University of Thessaly

Speech and Audio Processing

ECE443

Music Genre Recognition Project

Alexandra Gianni ID: 3382
Fani Banou ID: 3322

February 20, 2025

Contents

1 Introduction

1.1 Objectives

The primary objective of this project is to create and develop a robust, automated music genre recognition system using advanced audio processing techniques. More accurately, the main goals of the project are to:

- Extract relevant spectral features from audio signals by computing **Mel-Frequency Cepstral Coefficients (MFCCs)** with a 20ms frame length and 5ms step length.
- Train **Gaussian Mixture Models (GMMs)** for each music genre using the *Expectation-Maximization (EM)* algorithm.
- Implement a classification framework based on the *Maximum a Posteriori (MAP)* criterion, assigning genres based on the highest likehood.
- Evaluate the performance of the system by comparing accuracy across different GMM configurations and analyzing the impact of model complexity on classification performance.

1.2 Project overview

In this project, a comprehensive pipeline for music genre recognition is implemented, which begins with unprocessed (raw) audio files and concludes with genre classification. Initially, the process begins with feature extraction, where audio signals are segmented into short frames (20ms) with a 5ms overlap to capture the dynamic spectral properties of the music. MFCCs are then computed from these frames to provide a compact representation of the audio's frequency content, reflecting perceptual features of human hearing. In other words, it mimics human hearing. For classification, the system computes the log-likelihood of an input audio sample using each genre-specific GMM and assigns the genre with the **highest** probability.The entire system is developed using Python, utilizing libraries like *librosa* for audio processing. The project also includes a number of experiments that examine the impact of altering the number of Gaussian components in the GMMs, offering information on the approach's limitations and resilience. The methodology, experimental findings, and evaluation of the system's performance in relation to music genre recognition are all covered in length in this report.

2 Methodology

This section describes the steps followed to implement the music genre classification system. Data preparation, feature extraction, model training, classification, and evaluation are included in these steps. The approach leverages Mel-Frequency Cepstral Coefficients (MFCCs) for feature extraction and Gaussian Mixture Models (GMMs) for classification.

2.1 Data Preparation

The dataset consists of multiple audio files categorized into 3 different musical genres (blues, classical, reggae). The dataset is structured into separate folders, *Train* and *Test*. Each one of these two contains subfolders for individual genres. Sequentially, each genre folder contains multiple *.wav* files representing musical tracks from that genre. To ensure proper dataset organization, the program dynamically verifies the existence of the required directories and,if necessary, creates them. Each genre's training and testing audio files are processed separately, securing a clean separation between training and evaluating data.

2.2 Feature Extraction

The system begins by extracting Mel-Frequency Cepstral Coefficients (MFCCs) from each audio file. MFCCs are computed using a frame length of 20ms and a hop of 5ms (by requirements), effectively capturing short-term spectral characteristics of the audio signal.

To extract MFCCs, each audio file is loaded using the *librosa* library while preserving its original sampling rate. Afterwards, the computed MFCCs are stored in a matrix of shape $[N_{MFCC}, T]$, where $N_{MFCC} = 13$ is the number of MFCC coefficients and T is the number of frames.

```python
def extract_mfcc(file_path):
    # Load with original sample rate
    y, sr = librosa.load(file_path, sr=None)

    # Convert ms to samples
    frame_length = int(FRAME_SIZE * sr)
    hop_length = int(HOP_LENGTH * sr)

    # Compute MFCC features
    mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=NUM_MFCC, n_fft=frame_length,
        hop_length=hop_length)
    return mfccs
```

Extracted features are saved as `.mat` files for later processing, ensuring efficient storage and retrieval. The program automatically organizes these feature files under the `mfcc_features` directory, categorized into `train` and `test` folders based on the dataset split. The dataset is processed as:

```python
def process_dataset(dataset_type):
    # Construct the path to the dataset (either training or testing)
    dataset_path = os.path.join(DATASET_PATH, dataset_type)
    # Path where extracted MFCCs will be saved
    mfcc_path = os.path.join(MFCC_SAVE_PATH, dataset_type)

    # Loop through each genre in the dataset
    for genre in os.listdir(dataset_path):
        genre_path = os.path.join(dataset_path, genre)
        genre_mfcc_path = os.path.join(mfcc_path, genre)

        # Skip non-folder files
        if not os.path.isdir(genre_path):
            continue

        # Ensure the directory exists for saving MFCCs
        os.makedirs(genre_mfcc_path, exist_ok=True)

        # Loop through each audio file in the genre folder
        for file in os.listdir(genre_path):
            # Only process '.wav' audio files
            if file.endswith(".wav"):
                file_path = os.path.join(genre_path, file)

                # Extract MFCC features from the audio file
                mfcc_features = extract_mfcc(file_path)

                if mfcc_features is not None:
                    # Create a filename with a '.mat' extension for saving MFCCs
                    save_filename = os.path.splitext(file)[0] + ".mat"
                    save_path = os.path.join(genre_mfcc_path, save_filename)

                    # Save extracted MFCC features in MATLAB (.mat) format
                    scipy.io.savemat(save_path, {"mfcc": mfcc_features})

                    # Print confirmation message
                    print(f"Saved: {save_path}")
```

2.3 GMM Training

Once the MFCC features are extracted, a Gaussian Mixture Model (GMM) is trained for each genre. The GMM is trained using the **Expectation-Maximization (EM) algorithm**, with **K-Means clustering** for parameter initialization. The trained models are saved and later used for classification.

The training process consists of:

1. **initialize_gmm(X, n_components)** - Initializes GMM parameters using **K-Means clustering**.
2. **e_step(X, means, covariances, priors)** - Performs the **Expectation step** to compute responsibilities.
3. **m_step(X, gamma)** - Performs the **Maximization step** to update GMM parameters.
4. **train_gmm(X, n_components)** - Iteratively trains a GMM using the **EM algorithm** until convergence.
5. **train_gmm_models(genre_mfccs)** - Trains a separate GMM for each music genre and saves the models.

Initializing GMM Parameters

Before training, we must initialize the GMM parameters: **means, covariances, and priors**. Instead of random initialization, we use **K-Means clustering** to obtain well-separated initial cluster means.

Process:

- Apply **K-Means clustering** to partition the data into $n\_components$ clusters.
- Use the K-Means **cluster centers as initial means**.
- Compute **initial covariance matrices** for each cluster.
- Compute **prior probabilities** based on the number of data points in each cluster.

Equation for Priors:

$$P(q_k) = \frac{N_k}{N} \tag{1}$$

where:

- N_k is the number of points assigned to cluster k.
- N is the total number of data points.

It is implemented on the following snipet code:

```python
def initialize_gmm(X, n_components):
    # Apply K-Means clustering to group data into n_components clusters
    kmeans = KMeans(n_clusters=n_components, n_init=10, random_state=42)
    # Assigns each sample to a cluster
    labels = kmeans.fit_predict(X)
    # Use K-Means cluster centers as initial means
    means = kmeans.cluster_centers_

    # Compute initial covariance matrices for each cluster
    covariances = np.array([
        np.cov(X[labels == k].T) + np.eye(X.shape[1]) * 1e-6  # Regularization to
            avoid singularity
        for k in range(n_components)
    ])

    # Compute prior probabilities based on the proportion of data points in each
       cluster
    cluster_counts = np.bincount(labels, minlength=n_components)  # Count points per
       cluster
    priors = cluster_counts / X.shape[0]  # Normalize to get probabilities

    # Return the initialized parameters
    return means, covariances, priors
```

Expectation Step

The **E-step** computes the **responsibilities** (posterior probabilities) of each data point belonging to each Gaussian component.

Equation for Responsibilities:

$$\gamma_{nk} = \frac{P(q_k) \cdot \mathcal{N}(x_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} P(q_j) \cdot \mathcal{N}(x_n|\mu_j, \Sigma_j)} \tag{2}$$

where:

- $\mathcal{N}(x_n|\mu_k, \Sigma_k)$ is the Gaussian probability density function.

It is implemented on the following snipet code:

```python
def e_step(X, means, covariances, priors):
    N = X.shape[0]  # Number of data points
    K = len(priors)  # Number of Gaussian components

    # Initialize the responsibility matrix
    gamma = np.zeros((N, K))

    # Compute responsibilities for each Gaussian component
    for k in range(K):
        # Compute the likelihood of each data point under Gaussian k
        likelihood = multivariate_normal.pdf(X, mean=means[k], cov=covariances[k])

        # Multiply by the prior probability of the component
        gamma[:, k] = priors[k] * likelihood
    # Normalize responsibilities
    gamma /= gamma.sum(axis=1, keepdims=True)

    # Return the responsibility matrix
    return gamma
```

Maximization Step

The **M-step** updates the GMM parameters (means, covariances, and priors) using the computed **responsibilities**.

Equations:

- **Mean Update:**

$$\mu_k = \frac{\sum_{n=1}^{N} \gamma_{nk} x_n}{\sum_{n=1}^{N} \gamma_{nk}} \tag{3}$$

- **Covariance Update:**

$$\Sigma_k = \frac{\sum_{n=1}^{N} \gamma_{nk} (x_n - \mu_k)(x_n - \mu_k)^T}{\sum_{n=1}^{N} \gamma_{nk}} \tag{4}$$

- **Prior Update:**

$$P(q_k) = \frac{\sum_{n=1}^{N} \gamma_{nk}}{N} \tag{5}$$

It is implemented on the following snipet code:

```
def m_step(X, gamma):
    # Compute the effective number of points assigned to each cluster
    Nk = gamma.sum(axis=0)  # Sum of responsibilities for each Gaussian component

    # Update means: Compute weighted average of data points assigned to each Gaussian
    means = np.array([
        np.sum(gamma[:, k][:, np.newaxis] * X, axis=0) / Nk[k]
        for k in range(len(Nk))
    ])

    # Update covariances: Compute weighted covariance matrix for each Gaussian
       component
    covariances = np.array([
        np.sum(gamma[:, k][:, np.newaxis, np.newaxis] *
               (X - means[k])[:, :, np.newaxis] @ (X - means[k])[:, np.newaxis, :],
                 axis=0) / Nk[k]
        for k in range(len(Nk))
    ])

    # Update priors: The proportion of points assigned to each Gaussian
    priors = Nk / X.shape[0]

    return means, covariances, priors
```

Training GMM Using EM Algorithm

This function performs **iterative EM updates** to train a GMM until **log-likelihood convergence** is achieved.

Equation for Log-Likelihood:

$$\log P(X|\theta) = \sum_{n=1}^{N} \log \sum_{k=1}^{K} P(q_k)\mathcal{N}(x_n|\mu_k, \Sigma_k) \tag{6}$$

It is implemented on the following snipet code:

```python
def train_gmm(X, n_components):
    # Initialize GMM parameters using K-Means clustering
    means, covariances, priors = initialize_gmm(X, n_components)

    # Initialize previous log-likelihood for convergence check
    prev_log_likelihood = None

    # Iterate through the EM algorithm
    for iteration in range(MAX_ITER):

        # E-step: Compute responsibilities based on current GMM parameters
        gamma = e_step(X, means, covariances, priors)

        # M-step: Update GMM parameters using computed responsibilities
        means, covariances, priors = m_step(X, gamma)

        # Compute the new log-likelihood to track model convergence
        log_likelihood = np.sum(np.log(np.sum([
            priors[k] * multivariate_normal.pdf(X, mean=means[k], cov=covariances[k])
            for k in range(n_components)
        ], axis=0)))

        # Display log-likelihood progress
        print(f"Iteration␣{iteration␣+␣1},␣Log-Likelihood:␣{log_likelihood:.6f}")

        # Check for convergence
        if prev_log_likelihood is not None and abs(log_likelihood -
            prev_log_likelihood) < TOL:
            print("Convergence␣reached.")
            break  # Stop iterations if improvement is below threshold

        prev_log_likelihood = log_likelihood  # Update previous log-likelihood for
            next iteration

    # Return the optimized GMM parameters
    return means, covariances, priors
```

Training GMM Models for Each Genre (train_gmm_models)

This function **trains a separate GMM** for each genre using **MFCC features** extracted from music files.
Process:

- Loop through each **genre** in the dataset.

- **Stack all MFCC feature arrays** for that genre into a single dataset.
- Train a **GMM using train_gmm**.
- **Save the trained model** (means, covariances, and priors) using `joblib`.

It is also implemented on the following snipet code:

```python
def train_gmm_models(genre_mfccs):
    # Dictionary to store trained GMM models for each genre
    models = {}

    # Iterate over each genre in the dataset
    for genre, mfcc_list in genre_mfccs.items():
        print(f"Training GMM for genre: {genre}...")

        # Stack all MFCC feature arrays for this genre
        all_mfccs = np.vstack(mfcc_list)

        # Train the GMM model using the EM algorithm
        means, covariances, priors = train_gmm(all_mfccs, NUM_GAUSSIANS)

        # Save the trained GMM model parameters
        model_filename = os.path.join(GMM_SAVE_PATH, f"gmm_{genre}.pkl")
        joblib.dump((means, covariances, priors), model_filename)
        models[genre] = (means, covariances, priors)

        print(f"Saved model: {model_filename}")

    # Return trained models
    return models
```

Each trained GMM is stored as a *.pkl* file in the *gmm_models* directory.

2.4 Classification

Once trained, the GMMs are used to classify new audio samples. The classification process involves:

1. Extracting MFCC features from the test file.
2. Calculating the log-likelihood of the test sample under each trained GMM.
3. Assigning the genre corresponding to the highest log-likelihood.

```python
def classify_audio(mfcc, models):
    # Dictionary to store the log-likelihood for each genre
    log_likelihoods = {}

    # Iterate through each trained GMM model
    for genre, (means, covariances, priors) in models.items():

        # Compute log-likelihood for this genre
        likelihoods = np.array([
            priors[k] * multivariate_normal.pdf(mfcc.T, mean=means[k], cov=
                covariances[k])
            for k in range(len(priors))
        ])

        # Sum over all Gaussian components and take log
        log_likelihoods[genre] = np.sum(np.log(np.sum(likelihoods, axis=0)))

    # Return the genre with the highest log-likelihood
    return max(log_likelihoods, key=log_likelihoods.get)
```

2.5 Evaluation Metrics

The evaluation function follows these key steps:

1. Load the **pre-trained GMM models** for each genre.
2. Extract **MFCC features** from the test dataset.
3. Classify each test file using the trained models.
4. Construct a **confusion matrix** to analyze classification performance.
5. Compute the **classification accuracy**.

Classification Accuracy

The classification accuracy is computed as:

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}} \times 100 \tag{7}$$

where:

- **Correct Predictions** refers to the number of correctly classified audio samples.
- **Total Predictions** refers to the total number of test samples.

Confusion Matrix

The confusion matrix is a table that summarizes the performance of the classification model. Each row represents the actual genre, and each column represents the predicted genre. If the model performs perfectly, the confusion matrix will be diagonal.

Example structure:

$$\begin{bmatrix} \text{Blues} & 15 & 3 & 2 \\ \text{Classical} & 2 & 17 & 1 \\ \text{Reggae} & 1 & 1 & 18 \end{bmatrix}$$

where each entry (i, j) represents the number of samples from the actual genre i that were predicted as genre j.

It is implemented on the following snipet code:

```
def evaluate_model():
    # Load trained GMM models
    models = load_gmm_models()

    # Initialize counters for accuracy and confusion matrix
    total_samples = 0
    correct_predictions = 0
    confusion_matrix = {}

    # Iterate through each genre in the test dataset
    for genre in os.listdir(os.path.join(DATASET_PATH, "test")):
        genre_path = os.path.join(DATASET_PATH, "test", genre)
        print(f"Processing␣test␣samples␣for␣genre:␣{genre}")

        # Skip non-folder files
        if not os.path.isdir(genre_path):
            continue

        # Initialize confusion matrix entry for this genre
        if genre not in confusion_matrix:
            confusion_matrix[genre] = {}

        # Iterate through each audio file in the genre folder
        for file in os.listdir(genre_path):
            if file.endswith(".wav"):
                total_samples += 1  # Count the total number of test samples

                file_path = os.path.join(genre_path, file)

                # Extract MFCC features from the test file
                mfcc_features = extract_mfcc(file_path)

                if mfcc_features is not None:
                    # Classify the test file
                    predicted_genre = classify_audio(mfcc_features, models)

                    print(f"Testing␣file:␣{file},␣True␣genre:␣{genre},␣Predicted␣
                        genre:␣{predicted_genre}")

                    # Update confusion matrix
                    if predicted_genre not in confusion_matrix[genre]:
                        confusion_matrix[genre][predicted_genre] = 0

                    confusion_matrix[genre][predicted_genre] += 1

                    # Check if the prediction is correct
                    if predicted_genre == genre:
                        correct_predictions += 1

    # Calculate classification accuracy
    accuracy = (correct_predictions / total_samples) * 100 if total_samples > 0 else
        0
    print(f"\nModel␣Classification␣Accuracy:␣{accuracy:.2f}%")

    # Print confusion matrix
    print("\nConfusion␣Matrix:")
    for genre, predictions in confusion_matrix.items():
        print(f"{genre}:␣{predictions}")
```

3 Experiments and Results

The primary objective of this experiment is to analyze the impact of different numbers of Gaussian components (K) on the performance of the Gaussian Mixture Model (GMM) for music genre classification. We evaluate the classification accuracy for varying values of K to determine the optimal trade-off between model complexity and performance.

3.1 Experimental Setup

The dataset used for training and testing is split using an **80-20** ratio, meaning that 80% of the data is used for training and 20% for testing. However, instead of selecting test samples randomly, we take the **last files** from each genre, which may introduce a bias in the results. This method ensures that the dataset split is deterministic but does not guarantee randomness, potentially affecting the generalizability of the trained models.

We conduct the experiment by training and evaluating three different GMM classifiers with the following values of K:

- **Low-order GMM:** $K = 4$
- **Medium-order GMM:** $K = 8$
- **High-order GMM:** $K = 16$

For each configuration, the GMM is trained using the Expectation-Maximization (EM) algorithm with K-Means initialization, and the classification accuracy is computed on the test dataset. The evaluation metric used is classification accuracy.

3.2 Results

The classification accuracy obtained for different values of K is presented in Table 1.

Number of Gaussians (K)	**Classification Accuracy (%)**
4	91.67
8	90.00
16	86.67

Table 1: Comparison of classification accuracy for different numbers of Gaussians

To further analyze the performance of each model, we present the confusion matrices for different values of K. These matrices indicate how well the classifier differentiates between genres and highlight any misclassifications.

Actual \Predicted	**Blues**	**Classical**	**Reggae**
Blues	16	0	4
Classical	0	19	1
Reggae	0	0	20

Table 2: Confusion Matrix for $K = 4$

Actual \Predicted	**Blues**	**Classical**	**Reggae**
Blues	15	1	4
Classical	0	19	1
Reggae	0	0	20

Table 3: Confusion Matrix for $K = 8$

Actual \Predicted	**Blues**	**Classical**	**Reggae**
Blues	13	0	7
Classical	0	19	1
Reggae	0	0	20

Table 4: Confusion Matrix for $K = 16$

To visualize the trend of classification accuracy with increasing K, we plot the accuracy values against the number of Gaussian components in Figure 1.

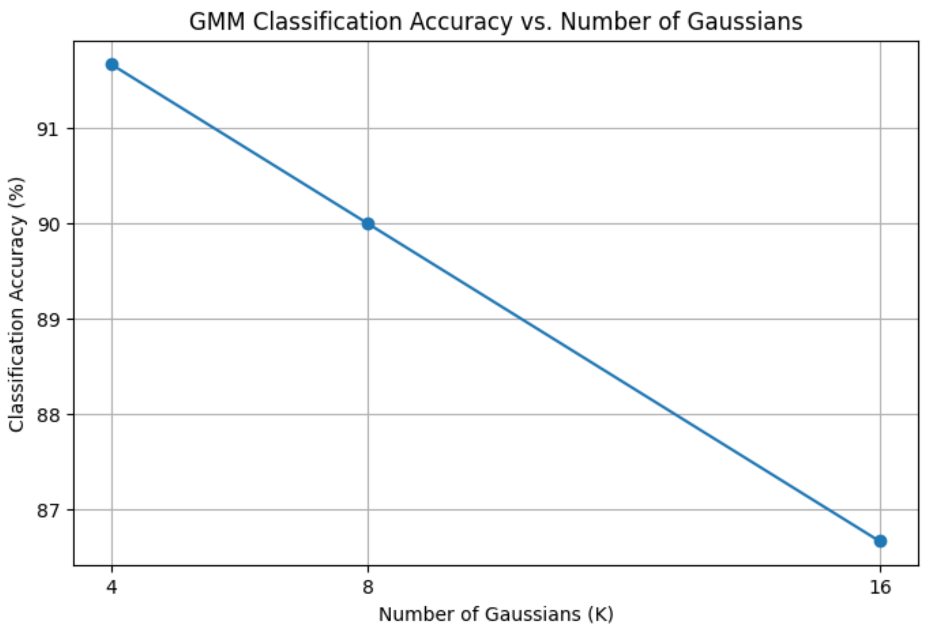

Figure 1: Classification Accuracy vs. Number of Gaussians (K)

3.3 Observations

From the experimental results presented in Table 1 and the confusion matrices (Tables 2 - 4), the following key observations can be made:

Classification Accuracy vs. Number of Gaussians (K)

- The classification accuracy **decreases** as the number of Gaussian components increases.
- The best accuracy (**91.67%**) is achieved with $K = 4$, while accuracy drops to **90.00%** for $K = 8$ and further declines to **86.67%** for $K = 16$.
- This trend suggests that a **lower number of Gaussian components generalizes better**, whereas a **higher K may lead to overfitting** to the training data.

Confusion Matrix Analysis

- Across all values of K, the **Reggae and Classical genres** are classified with **high accuracy**, as they have minimal misclassifications.
- The **Blues genre exhibits the highest misclassification rate**, particularly for $K = 16$, where it is confused with both Classical and Reggae.
- As K increases, **Blues becomes more prone to misclassification**, indicating that increasing the model complexity does not necessarily improve class separation.

Effect of Model Complexity

- $K = 4$ **achieves the best overall performance**, with Blues being correctly classified **16 times** and misclassified **4 times** as Reggae.
- $K = 8$ **shows a slight decline** in Blues classification accuracy, with **15 correct classifications** and an increase in misclassifications.
- $K = 16$ **performs the worst for Blues**, classifying it correctly only **13 times**, while misclassifying it as Classical **7 times**.
- This suggests that **higher complexity does not always translate to better discrimination** but instead **may increase the risk of overfitting**.

4 Conclusion

This study implemented and analyzed a **Gaussian Mixture Model (GMM)-based classifier** for **music genre recognition** using **Mel-Frequency Cepstral Coefficients (MFCCs)** as feature representations. The **Expectation-Maximization (EM) algorithm** with **K-Means initialization** was used for training, and classification was performed using **Maximum Likelihood Estimation (MLE)**. Experiments with different numbers of Gaussian components ($K = 4, 8, 16$) showed that $K = 4$ **achieved the highest classification accuracy** while increasing K led to diminishing returns and higher misclassification rates, particularly for the **Blues genre**. Increasing the number of Gaussians does not improve accuracy and may introduce more misclassifications.