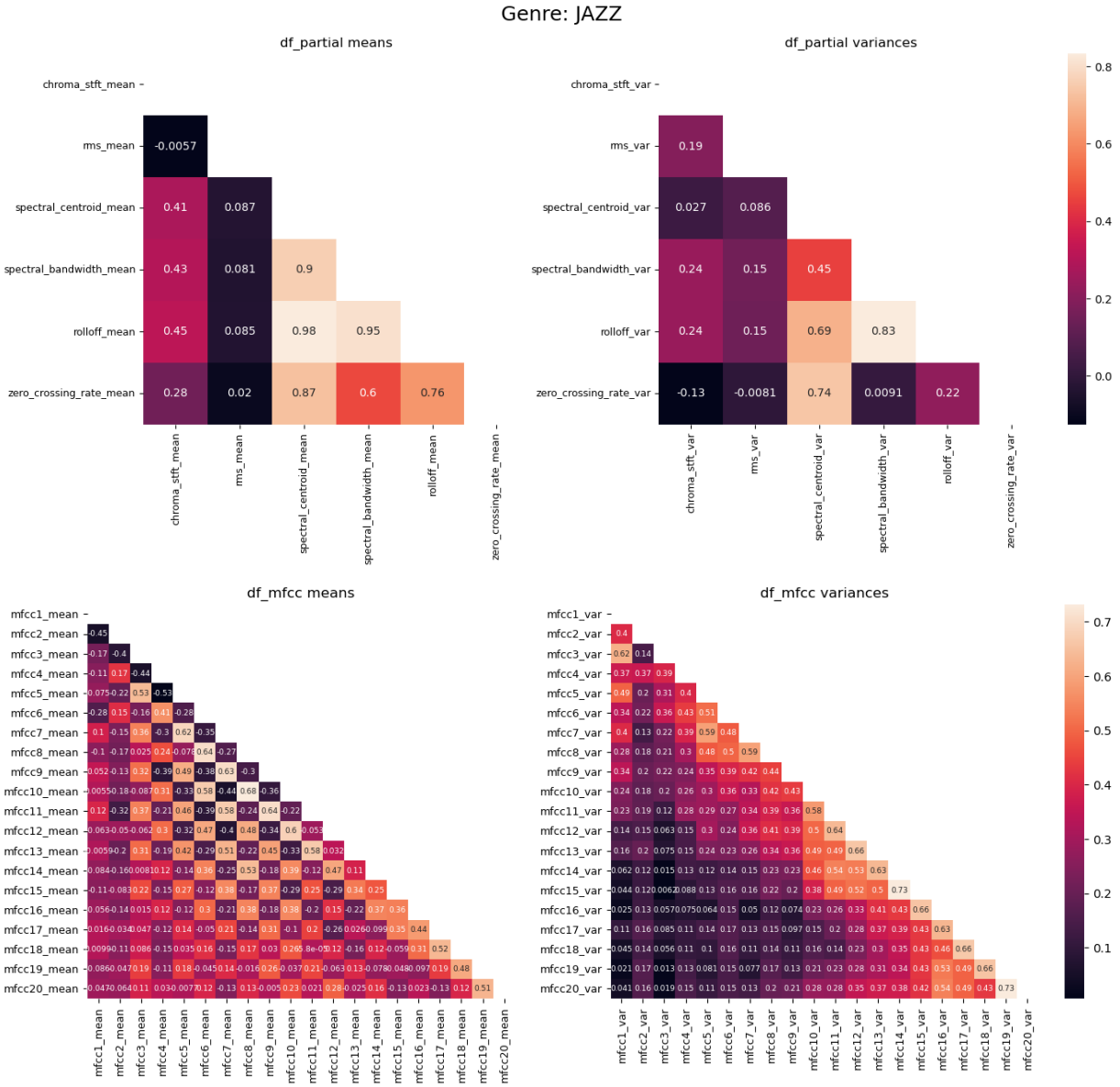
In [16]:
```
def heatmap_label(df, label):

    """Draw the heatmap for the given label."""

    # Get all the data with the given label
    df = df[df['label'] == label]

    # Seperate original dataframe into two, one with MFCCs and one without
    df_mfcc = df.filter(regex=r'(mfcc*|label)')
    df_partial = df.drop(labels=df.filter(regex=r'mfcc*').columns, axis=1)

    # Correlation matrices for df_partial
```

```
    corr_matrix_means_partial = df_partial.filter(regex=r'_mean').corr()
    corr_matrix_vars_partial = df_partial.filter(regex=r'_var').corr()

    # Plot heatmap of df_partial
    fig, axs = plt.subplots(ncols=2, gridspec_kw={'width_ratios': [0.8, 1]})
    fig.suptitle(f'Genre: {label.upper()}', fontsize=18, x=0.45)
    fig = plt.gcf()
    fig.set_size_inches(16, 6)
    plt.subplots_adjust(wspace=0.3)
    mask1 = np.triu(np.ones_like(corr_matrix_means_partial, dtype=bool))
    fig1 = sns.heatmap(corr_matrix_means_partial, ax=axs[0], cbar=False, annot=
    fig1.set_xticklabels(fig1.get_yticklabels(), fontsize=9)
    fig1.set_yticklabels(fig1.get_yticklabels(), fontsize=9)
    fig1.set_title('df_partial means')
    mask2 = np.triu(np.ones_like(corr_matrix_vars_partial, dtype=bool))
    fig2 = sns.heatmap(corr_matrix_vars_partial, ax=axs[1], cbar=True, annot=Tr
    fig2.set_xticklabels(fig2.get_yticklabels(), fontsize=9)
    fig2.set_yticklabels(fig2.get_yticklabels(), fontsize=9) # rotation=-30
    fig2.set_title('df_partial variances')
    plt.show()

    # Correlation matrices for df_mfcc
    corr_matrix_means_mfcc = df_mfcc.filter(regex=r'_mean').corr()
    corr_matrix_vars_mfcc = df_mfcc.filter(regex=r'_var').corr()


    # Plot heatmap of correlation matrix
    fig, axs = plt.subplots(ncols=2, gridspec_kw={'width_ratios': [0.8, 1]})
    fig = plt.gcf()
    fig.set_size_inches(16, 6)
    plt.subplots_adjust(wspace=0.2)
    mask1 = np.triu(np.ones_like(corr_matrix_means_mfcc, dtype=bool))
    fig1 = sns.heatmap(corr_matrix_means_mfcc, ax=axs[0], cbar=False, annot=Tru
    fig1.set_xticklabels(fig1.get_yticklabels(), fontsize=9)
    fig1.set_yticklabels(fig1.get_yticklabels(), fontsize=9)
    fig1.set_title('df_mfcc means')
    mask2 = np.triu(np.ones_like(corr_matrix_vars_mfcc, dtype=bool))
    fig2 = sns.heatmap(corr_matrix_vars_mfcc, ax=axs[1], cbar=True, annot=True,
    fig2.set_xticklabels(fig2.get_yticklabels(), fontsize=9)
    fig2.set_yticklabels(fig2.get_yticklabels(), fontsize=9) # rotation=-30
    fig2.set_title('df_mfcc variances')
    plt.show()
```

In [17]:
```
df_label['label'].unique()
```

Out[17]:
```
array(['blues', 'classical', 'country', 'disco', 'hiphop', 'jazz',
       'metal', 'pop', 'reggae', 'rock'], dtype=object)
```

In [18]:
```
heatmap_label(df, 'jazz')
```

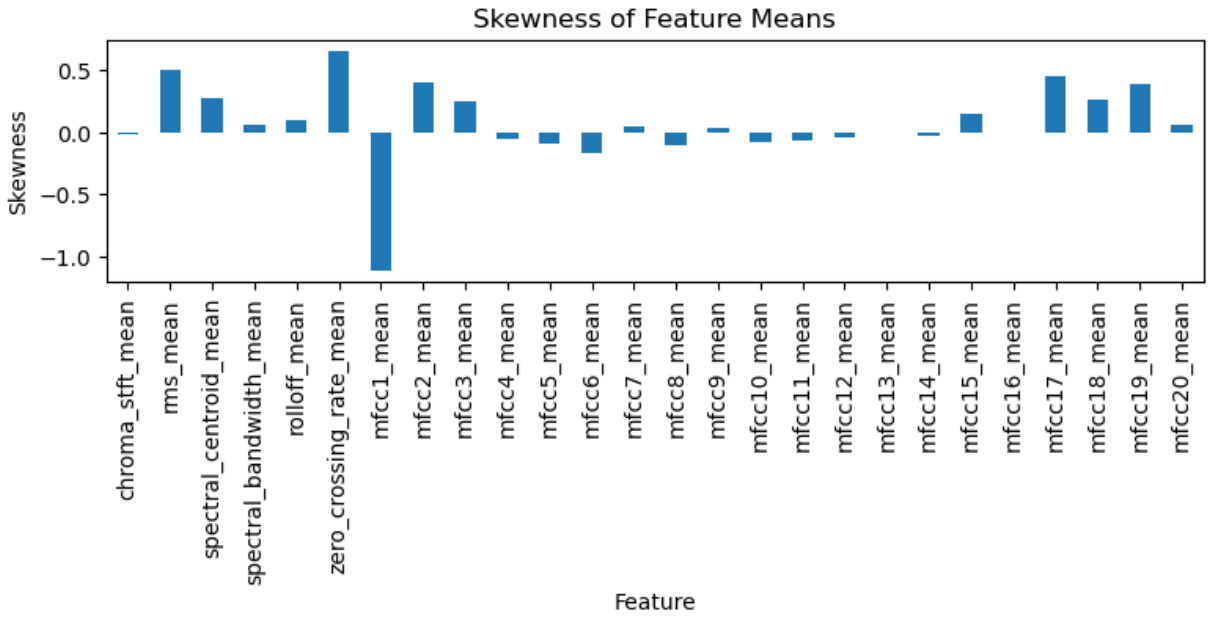


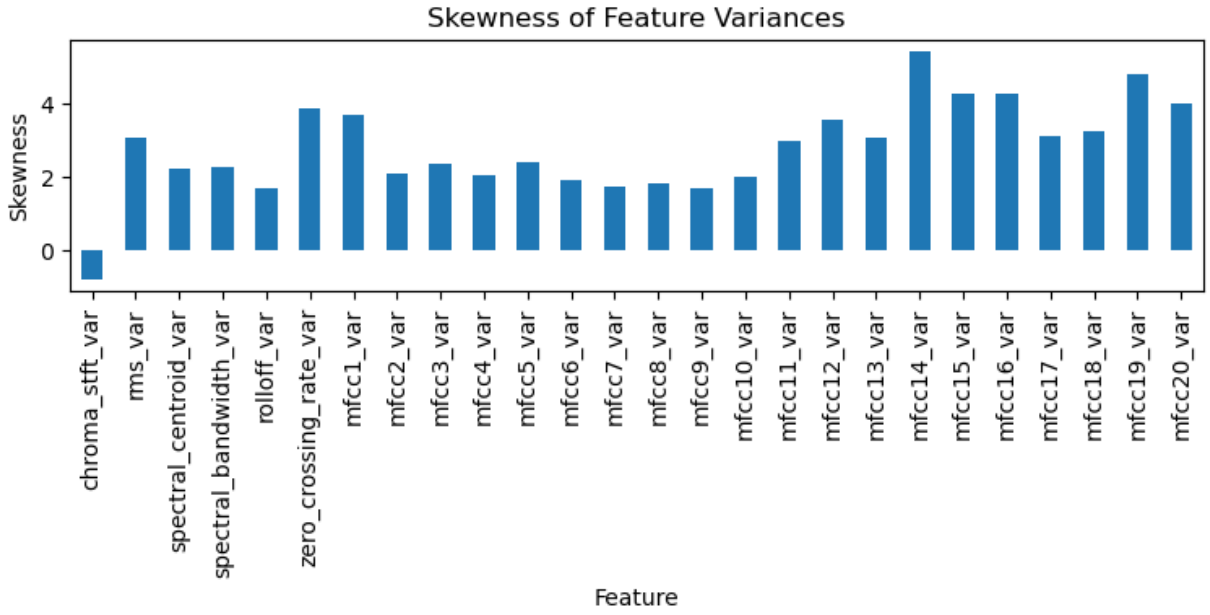
Feature Distribution

- Feature distribution for the entire dataset

In [19]:
```
# Check skewness
skewness_means = df[[col for col in df.columns if 'mean' in col]].skew()
skewness_vars = df[[col for col in df.columns if 'var' in col]].skew()

# Bar plot for skewness
skewness_means.plot(kind='bar', figsize=(9,2))
plt.title('Skewness of Feature Means')
plt.xlabel('Feature')
plt.ylabel('Skewness')
plt.show()

skewness_vars.plot(kind='bar', figsize=(9,2))
plt.title('Skewness of Feature Variances')
plt.xlabel('Feature')
```

```
plt.ylabel('Skewness')
plt.show()
```





- Feature distribution by label

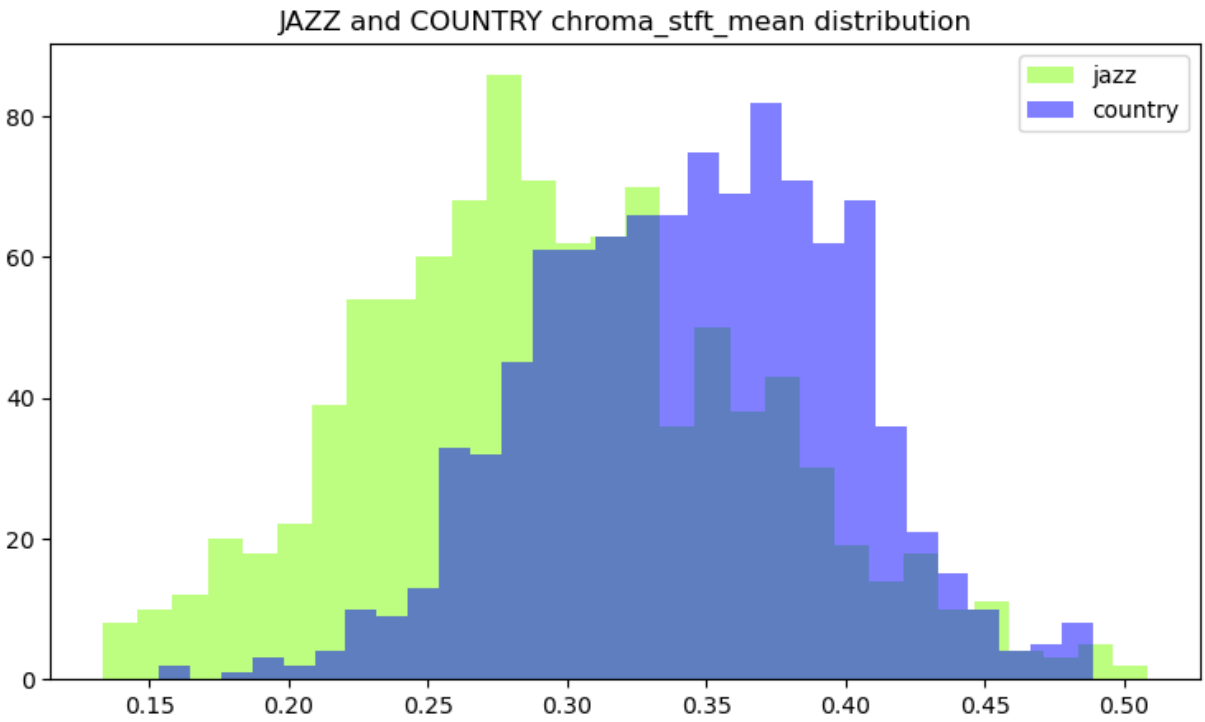
In [20]:
```
def compare_feature(df, feature, label_1, label_2):
    """
    Draw and comapre given labels' selected feature distribution.
    """
    label_1_df = df[df['label']==label_1][feature]
    label_2_df = df[df['label']==label_2][feature]
    bins = 30
    plt.figure(figsize=(9,5))
    plt.hist(label_1_df, alpha=0.5, label=label_1, bins=bins, color='lawngreen
    plt.hist(label_2_df, alpha=0.5, label=label_2, bins=bins, color='blue')
    plt.legend(loc='upper right')
```

```
    plt.title(f"{label_1.upper()} and {label_2.upper()} {feature} distribution'
    plt.show()
```

In [21]:
```
df.columns
```

Out[21]:
```
Index(['chroma_stft_mean', 'chroma_stft_var', 'rms_mean', 'rms_var',
       'spectral_centroid_mean', 'spectral_centroid_var',
       'spectral_bandwidth_mean', 'spectral_bandwidth_var', 'rolloff_mean',
       'rolloff_var', 'zero_crossing_rate_mean', 'zero_crossing_rate_var',
       'tempo', 'mfcc1_mean', 'mfcc1_var', 'mfcc2_mean', 'mfcc2_var',
       'mfcc3_mean', 'mfcc3_var', 'mfcc4_mean', 'mfcc4_var', 'mfcc5_mean',
       'mfcc5_var', 'mfcc6_mean', 'mfcc6_var', 'mfcc7_mean', 'mfcc7_var',
       'mfcc8_mean', 'mfcc8_var', 'mfcc9_mean', 'mfcc9_var', 'mfcc10_mean',
       'mfcc10_var', 'mfcc11_mean', 'mfcc11_var', 'mfcc12_mean', 'mfcc12_var',
       'mfcc13_mean', 'mfcc13_var', 'mfcc14_mean', 'mfcc14_var', 'mfcc15_mea
n',
       'mfcc15_var', 'mfcc16_mean', 'mfcc16_var', 'mfcc17_mean', 'mfcc17_var',
       'mfcc18_mean', 'mfcc18_var', 'mfcc19_mean', 'mfcc19_var', 'mfcc20_mea
n',
       'mfcc20_var', 'label'],
      dtype='object')
```

In [22]:
```
compare_feature(df, 'chroma_stft_mean', 'jazz', 'country')
```



Dimensionality Reduction

Due to the nature of music, which is complex and contains a lot of features, our group decided to run two ways of dimensionality reduction algorithms, in order to have a better understanding of the input data.

In [23]:
```
df.head()
```

Out[23]:

	chroma_stft_mean	chroma_stft_var	rms_mean	rms_var	spectral_centroid_mean	s
0	0.335406	0.091048	0.130405	0.003521	1773.065032	
1	0.343065	0.086147	0.112699	0.001450	1816.693777	
2	0.346815	0.092243	0.132003	0.004620	1788.539719	
3	0.363639	0.086856	0.132565	0.002448	1655.289045	
4	0.335579	0.088129	0.143289	0.001701	1630.656199	

5 rows × 54 columns

In [24]:
```
df_features = df_partial.drop(labels=df_partial.filter(regex=r'_var').columns,
df_features = df_features.drop(labels=df_features.filter(regex=r'mfcc*').colum
df_features.head()
```

Out[24]:

	chroma_stft_mean	rms_mean	spectral_centroid_mean	spectral_bandwidth_mean	ro
0	0.335406	0.130405	1773.065032	1972.744388	37
1	0.343065	0.112699	1816.693777	2010.051501	38
2	0.346815	0.132003	1788.539719	2084.565132	39
3	0.363639	0.132565	1655.289045	1960.039988	35
4	0.335579	0.143289	1630.656199	1948.503884	34

In [25]:
```
mfcc_means = df_mfcc.filter(regex=r'(_mean|label)')
mfcc_means.head()
```

Out[25]:

	mfcc1_mean	mfcc2_mean	mfcc3_mean	mfcc4_mean	mfcc5_mean	mfcc6_mean	m
0	-118.627914	125.083626	-23.443724	41.321484	-5.976108	20.115141	
1	-125.590706	122.421227	-20.718019	50.128387	-11.333302	21.385401	
2	-132.441940	115.085175	-14.811666	50.189293	-0.680819	24.650375	
3	-118.231087	132.116501	-18.758335	39.769306	-13.260426	20.468134	
4	-105.968376	134.643646	-19.961748	40.171753	-14.271939	18.734617	

5 rows × 21 columns

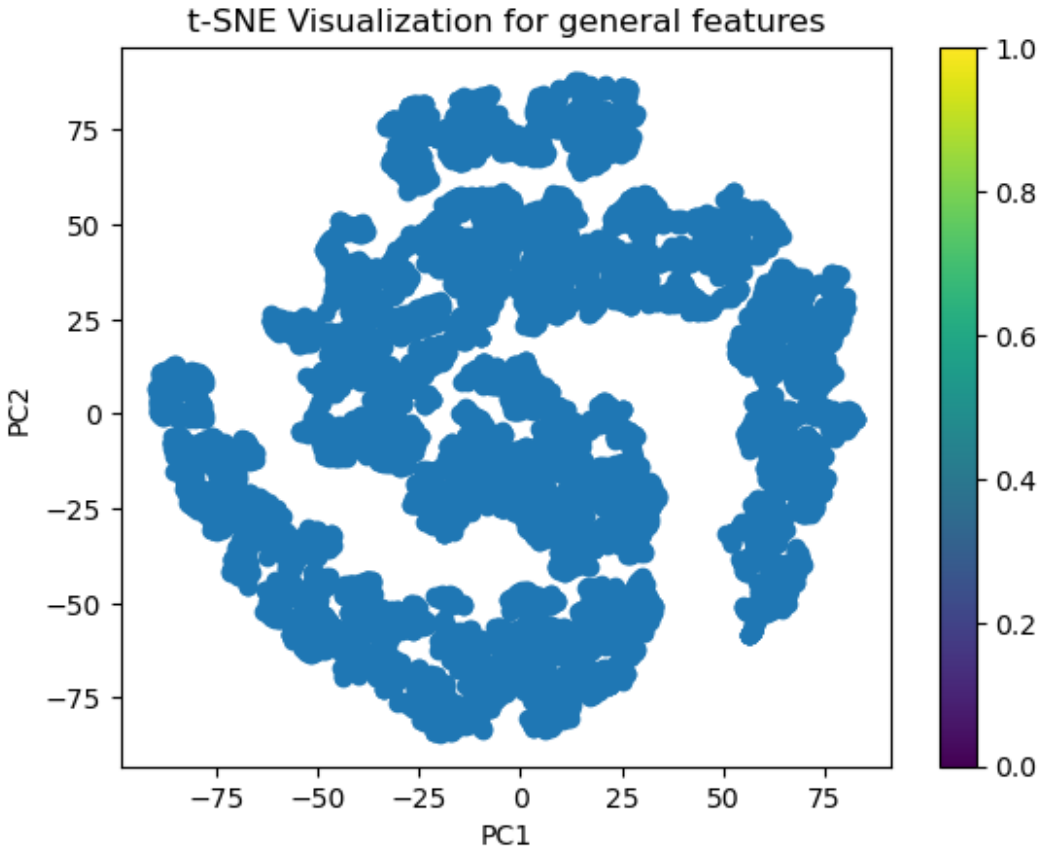
t-Distributed Stochastic Neighbor Embedding (tSNE)

In [26]:
```
tsne = TSNE(n_components=2, random_state=42)

df_features_X = df_features.drop(columns='label')
mfcc_means_X = mfcc_means.drop(columns='label')
```

In [27]:
```
features_tsne = tsne.fit_transform(df_features_X)

plt.scatter(features_tsne[:, 0], features_tsne[:, 1])
```

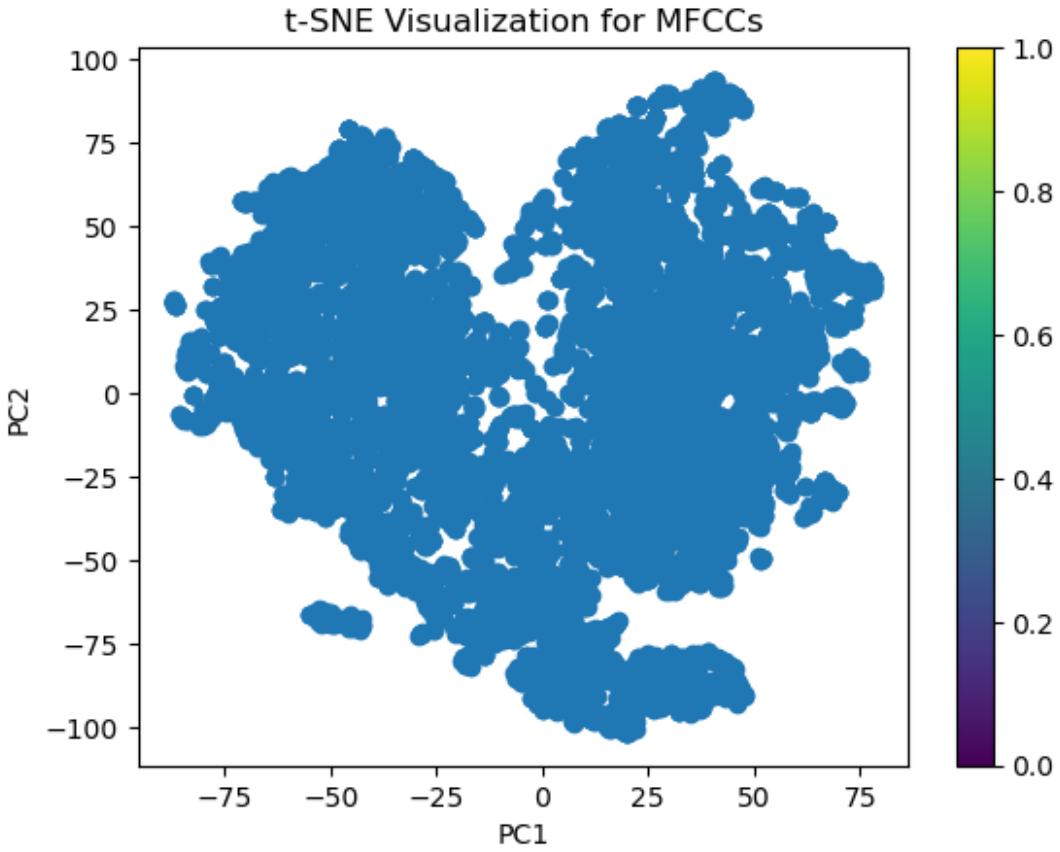
```
plt.colorbar()

plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('t-SNE Visualization for general features')

plt.show()
```

```
d:\anaconda3\lib\site-packages\sklearn\manifold\_t_sne.py:780: FutureWarning: T
he default initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
d:\anaconda3\lib\site-packages\sklearn\manifold\_t_sne.py:790: FutureWarning: T
he default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
```



In [28]:
```
mfcc_tsne = tsne.fit_transform(mfcc_means_X)

plt.scatter(mfcc_tsne[:, 0], mfcc_tsne[:, 1])
plt.colorbar()

plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title('t-SNE Visualization for MFCCs')

plt.show()
```

```
d:\anaconda3\lib\site-packages\sklearn\manifold\_t_sne.py:780: FutureWarning: T
he default initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
d:\anaconda3\lib\site-packages\sklearn\manifold\_t_sne.py:790: FutureWarning: T
he default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
```



Principal Component Analysis (PCA)

In [29]:
```
pca = PCA(n_components=2)
pca.fit(df_features_X)

X_pca = pca.transform(df_features_X)

explained_var = pca.explained_variance_ratio_
components = pca.components_

for i, (pc1, pc2) in enumerate(zip(components[0], components[1])):
    print(f"PC{i+1}:")
    for selected_feature, component in zip(df_features_X.columns, [pc1, pc2]):
        print(f"{selected_feature}: {component}")
    print()
```

```
PC1:
chroma_stft_mean: 2.5472343776365152e-05
rms_mean: 5.1443773184812960e-05

PC2:
chroma_stft_mean: 1.826849128205854e-05
rms_mean: -2.7220616716002594e-05

PC3:
chroma_stft_mean: 0.3932166279294755
rms_mean: 0.6709516844938745

PC4:
chroma_stft_mean: 0.27674153531669893
rms_mean: -0.7383820456181397

PC5:
chroma_stft_mean: 0.8768094376059006
rms_mean: -0.06784592279506878

PC6:
chroma_stft_mean: 1.863603110869982e-05
rms_mean: 0.00012822365416466244

PC7:
chroma_stft_mean: 0.00012225250124726081
rms_mean: -0.003564086834236504
```

In [30]:
```
print("Transformed Data:")
print(X_pca)
print("\nExplained Variance Ratio:")
print(explained_var)
print("\nPrincipal Components:")
print(components)
```

```
Transformed Data:
[[ -988.53222047   -29.81236614]
 [ -825.04066662   -38.5889469 ]
 [ -703.29645971  -121.17984422]
 ...
 [-2385.58447282    73.47913212]
 [ -328.4891177    104.87154982]
 [-1293.62950057   -28.37332455]]

Explained Variance Ratio:
[0.98465023 0.01207904]

Principal Components:
[[ 2.54723438e-05  1.82684913e-05  3.93216628e-01  2.76741535e-01
   8.76809438e-01  1.86360311e-05  1.22252501e-04]
 [ 5.14437732e-05 -2.72206167e-05  6.70951684e-01 -7.38382046e-01
  -6.78459228e-02  1.28223654e-04 -3.56408683e-03]]
```

In [31]:
```
pca = PCA(n_components=2)
pca.fit(mfcc_means_X)

mfccX_pca = pca.transform(mfcc_means_X)
```

```
explained_var_mfcc = pca.explained_variance_ratio_
mfcc_components = pca.components_

for i, (pc1, pc2) in enumerate(zip(mfcc_components[0], mfcc_components[1])):
    print(f"PC{i+1}:")
    for selected_feature, component in zip(mfcc_means_X.columns, [pc1, pc2]):
        print(f"{selected_feature}: {component}")
    print()
```

```
PC1:
mfcc1_mean: -0.9767964554570698
mfcc2_mean: -0.09272412397613422

PC2:
mfcc1_mean: 0.20751395325251398
mfcc2_mean: -0.5582042935341508

PC3:
mfcc1_mean: 0.015675146541436858
mfcc2_mean: 0.5794976363926588

PC4:
mfcc1_mean: -0.006988342459386438
mfcc2_mean: -0.285804980514221

PC5:
mfcc1_mean: 0.009012440681307252
mfcc2_mean: 0.27315878390878023

PC6:
mfcc1_mean: -0.01855881651748591
mfcc2_mean: -0.15918144717090968

PC7:
mfcc1_mean: -0.004006812258054187
mfcc2_mean: 0.22080259041937808

PC8:
mfcc1_mean: -0.027061585414887897
mfcc2_mean: -0.13796458777833373

PC9:
mfcc1_mean: -0.006219913266092596
mfcc2_mean: 0.17537182652135175

PC10:
mfcc1_mean: -0.022614347671483848
mfcc2_mean: -0.10405941306262753

PC11:
mfcc1_mean: -0.0005339575499214545
mfcc2_mean: 0.12843840558069144

PC12:
mfcc1_mean: -0.017653110231813218
mfcc2_mean: -0.08766208936550453

PC13:
mfcc1_mean: 0.007897852751977557
mfcc2_mean: 0.11496074655129303

PC14:
mfcc1_mean: -0.010216451282698285
mfcc2_mean: -0.047469331149659046

PC15:
mfcc1_mean: 0.007961288026712074
```

```
mfcc2_mean: 0.07764147933445208

PC16:
mfcc1_mean: -0.007385578164090786
mfcc2_mean: -0.041967505848294506

PC17:
mfcc1_mean: 0.00831212117529264
mfcc2_mean: 0.05952902007194553

PC18:
mfcc1_mean: -0.00732268106568712
mfcc2_mean: -0.02005781117439903

PC19:
mfcc1_mean: 0.004843169435758939
mfcc2_mean: 0.039845592996054446

PC20:
mfcc1_mean: -0.0067227500960797965
mfcc2_mean: -0.002926182075475912
```

In [32]:
```
print("Transformed Data:")
print(mfccX_pca)
print("\nExplained Variance Ratio:")
print(explained_var_mfcc)
print("\nPrincipal Components:")
print(mfcc_components)
```

```
Transformed Data:
[[-21.84154369 -32.62291055]
 [-16.00848984 -34.59836586]
 [-10.38134324 -21.0838663 ]
 ...
 [145.21476878  -8.73660249]
 [ 16.19593476 -38.01334946]
 [ 92.67213926 -23.77936789]]

Explained Variance Ratio:
[0.81301072 0.07957108]

Principal Components:
[[-9.76796455e-01  2.07513953e-01  1.56751465e-02 -6.98834246e-03
   9.01244068e-03 -1.85588165e-02 -4.00681226e-03 -2.70615854e-02
  -6.21991327e-03 -2.26143477e-02 -5.33957550e-04 -1.76531102e-02
   7.89785275e-03 -1.02164513e-02  7.96128803e-03 -7.38557816e-03
   8.31212118e-03 -7.32268107e-03  4.84316944e-03 -6.72275010e-03]
 [-9.27241240e-02 -5.58204294e-01  5.79497636e-01 -2.85804981e-01
   2.73158784e-01 -1.59181447e-01  2.20802590e-01 -1.37964588e-01
   1.75371827e-01 -1.04059413e-01  1.28438406e-01 -8.76620894e-02
   1.14960747e-01 -4.74693311e-02  7.76414793e-02 -4.19675058e-02
   5.95290201e-02 -2.00578112e-02  3.98455930e-02 -2.92618208e-03]]
```

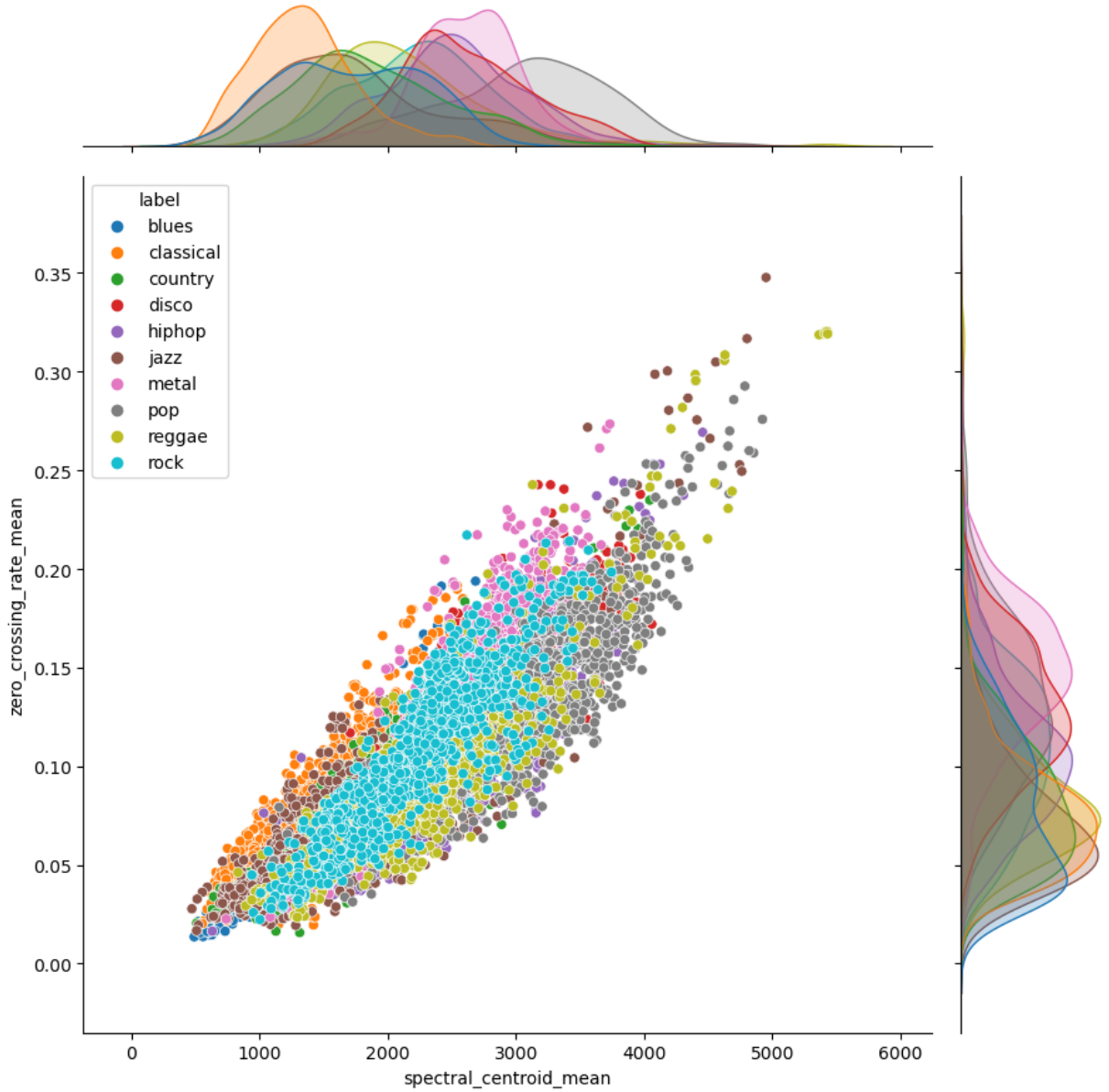
Joint Plot Visualization

With our previous visualization and analysis, we can infer some information:

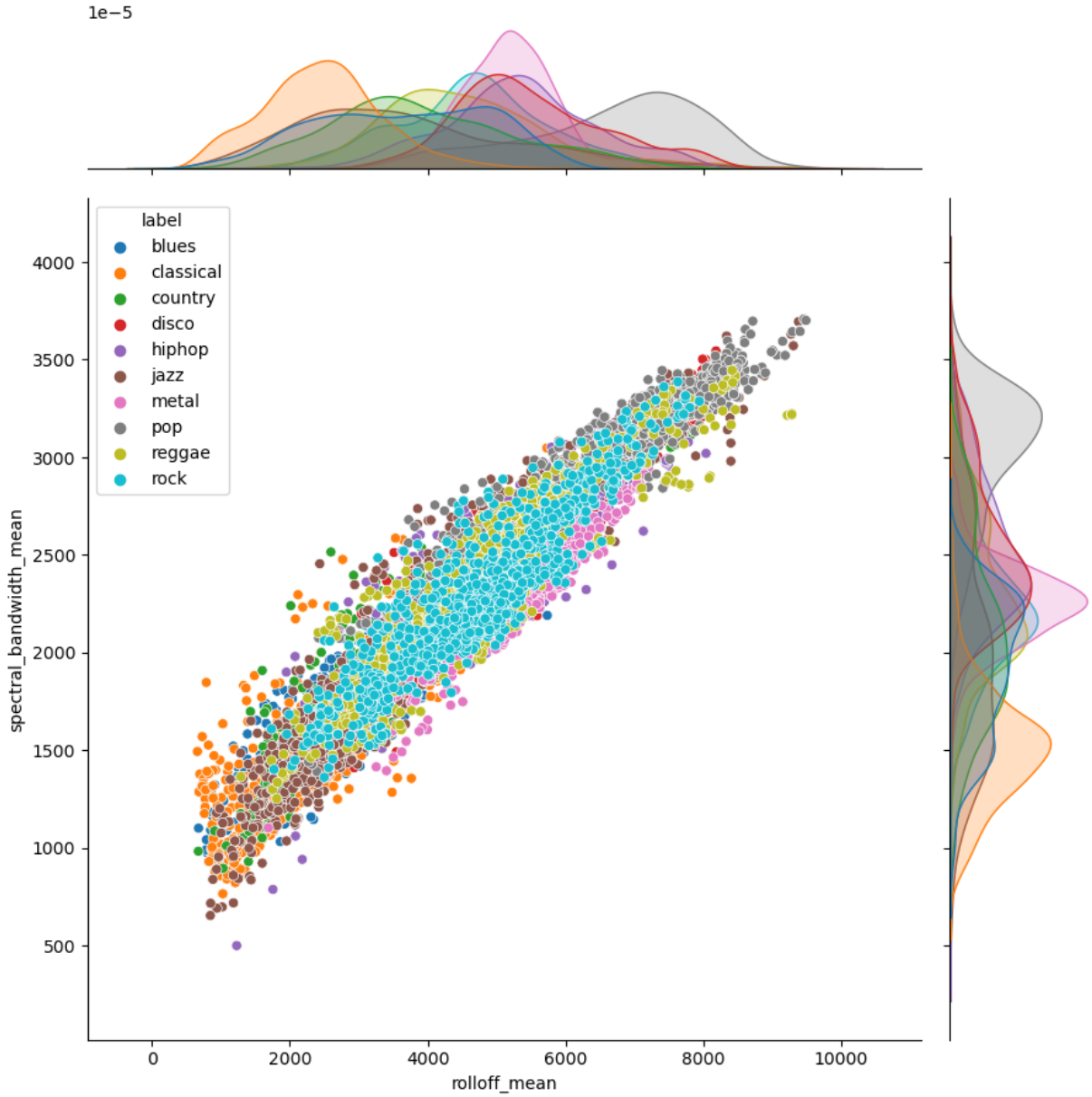
Based on the data we have, since spectral centroid and spectral bandwidth are related, it's not suprised to see their high correlation. But we also found the there's relatively high correlation in between `rolloff` & `spectral centroid`, `rolloff` & `spectral bandwidth`, `rolloff` & `zero crossing rate`, which are worth discovering more.

Moving forward, we generated several joint plots for visualization with labeled genres:

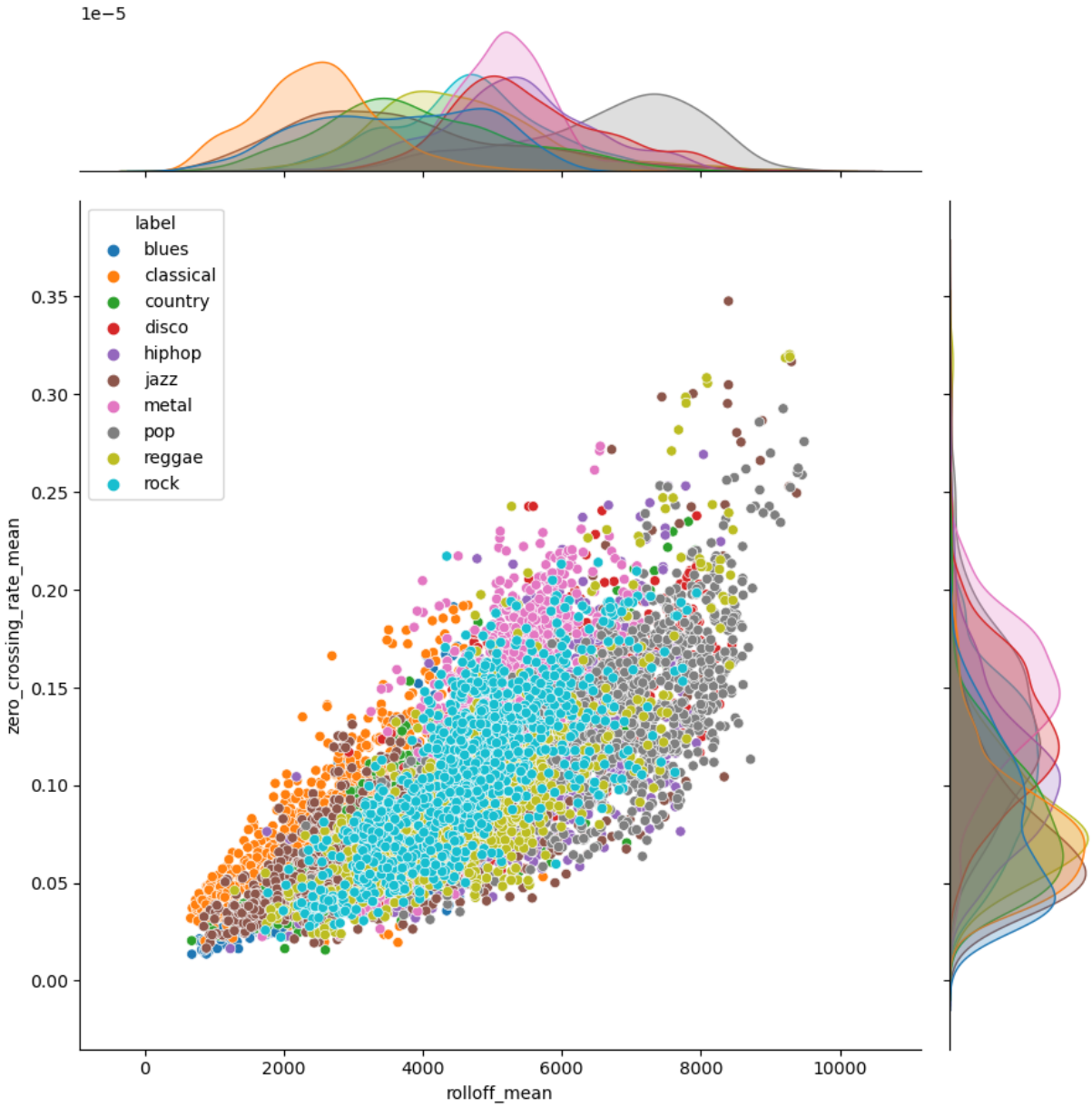
In [33]:
```
sns.jointplot(data=df_partial, x=df['spectral_centroid_mean'], y=df['zero_cross
              hue='label', kind='scatter', height=9)
plt.show()
```



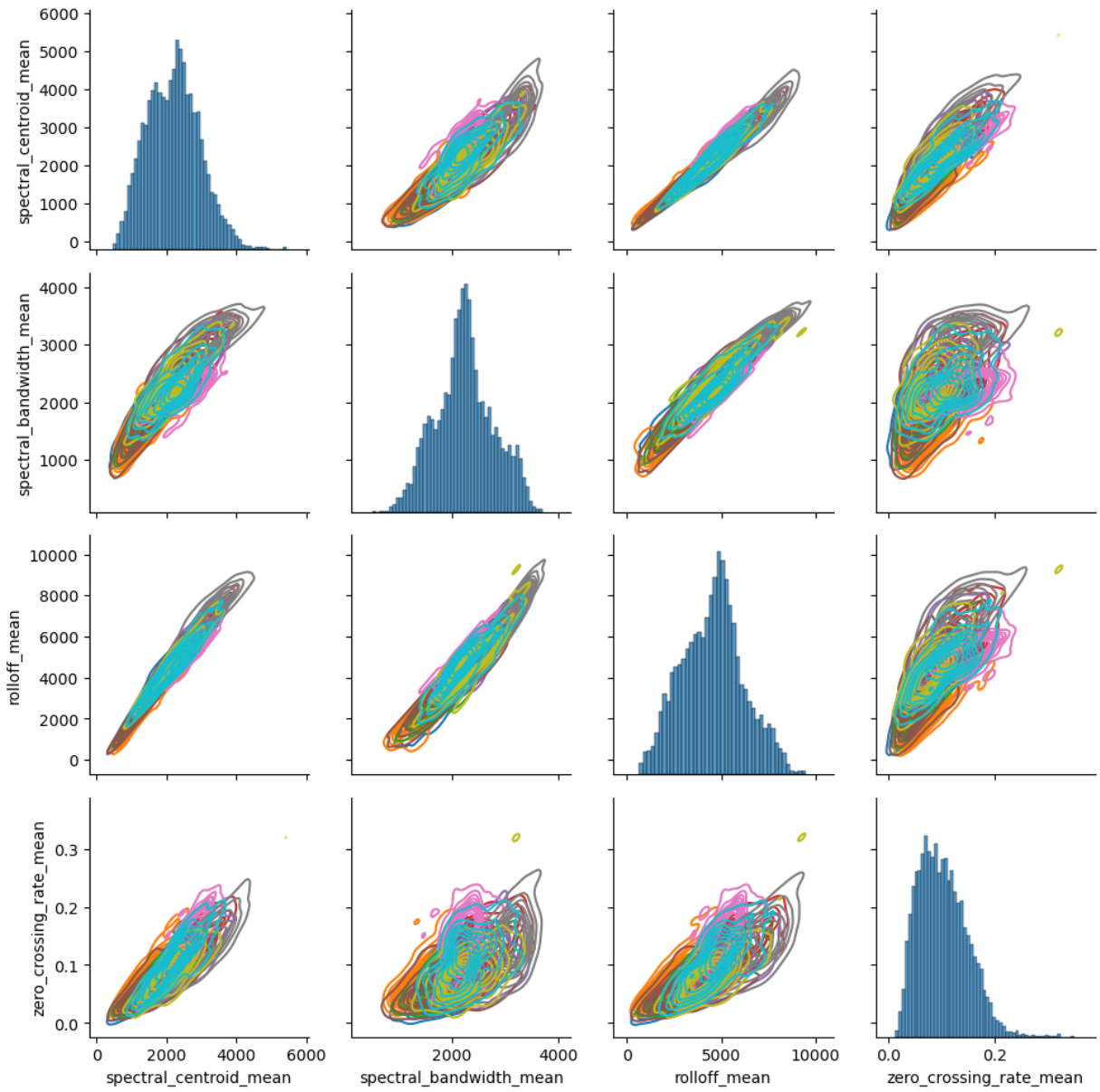
In [34]:
```
sns.jointplot(data=df_partial, x=df['rolloff_mean'], y=df['spectral_bandwidth_m
              hue='label', kind='scatter', height=9)
plt.show()
```



In [35]:
```
sns.jointplot(data=df_partial, x=df['rolloff_mean'], y=df['zero_crossing_rate_m
              hue='label', kind='scatter', height=9)
plt.show()
```



In [36]:
```
variables = ["spectral_centroid_mean", "spectral_bandwidth_mean", "rolloff_mean
graph = sns.PairGrid(df_raw, hue='label', vars=variables)
graph.map_diag(sns.histplot, hue=None)
graph.map_offdiag(sns.kdeplot)
plt.show()
```



Proposed Solution

In general, the solution will be the following in steps:

1. Data Processing: We will begin by pre-processing our dataset. This involves handling missing data, dealing with outliers, and possibly transforming variables as necessary to make them appropraite to use.
2. Feature Selection: We will choose chroma_stft, rms, spectral_centroid, spectral_bandwidth, roll_off, zero_crossing_rate, harmony, perceptr, tempo, mfcc(1-20) to determine the genre of songs. Since we can't get those data in every frame or seconds, so we just get their mean and variance for future usage.
3. Model Training: We will utilize KNN, SVM, and Neural Networks (if possible) to train our models, adjusting parameters to find the most optimal configuration for each model.

4. Model Evaluation and Selection: We will evaluate each model's performance using various evaluation metrics such as accuracy, precision, recall, F1 score, and confusion matrix. The model that performs the best across all metrics will be selected as our final model.
5. Testing on Unseen Data: Finally, we will test our selected model on unseen data to assess its ability to generalize and make accurate predictions on new data to ensure that it doesn't overfit.

Then, to be specific, we will talk about them in details:

1. For algorithm of multi-class classification, we will use K-Nearest Neighbors (KNN), Support Vector Machines (SVM), and Neural Networks (If possible). We believe these methods are particularly suitable for our task due to their ability to handle high-dimensional data and flexibility in model complexity.
2. The machine learning algorithms will be trained on a comprehensive dataset comprising multiple music genre samples, where each song is represented by an array of extracted audio features. The aim is to create a model that, given these features of a song, can predict its genre with high accuracy.
3. For the library we may wanna use, we plan to implement our solution using Python and relevant libraries such as Pandas for data preprocessing, Scikit-learn for machine learning algorithms, and Matplotlib for data visualization.
4. For benchmark comparison, we will use a simple classification model, like Logistic Regression. The performance of our machine learning models will be used to be against this benchmark model to assess their efficiency and accuracy.

Algorithms implementation

In [37]:
```
# Suppress sklearn future warnings
import warnings
warnings.filterwarnings("ignore")
```

Split data into train and test sets

In [38]:
```
data = df.drop('label', axis=1)
label_in_str = df['label']

# Encode target strings with values from 0 to n_classes - 1
le = preprocessing.LabelEncoder()
label = le.fit_transform(label_in_str)

# Standardize feature values
scalar = preprocessing.StandardScaler()
scaled_data = scalar.fit_transform(data)

# Train test split
TEST_SIZE = 0.2
RANDOM_STATE = 42
```

```
X_train, X_test, y_train, y_test = train_test_split(scaled_data, label, test_s
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[38]: ((7992, 53), (1998, 53), (7992,), (1998,))

Kth Nearest Neighbors

Grid Search to get the best param for knn

In [39]:
```
# Grid Search
parameters = {"n_neighbors": range(1, 50), 'weights': ['uniform', 'distance']}
gridsearch = GridSearchCV(KNeighborsRegressor(), parameters, cv = 5)

gridsearch.fit(X_train, y_train)

best_param = gridsearch.best_params_
print(best_param)

best_k = best_param["n_neighbors"]
best_weights = best_param["weights"]

preds_grid = gridsearch.predict(X_test)
rmse_grid = sqrt(mean_squared_error(y_test, preds_grid))
print(rmse_grid)

bagged_knn = KNeighborsRegressor(n_neighbors=best_k, weights=best_weights)
bagging_model = BaggingRegressor(bagged_knn, n_estimators=50)
bagging_model.fit(X_train, y_train)
preds_bag = bagging_model.predict(X_test)
rmse_bag = sqrt(mean_squared_error(y_test, preds_bag))
print(rmse_bag)

knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print('Train Score', knn.score(X_train, y_train))
print('Test Score', knn.score(X_test, y_test))
accuracy = (y_pred==y_test).sum() / len(y_test)
print(f"Accuracy for this model is {accuracy*100:.1f}%")
```

```
{'n_neighbors': 3, 'weights': 'distance'}
1.3662821821138516
1.3354039991302267
Train Score 0.9077827827827828
Test Score 0.8573573573573574
Accuracy for this model is 85.7%
```

In [40]:
```
# Set up StratifiedKFold
strat_k_fold = StratifiedKFold(n_splits=5, shuffle=True, random_state=RANDOM_S

# Grid search
parameters = {"n_neighbors": range(1, 50),
              'weights': ['uniform', 'distance'],
              'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
              'p': [1, 2]}
```

```
gridsearch = GridSearchCV(KNeighborsClassifier(), parameters, cv=strat_k_fold,

gridsearch.fit(X_train, y_train)

best_param = gridsearch.best_params_
print("Best parameters: ", best_param)

best_k = best_param["n_neighbors"]
best_weights = best_param["weights"]
best_algorithm = best_param["algorithm"]
best_p = best_param["p"]

# KNN model with best parameters
knn_best = KNeighborsClassifier(n_neighbors=best_k, weights=best_weights, algo

# Use cross_val_score for model evaluation
cv_scores_train = cross_val_score(knn_best, X_train, y_train, cv=strat_k_fold,
cv_scores_test = cross_val_score(knn_best, X_test, y_test, cv=strat_k_fold, sc

print('Cross-validation Training Score: ', cv_scores_train.mean())
print('Cross-validation Testing Score: ', cv_scores_test.mean())

# BaggingClassifier with KNeighborsClassifier as base estimator
bagged_knn = KNeighborsClassifier(n_neighbors=best_k, weights=best_weights, alg
bagging_model = BaggingClassifier(bagged_knn, n_estimators=50, random_state=RAN

bagging_model.fit(X_train, y_train)

y_pred_bag = bagging_model.predict(X_test)

accuracy_bag = accuracy_score(y_test, y_pred_bag)
print("Bagging Model Accuracy: ", accuracy_bag)
```

```
Best parameters:  {'algorithm': 'auto', 'n_neighbors': 1, 'p': 1, 'weights': 'u
niform'}
Cross-validation Training Score:  0.9015260633014532
Cross-validation Testing Score:  0.7382518796992481
Bagging Model Accuracy:  0.918918918918919
```

In [41]:

```
# Use SMOTE for class balancing
smote = SMOTE(random_state=RANDOM_STATE)
X_smote, y_smote = smote.fit_resample(X_train, y_train)

# Standardization
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_smote)
X_test_scaled = scaler.transform(X_test)

# Set up StratifiedKFold
strat_k_fold = StratifiedKFold(n_splits=5, shuffle=True, random_state=RANDOM_S

# Grid search
parameters = {"n_neighbors": range(1, 50),
              'weights': ['uniform', 'distance'],
              'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
              'p': [1, 2]}

gridsearch = GridSearchCV(KNeighborsClassifier(), parameters, cv=strat_k_fold,
```

```
gridsearch.fit(X_train_scaled, y_smote)

best_param = gridsearch.best_params_
print("Best parameters: ", best_param)

best_k = best_param["n_neighbors"]
best_weights = best_param["weights"]
best_algorithm = best_param["algorithm"]
best_p = best_param["p"]

# KNN model with best parameters
knn_best = KNeighborsClassifier(n_neighbors=best_k, weights=best_weights, algo

# Use cross_val_score for model evaluation
cv_scores_train = cross_val_score(knn_best, X_train_scaled, y_smote, cv=strat_
cv_scores_test = cross_val_score(knn_best, X_test_scaled, y_test, cv=strat_k_f

print('Cross-validation Training Score: ', cv_scores_train.mean())
print('Cross-validation Testing Score: ', cv_scores_test.mean())

# BaggingClassifier with KNeighborsClassifier as base estimator
bagged_knn = KNeighborsClassifier(n_neighbors=best_k, weights=best_weights, al
bagging_model = BaggingClassifier(bagged_knn, n_estimators=50, random_state=RA

bagging_model.fit(X_train_scaled, y_smote)

y_pred_bag = bagging_model.predict(X_test_scaled)

accuracy_bag = accuracy_score(y_test, y_pred_bag)
print("Bagging Model Accuracy: ", accuracy_bag)
```

```
Best parameters:  {'algorithm': 'ball_tree', 'n_neighbors': 1, 'p': 1, 'weight
s': 'uniform'}
Cross-validation Training Score:  0.9053658536585365
Cross-validation Testing Score:  0.739250626566416
Bagging Model Accuracy:  0.918918918918919
```

SVM

In [42]:

```
svm = SVC(random_state=RANDOM_STATE)

svm_grid = {
    'C': [0.1, 1, 10, 100, 1000],  # Regularization parameter
    'gamma': [1, 0.1, 0.01, 0.001, 0.0001],  # Kernel coefficient
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid']  # Kernel types
}

svm_search = GridSearchCV(
    svm,
    param_grid=svm_grid,
    cv=StratifiedKFold(n_splits=5),  # maintain distribution of classes across
    verbose=1,
    n_jobs=-1  # use all processors
)

svm_search.fit(X_train, y_train)
```

```
print(f'Best parameters for SVM: {svm_search.best_params_}')

y_pred_svm = svm_search.predict(X_test)
accuracy_svm = accuracy_score(y_test, y_pred_svm)

print(f'Accuracy for SVM: {accuracy_svm*100:.1f}%')
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
Best parameters for SVM: {'C': 100, 'gamma': 0.01, 'kernel': 'rbf'}
Accuracy for SVM: 88.9%
```

In [62]:
```
RANDOM_STATE = 42

svm_grid = {
    'C': [0.01, 0.1, 1, 10],  # Regularization parameter
    'gamma': [1, 0.5, 0.1, 0.01],  # Kernel coefficient
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],  # Kernel types
    'degree': [2, 3, 4]  # Degree for 'poly' kernel
}

svm = SVC(random_state=RANDOM_STATE)

svm_search = GridSearchCV(
    svm,
    param_grid=svm_grid,
    cv=StratifiedKFold(n_splits=5),  # maintain distribution of classes across
    verbose=1,
    n_jobs=-1  # use all processors
)

svm_search.fit(X_train, y_train)

print(f'Best parameters for SVM: {svm_search.best_params_}')

y_pred_svm = svm_search.predict(X_test)
accuracy_svm = accuracy_score(y_test, y_pred_svm)
print(f'Accuracy for SVM: {accuracy_svm*100:.1f}%')

# BaggingClassifier with SVC as base estimator
best_svm = SVC(C=svm_search.best_params_['C'], gamma=svm_search.best_params_['
bagging_svm = BaggingClassifier(best_svm, n_estimators=10, random_state=RANDOM_

bagging_svm.fit(X_train, y_train)

y_pred_bagging_svm = bagging_svm.predict(X_test)
accuracy_bagging_svm = accuracy_score(y_test, y_pred_bagging_svm)
print(f'Bagging SVM Accuracy: {accuracy_bagging_svm*100:.1f}%')

# Handling class imbalance with SMOTE
# smote = SMOTE()
# X_train_balanced, y_train_balanced = smote.fit_resample(X_train_scaled, y_tra

# Train the final model
# best_svm.fit(X_train_scaled, y_train)
# y_pred_final = best_svm.predict(X_test_scaled)
# accuracy_final = accuracy_score(y_test, y_pred_final)
# print(f'Final SVM Accuracy: {accuracy_final*100:.1f}%')
```

```
Fitting 5 folds for each of 192 candidates, totalling 960 fits
Best parameters for SVM: {'C': 0.1, 'degree': 3, 'gamma': 0.1, 'kernel': 'pol
y'}
Accuracy for SVM: 88.3%
Bagging SVM Accuracy: 87.4%
```

Neural Network Models

In [45]:
```
# setting up the data
data = df.drop(['label', 'tempo'], axis=1)
label_in_str = df['label']

# Encode target strings with values from 0 to n_classes - 1
le = preprocessing.LabelEncoder()
label = le.fit_transform(label_in_str)

# Standardize feature values
scalar = preprocessing.StandardScaler()
scaled_data = scalar.fit_transform(data)

# Train test split
TEST_SIZE = 0.2
RANDOM_STATE = 42
X_train, X_test, y_train, y_test = train_test_split(scaled_data, label, test_s
```

Recurrent Neural Network (RNN)

In [46]:
```
X_train_rnn = np.reshape(X_train, (7992, 26, 2))
X_test_rnn = np.reshape(X_test, (1998, 26, 2))
X_train_rnn.shape, X_test.shape, X_test_rnn.shape, y_test.shape
```

Out[46]:
```
((7992, 26, 2), (1998, 52), (1998, 26, 2), (1998,))
```

In [47]:
```
# Turn numpy array into pytorch tensor
X_train_tensor = torch.tensor(X_train_rnn, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test_rnn, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

# Create train and test datasets
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

# Put dataset into dataloader
BATCH_SIZE = 128
train_loader_rnn = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=Tru
test_loader_rnn = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=True)
```

In [48]:
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size, device
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
```

```
        self.output_size = output_size
        self.num_layers = num_layers
        self.device = device

        self.rnn1 = nn.RNN(input_size=input_size,
                           hidden_size=hidden_size,
                           num_layers=num_layers,
                           batch_first=True)
        self.fc = nn.Linear(in_features=hidden_size*26,
                            out_features=output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(self

        out, h0 = self.rnn1(x, h0)
        out = out.reshape(out.shape[0], -1)
        out = self.fc(out)
        return out
```

In [49]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
NUM_EPOCH = 300
LEARNING_RATE = 0.00022
```

In [50]:
```
rnn_model = RNN(2, 128, 10, 10, device).to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(rnn_model.parameters(), lr=LEARNING_RATE)
```

In [51]:
```
torch.manual_seed(42)
torch.cuda.manual_seed(42)

for epoch in tqdm(range(1, NUM_EPOCH+1)):
    rnn_model.train()
    train_loss = 0
    train_acc = 0

    for X, y in train_loader_rnn:
        X = X.to(device)
        y = y.to(device)

        y_pred = rnn_model(X)
        loss = loss_fn(y_pred, y)
        train_loss += loss

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        y_pred = torch.softmax(y_pred, dim=1).argmax(dim=1)
        train_acc += (y==y_pred).sum() / len(y)

    train_loss = train_loss / len(train_loader_rnn)
    train_acc = train_acc * 100 / len(train_loader_rnn)

    rnn_model.eval()
    with torch.inference_mode():
        test_loss = 0
        test_acc = 0
```

```
        for X, y in test_loader_rnn:
            X = X.to(device)
            y = y.to(device)

            y_pred = rnn_model(X)
            loss = loss_fn(y_pred, y)
            test_loss += loss
            y_pred = torch.softmax(y_pred, dim=1).argmax(dim=1)
            test_acc += (y==y_pred).sum() / len(y)

        test_loss = test_loss / len(test_loader_rnn)
        test_acc = test_acc * 100 / len(test_loader_rnn)
    if epoch % 25 == 0:
        print(f'Epoch:{epoch} | Train loss:{train_loss:.2f} | Train acc:{train_
```

```
  0%|          | 0/300 [00:00<?, ?it/s]
Epoch:25 | Train loss:0.59 | Train acc:79.91% | Test loss:0.68 | Test acc:75.6
7%
Epoch:50 | Train loss:0.24 | Train acc:93.01% | Test loss:0.48 | Test acc:84.0
8%
Epoch:75 | Train loss:0.05 | Train acc:99.44% | Test loss:0.47 | Test acc:86.0
3%
Epoch:100 | Train loss:0.01 | Train acc:99.90% | Test loss:0.53 | Test acc:86.1
6%
Epoch:125 | Train loss:0.00 | Train acc:99.90% | Test loss:0.58 | Test acc:85.9
8%
Epoch:150 | Train loss:0.00 | Train acc:99.89% | Test loss:0.59 | Test acc:87.0
1%
Epoch:175 | Train loss:0.00 | Train acc:99.93% | Test loss:0.60 | Test acc:86.8
5%
Epoch:200 | Train loss:0.00 | Train acc:99.85% | Test loss:0.71 | Test acc:85.1
5%
Epoch:225 | Train loss:0.00 | Train acc:99.83% | Test loss:0.80 | Test acc:83.5
5%
Epoch:250 | Train loss:0.00 | Train acc:99.90% | Test loss:0.59 | Test acc:87.2
0%
Epoch:275 | Train loss:0.09 | Train acc:97.22% | Test loss:0.85 | Test acc:82.4
4%
Epoch:300 | Train loss:0.05 | Train acc:98.33% | Test loss:0.77 | Test acc:84.0
7%
```

Convolutional Neural Network (CNN)

In [52]:
```
# Turn numpy array into pytorch tensor
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

# Create train and test datasets
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

# Put dataset into dataloader
BATCH_SIZE = 100
```

```
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=True)
```

In [53]:
```
# Create a CNN model
class CNN(nn.Module):
    def __init__(self, output_size, device):
        super(CNN, self).__init__()
        self.output_size = output_size
        self.device = device

        self.layer_1 = nn.Sequential(
            nn.Conv1d(1, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2)
        )

        self.layer_2 = nn.Sequential(
            nn.Conv1d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2)
        )

        self.layer_3 = nn.Sequential(
            nn.Conv1d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2)
        )

        self.flatten = nn.Flatten()

        self.fc_1 = nn.Sequential(
            nn.Dropout(0.2),
            nn.Linear(128*6, 512),
            nn.ReLU(),
            nn.Dropout(0.2),
        )

        self.fc_2 = nn.Sequential(
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.2),
        )

        self.fc_3 = nn.Sequential(
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.2),
        )

        self.fc_4 = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.2),
        )

        self.fc_5 = nn.Sequential(
            nn.Linear(64, 32),
```

```
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(32, output_size),
        )

    def forward(self, x):
        x = x.unsqueeze(1)
        x = self.layer_1(x)
        x = self.layer_2(x)
        x = self.layer_3(x)
        x = self.flatten(x)
        x = self.fc_1(x)
        x = self.fc_2(x)
        x = self.fc_3(x)
        x = self.fc_4(x)
        x = self.fc_5(x)
        return nn.functional.softmax(x, dim=1)
```

In [54]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
NUM_EPOCH = 500
LEARNING_RATE = 0.00022
print(device)
```

```
cuda
```

In [55]:
```
cnn_model = CNN(output_size=10, device=device).to(device)
# Use sparse_softmax_cross_entropy
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn_model.parameters(), lr=LEARNING_RATE)
```

In [56]:
```
torch.manual_seed(42)
torch.cuda.manual_seed(42)

for epoch in tqdm(range(NUM_EPOCH)):
    for i, (X, y) in enumerate(train_loader):
        X = X.to(device)
        y = y.to(device)
        y_pred = cnn_model(X)
        l = loss(y_pred, y)
        optimizer.zero_grad()
        l.backward()
        optimizer.step()
    if epoch % 25 == 0:
        print(f'Epoch: {epoch}, Loss: {l.item():.4f}')

    with torch.no_grad():
        correct = 0
        total = 0
        for X, y in test_loader:
            X = X.to(device)
            y = y.to(device)
            y_pred = cnn_model(X)
            _, predicted = torch.max(y_pred.data, 1)
            total += y.size(0)
            correct += (predicted == y).sum().item()
        if epoch % 25 == 0:
            print(f'Accuracy of the network on the {total} test images: {100 *
```

```
# y_pred = []
# y_true = []
# with torch.no_grad():
#     for X, y in test_loader:
#         X = X.to(device)
#         y = y.to(device)
#         y_pred.extend(torch.argmax(model(X), 1).tolist())
#         y_true.extend(y.tolist())

# cf_matrix = confusion_matrix(y_true, y_pred)
# print(classification_report(y_true, y_pred))
```

```
  0%|          | 0/500 [00:00<?, ?it/s]
Epoch: 0, Loss: 2.2489
Accuracy of the network on the 1998 test images: 16.97%
Epoch: 25, Loss: 1.9067
Accuracy of the network on the 1998 test images: 55.26%
Epoch: 50, Loss: 1.7566
Accuracy of the network on the 1998 test images: 66.27%
Epoch: 75, Loss: 1.6765
Accuracy of the network on the 1998 test images: 70.77%
Epoch: 100, Loss: 1.6382
Accuracy of the network on the 1998 test images: 75.93%
Epoch: 125, Loss: 1.6053
Accuracy of the network on the 1998 test images: 78.03%
Epoch: 150, Loss: 1.5623
Accuracy of the network on the 1998 test images: 80.53%
Epoch: 175, Loss: 1.5213
Accuracy of the network on the 1998 test images: 81.78%
Epoch: 200, Loss: 1.5076
Accuracy of the network on the 1998 test images: 81.68%
Epoch: 225, Loss: 1.5621
Accuracy of the network on the 1998 test images: 81.08%
Epoch: 250, Loss: 1.4938
Accuracy of the network on the 1998 test images: 81.13%
Epoch: 275, Loss: 1.5631
Accuracy of the network on the 1998 test images: 80.28%
Epoch: 300, Loss: 1.5012
Accuracy of the network on the 1998 test images: 84.08%
Epoch: 325, Loss: 1.5155
Accuracy of the network on the 1998 test images: 83.98%
Epoch: 350, Loss: 1.5148
Accuracy of the network on the 1998 test images: 83.98%
Epoch: 375, Loss: 1.4699
Accuracy of the network on the 1998 test images: 83.73%
Epoch: 400, Loss: 1.4945
Accuracy of the network on the 1998 test images: 83.43%
Epoch: 425, Loss: 1.5004
Accuracy of the network on the 1998 test images: 85.49%
Epoch: 450, Loss: 1.4978
Accuracy of the network on the 1998 test images: 81.83%
Epoch: 475, Loss: 1.4922
Accuracy of the network on the 1998 test images: 84.43%
```

Evaluation

Here are four best models of four different types of model: KNN, SVM, RNN and CNN. We will plot out the confusion matrix for each model and other benchmark metrices.

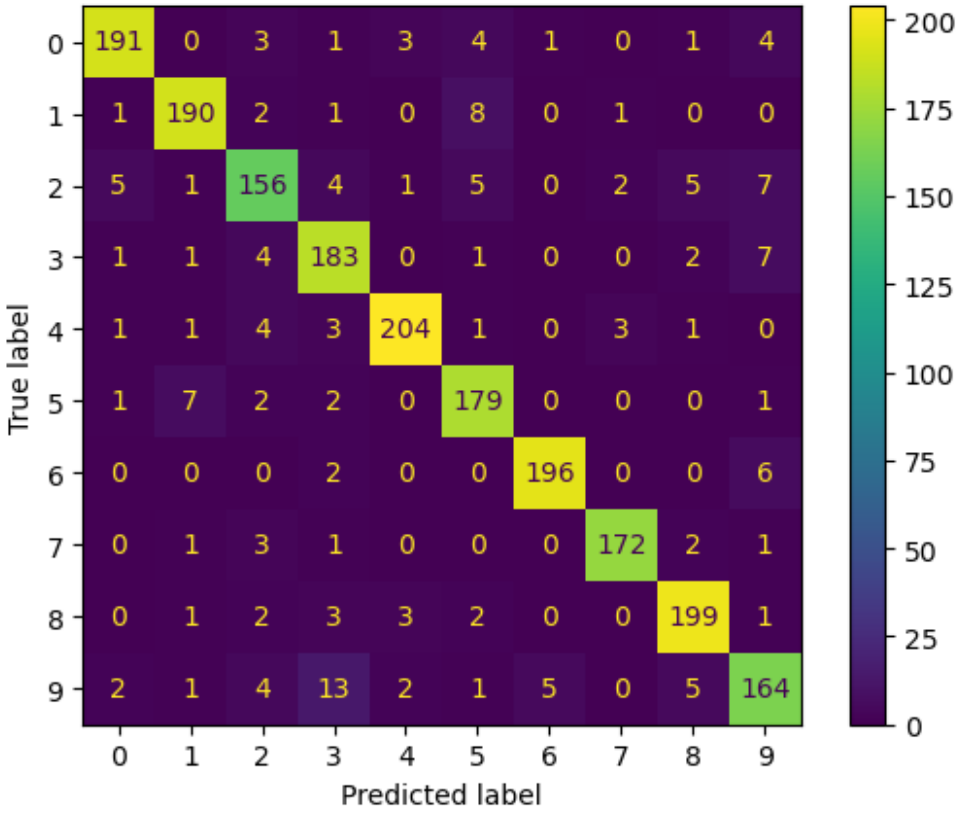
In [57]:
```
# Show the mapping between label and actual class names
le_name_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
le_name_mapping
```

Out[57]:
```
{'blues': 0,
 'classical': 1,
 'country': 2,
 'disco': 3,
 'hiphop': 4,
 'jazz': 5,
 'metal': 6,
 'pop': 7,
 'reggae': 8,
 'rock': 9}
```

- KNN

In [58]:
```
best_knn_model = KNeighborsClassifier(n_neighbors=1, weights='uniform', algori
best_knn_model.fit(X_train, y_train)
y_pred_knn = best_knn_model.predict(X_test)
knn_acc = accuracy_score(y_test, y_pred_knn)
print(f'KNN Accuracy: {knn_acc*100:.2f}%')
print(classification_report(y_test, y_pred_knn))
knn_cm = confusion_matrix(y_true=y_test, y_pred=y_pred_knn)
ConfusionMatrixDisplay(knn_cm).plot();
```

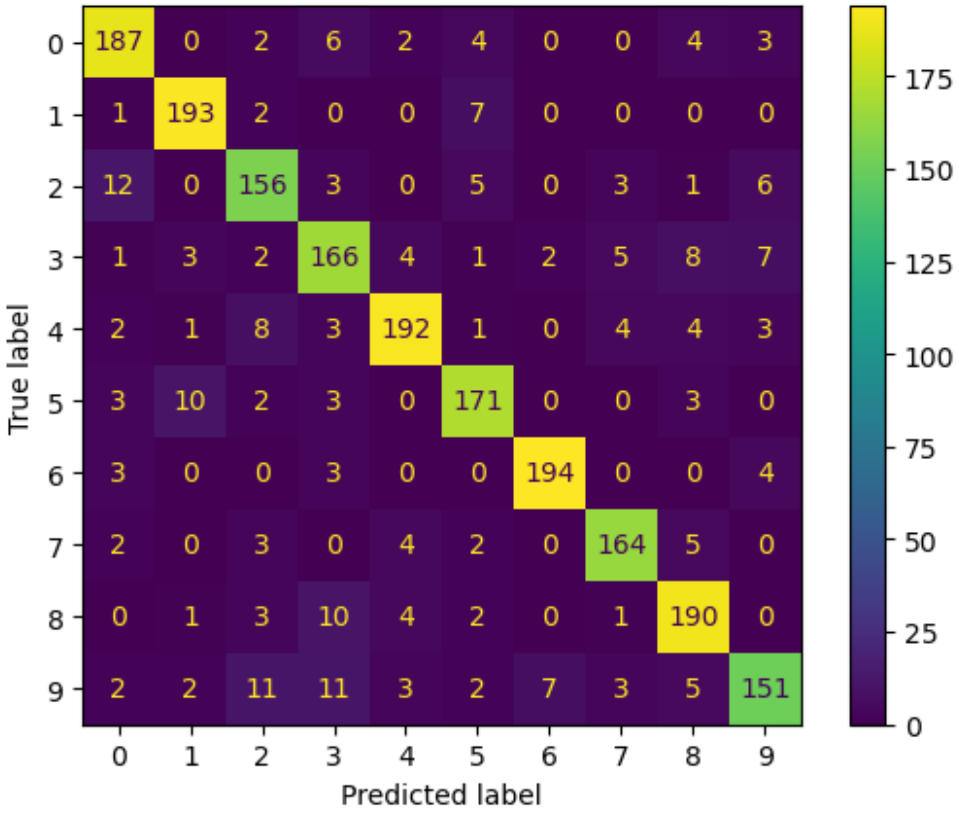
```
KNN Accuracy: 91.79%
              precision    recall  f1-score   support

           0       0.95      0.92      0.93       208
           1       0.94      0.94      0.94       203
           2       0.87      0.84      0.85       186
           3       0.86      0.92      0.89       199
           4       0.96      0.94      0.95       218
           5       0.89      0.93      0.91       192
           6       0.97      0.96      0.97       204
           7       0.97      0.96      0.96       180
           8       0.93      0.94      0.93       211
           9       0.86      0.83      0.85       197

    accuracy                           0.92      1998
   macro avg       0.92      0.92      0.92      1998
weighted avg       0.92      0.92      0.92      1998
```



- SVM

In [59]:
```
best_svm_model = SVC(C=0.1, degree=3, gamma=0.1, kernel='poly', random_state=RA
best_svm_model.fit(X_train, y_train)
y_pred_svm = best_svm_model.predict(X_test)
svm_acc = accuracy_score(y_test, y_pred_svm)
print(f'Bagging SVM Accuracy: {svm_acc*100:.2f}%')
print(classification_report(y_test, y_pred_svm))
svm_cm = confusion_matrix(y_true=y_test, y_pred=y_pred_svm)
ConfusionMatrixDisplay(svm_cm).plot();
```

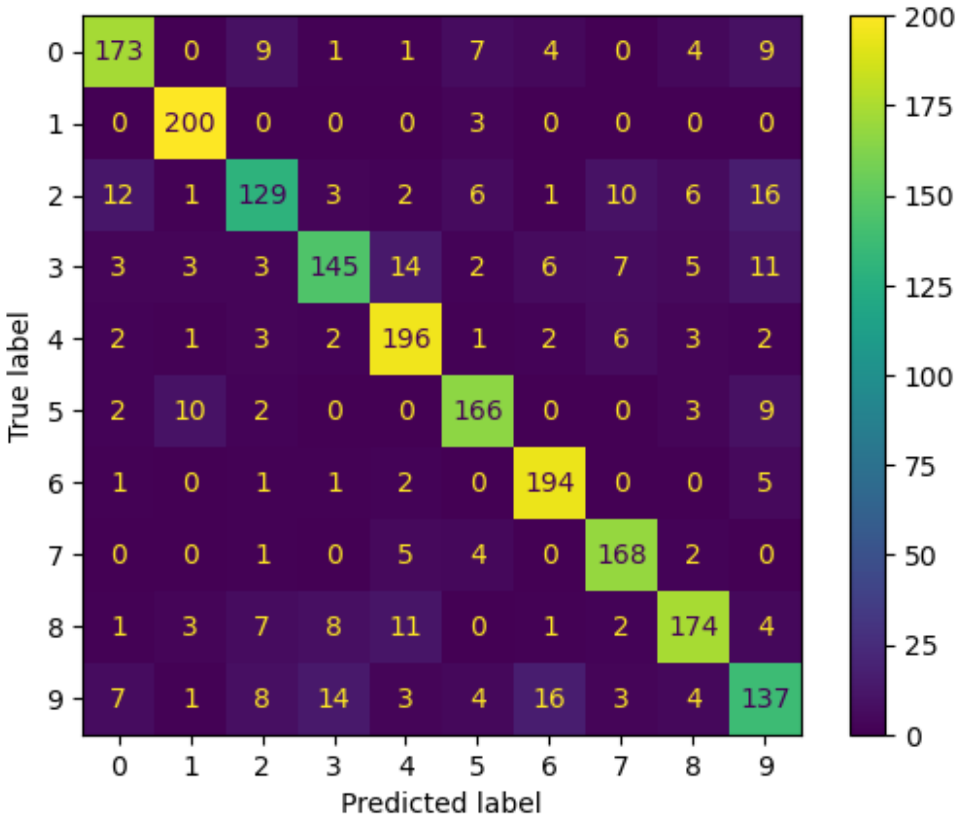
```
Bagging SVM Accuracy: 88.29%
              precision    recall  f1-score   support

           0       0.88      0.90      0.89       208
           1       0.92      0.95      0.93       203
           2       0.83      0.84      0.83       186
           3       0.81      0.83      0.82       199
           4       0.92      0.88      0.90       218
           5       0.88      0.89      0.88       192
           6       0.96      0.95      0.95       204
           7       0.91      0.91      0.91       180
           8       0.86      0.90      0.88       211
           9       0.87      0.77      0.81       197

    accuracy                           0.88      1998
   macro avg       0.88      0.88      0.88      1998
weighted avg       0.88      0.88      0.88      1998
```



- RNN

In [60]:
```
y_pred_rnn = []
y_true_rnn = []

rnn_model.eval()
with torch.no_grad():
    test_acc_rnn = 0

    for X, y in test_loader_rnn:
        X = X.to(device)
        y = y.to(device)

        y_pred = rnn_model(X)
        y_pred = torch.softmax(y_pred, dim=1).argmax(dim=1)
        test_acc_rnn += (y==y_pred).sum() / len(y)

        y_pred_rnn.extend(y_pred.tolist())
        y_true_rnn.extend(y.tolist())

test_acc_rnn = test_acc_rnn / len(test_loader_rnn)
print(f'RNN Accuracy: {test_acc_rnn*100:.2f}%')
print(classification_report(y_true_rnn, y_pred_rnn))

rnn_cm = confusion_matrix(y_true=y_true_rnn, y_pred=y_pred_rnn)
ConfusionMatrixDisplay(rnn_cm).plot();
```

```
RNN Accuracy: 84.04%
              precision    recall  f1-score   support

           0       0.86      0.83      0.85       208
           1       0.91      0.99      0.95       203
           2       0.79      0.69      0.74       186
           3       0.83      0.73      0.78       199
           4       0.84      0.90      0.87       218
           5       0.86      0.86      0.86       192
           6       0.87      0.95      0.91       204
           7       0.86      0.93      0.89       180
           8       0.87      0.82      0.84       211
           9       0.71      0.70      0.70       197

    accuracy                           0.84      1998
   macro avg       0.84      0.84      0.84      1998
weighted avg       0.84      0.84      0.84      1998
```

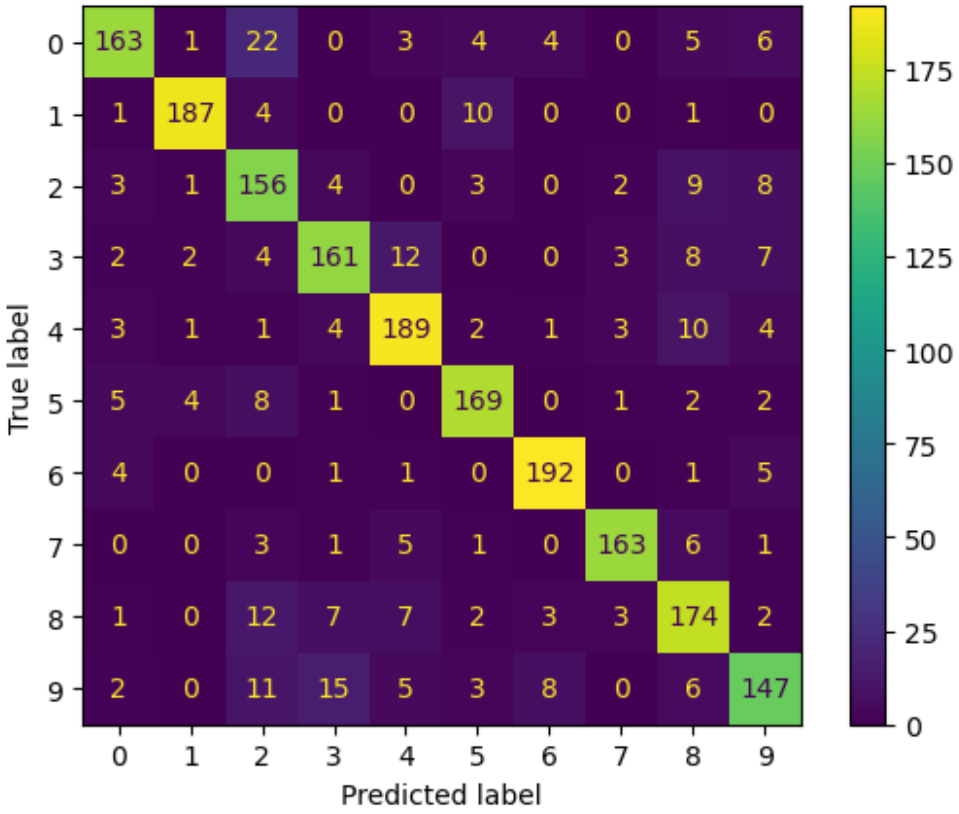


- CNN

In [61]:
```
y_pred_cnn = []
y_true_cnn = []

rnn_model.eval()
with torch.no_grad():
    test_acc_cnn = 0

    for X, y in test_loader:
        X = X.to(device)
        y = y.to(device)
```

```
        y_pred = cnn_model(X)
        y_pred = torch.softmax(y_pred, dim=1).argmax(dim=1)
        test_acc_cnn += (y==y_pred).sum() / len(y)

        y_pred_cnn.extend(y_pred.tolist())
        y_true_cnn.extend(y.tolist())

test_acc_cnn = test_acc_cnn / len(test_loader)
print(f'CNN Accuracy: {test_acc_cnn*100:.2f}%')
print(classification_report(y_true_cnn, y_pred_cnn))

cnn_cm = confusion_matrix(y_true=y_true_cnn, y_pred=y_pred_cnn)
ConfusionMatrixDisplay(cnn_cm).plot();
```

```
CNN Accuracy: 85.13%
              precision    recall  f1-score   support

           0       0.89      0.78      0.83       208
           1       0.95      0.92      0.94       203
           2       0.71      0.84      0.77       186
           3       0.83      0.81      0.82       199
           4       0.85      0.87      0.86       218
           5       0.87      0.88      0.88       192
           6       0.92      0.94      0.93       204
           7       0.93      0.91      0.92       180
           8       0.78      0.82      0.80       211
           9       0.81      0.75      0.78       197

    accuracy                           0.85      1998
   macro avg       0.85      0.85      0.85      1998
weighted avg       0.85      0.85      0.85      1998
```



We have builded and trained four different types of model, with each model using the best hyperparameters we tested. For KNN and SVM models, we used grid search and for CNN and RNN we manually tuned the hyperparameters. There are several evaluation metrics we can use to estimate the overall performance of the models we trained. However, since the models we build are for multiclassification problem, we will only use the accuracy as the main evaluation metric. Therefore the **best model** for this classification problem is **KNN**, which has the highest accuracy score which is 91.79%, following by SVM with 88.29% of accuracy.

We also plotted out the confusion matrics of these four different models. It is interesting to find that all four model have a hard time classifying label 9 which is rock music. All four have classfied several rock music to country is disco.

Conclusion

Overall, the KNN model achieved the highest accuracy (91.79%), followed by Bagging SVM (88.29%), RNN (86.21%), and CNN (85.94%). All of our 4 models achieved relatively high accuracy and balanced scores for precision, recall, and f1-score across most music genres, with some weaknesses in working with label 2 -- `country` music and label 9 -- `rock`. This indicates that the model performed generally well in accurately classifying different music genres, with a high level of precision and recall, while further evaluation and fine-tuning may be required to optimize the models for specific genre classification tasks.

Considering that music genre classification is a complex task with multiple dimensions and features that are hard to operationalize, our group is satisfied with the models' performance. The KNN models generally outperformed the RNN and CNN models in terms of accuracy and overall precision-recall balance. However, it's important to note that the performance and time cost may vary depending on the specific music genres and dataset used. Through our experience, with GPU, running RNN and CNN models can be very fast, while SVM is still relatively slow. Thus, making SVM the least preferred model when we have access to GPU.

Ethics & Privacy

The dataset we get is a free dataset named GTZAN, the MNIST of sounds, from Kaggle which should not give any privacy concern since many music genre recognition ML models are trained on this dataset. However, if in future we are going to train a larger model based on more data, copy right may be one of the issue related to privacy.

As music creation develops, the boundries between genres get vague. More and more music have mixed styles and new genres will be created. While it makes the classification harder, classifying songs into specific genres may raise social issues at this time. Many song writeer may not be happy with their songs being classified into a specific genre. Labeling the dataset may become a harder work in the future.

If a powerful MGR model is developed and used in recommendation system of music, it may cause people to have a music taste bias as the system can always find the music that fit users current taste, thus reduces the chance for user to find new types music that they may like.

Team Expectations

- *Weekly meetings on Sundays on general progress check*
- *Bi-weekly quick meetings on Wednesdays before each check-point submission*
- *Frequent discussion through online platforms (comments inside notebook, text, zoom meetings, etc.)*

Project Timeline Proposal

Meeting Date	Meeting Time	Completed Before Meeting	Discuss at Meeting
5/14	7 PM	Determine best form of communication; Brainstorm topics/questions	Decide on final project topic (all); discuss ideal datasets and ethics (all); do background research (Xiaoxuan)

Meeting Date	Meeting Time	Completed Before Meeting	Discuss at Meeting
5/16	10 PM	Do background research on topic	Draft project proposal (Xiaoxuan, Jiayi)
5/17	6 PM	Draft project proposal (all)	Edit, finalize, and submit proposal (all)
5/19	10 PM	Search for extra datasets (Xiaoxuan)	Assign group members to lead each specific part (Xiaoxuan)
5/25	4 PM	Discuss Wrangling and possible analytical approaches	EDA (Yunxiang, Xiaoxuan)
5/28	6 PM	Import & Wrangle Data , EDA (Xiaoxuan)	Review/Edit wrangling/EDA; Discuss Analysis Plan (all)
5/31	7 PM	Finalize wrangling/EDA (Xiaoxuan, Jiayi)	Checkpoint submission (all)
6/2	7 PM	N/A	Discuss algorithms for the project (all)
6/5	3 PM	Programming of all models (all)	Discuss/edit algorithms (all)
6/9	9 PM	Algorithm implementation (all)	Updates from different algorithms (all)
6/12	2 PM	Model selection (all)	Optimization (all)
6/14	8 PM	Complete analysis; Draft results/conclusion/discussion	Complete project; Turn in Final Project (all)

Team Contribution

Exploratory Data Analysis / Data Visualization: Yunxiang Chi, Xiaoxuan Zhang, Jiayi Dong

K- Nearest Neighbors: Elaine Ge

Support Vector Machine: Yunxiang Chi

Convolutional Neural Network: Xiaoyan He, Xiaoxuan Zhang

Recurrent Neural Network: Jiayi Dong

Evaluation Metrics: Jiayi Dong

Conclusion & Discussion; Miscellaneous: Xiaoxuan Zhang

Footnotes

1.^: GTZAN Dataset - Music Genre Classification.
https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification