University of Thessaly

Speech and Audio Processing

Semester Project

Supervisor: Katerina Papadimitriou

Music Genre Classification

Author:

Theodora Poultsidou 03176

February 20, 2025

Introduction

In this project, I implement a music genre classification system. First, I extract the key audio features using Mel Frequency Cepstral Coefficients (MFCCs). Then, I train the Gaussian Mixture Models (GMMs) using a custom Expectation Maximization (EM) algorithm. Finally, I classify the genres using a Maximum a Posteriori (MAP) approach based on log-likelihood. Each step is implemented using the algorithms and formulas presented in the lecture.

Feature Extraction Using MFCCs

Dataset Split

The dataset consists of musical tracks from three genres(Blues, Raggae, Classical), with 100 songs per genre. I manually split the dataset into training and testing sets using an 80-20 ratio, meaning 80 songs per genre were used for training and 20 for testing.

MFCCs Extraction

For each audio file, I extracted MFCC features to use as input for the classification model. This step converts raw audio into multidimensionl features that capture important characteristics of the sound.

I used the `librosa` library to extract MFCCs with the following parameters:

- **Frame Size:** 0.02 seconds
- **Step Size:** 0.005 seconds
- **Number of MFCCs:** 13
- **Sampling Rate:** 22050 Hz

The `librosa.load()` function loads the audio file and resamples it to the given sampling rate. Then, the MFCC features are extracted using `librosa.feature.mfcc()`, with parameters defining the frame length, hop length, and number of coefficients.

Code Implementation:

```
def extract_mfcc(audio_path, frame_size=0.02, step_size=0.005, n_mfcc=13, sr=22050):
    signal, sample_rate = librosa.load(audio_path, sr=sr)
    frame_length = int(frame_size * sample_rate)
    hop_length = int(step_size * sample_rate)

    features = librosa.feature.mfcc(y=signal, sr=sample_rate,
                                    n_mfcc=n_mfcc, n_fft=frame_length,
                                    hop_length=hop_length).T
    return features
```

Now that the audio is processed and the features have been successfully extracted and saved, let's move on to how I trained the GMM.

Training GMMs

To implement Gaussian Mixture Models (GMMs), I used the Expectation-Maximization (EM) algorithm. The EM algorithm iteratively **estimates** the probability that each data point belongs to a particular Gaussian component (posterior) and then updates the model parameters (means, covariance, priors) to **maximize** the likelihood of the data.

Estimation Step in EM Algorithm

To update the posterior probabilities in the Expectation step of the EM algorithm, I implemented the following function. It computes the probability that each data point belongs to a particular Gaussian component, based on the given priors, means, and covariances. This corresponds to the following mathematical formula:

$$P(q_k^{(old)}|x_n, \Theta^{(old)}) = \frac{P(q_k^{(old)}|\Theta^{(old)}) \cdot p(x_n|\mu_k^{(old)}, \Sigma_k^{(old)})}{\sum_j P(q_j^{(old)}|\Theta^{(old)}) \cdot p(x_n|\mu_j^{(old)}, \Sigma_j^{(old)})} \tag{1}$$

Code Implementation:

```
def update_posterior(data_points, priors, means, covariances):

    number_of_data_points, number_of_features = data_points.shape
    number_of_gaussians = len(priors)

    # Compute the posterior for each data point and each Gaussian
    posterior = np.zeros((number_of_data_points, number_of_gaussians))

    for g in range(number_of_gaussians):
        # Evaluate the Gaussian pdf for each data point
        gaussian_pdf_values = multivariate_normal.pdf(
            data_points, mean=means[g], cov=covariances[g], allow_singular=True
        )
        # Multiply by the prior
        posterior[:, g] = priors[g] * gaussian_pdf_values

    # Normalize so that each data point's posterior sums to 1
    sum_posterior = np.sum(posterior, axis=1, keepdims=True)
    sum_posterior[sum_posterior == 0] = 1  # Avoid division by zero
    posterior /= sum_posterior

    return posterior
```

This function follows the Expectation step of the EM algorithm, where it calculates the posterior probabilities by evaluating the Gaussian probability density function (PDF) for each data point and normalizing the values so that they sum to one.

Maximization Step in EM Algorithm

In the **Maximization Step** of the Expectation-Maximization (EM) algorithm, we update the parameters of the Gaussian Mixture Model (GMM) using the posterior probabilities obtained in the Expectation Step. These parameters include:

1. The means μ_k
2. The covariance matrices Σ_k
3. The priors $P(q_k)$

Each parameter is updated as follows:

1. Updating the Means

The new mean for each Gaussian component is computed as the weighted sum of the data points, where the weights are the posterior probabilities:

$$\mu_k^{(new)} = \frac{\sum_{n=1}^{N} x_n P(q_k^{(old)}|x_n, \Theta^{(old)})}{\sum_{n=1}^{N} P(q_k^{(old)}|x_n, \Theta^{(old)})} \tag{2}$$

Code Implementation:

```
def update_means(data_points, posterior):
    number_of_data_points, number_of_features = data_points.shape
    number_of_gaussians = posterior.shape[1]

    # Initialize the means
    means = np.zeros((number_of_gaussians, number_of_features))

    # Sum of the posterior probabilities for each Gaussian
    sum_posteriors = np.sum(posterior, axis=0)

    for g in range(number_of_gaussians):
        # Compute the weighted sum of data points
        weighted_sum = np.sum(data_points * posterior[:, g][:, np.newaxis], axis=0)
        means[g] = weighted_sum / (sum_posteriors[g] + 1e-10)  # Avoid division by
            zero

    return means
```

Each new mean $\mu_k^{(new)}$ is computed as a weighted average of the data points, where the weight is given by the posterior probability that a data point belongs to that Gaussian.

2. Updating the Covariance Matrices

The covariance matrix for each Gaussian is updated as follows:

$$\Sigma_k^{(new)} = \frac{\sum_{n=1}^{N} P(q_k^{(old)}|x_n, \Theta^{(old)})(x_n - \mu_k^{(new)})(x_n - \mu_k^{(new)})^T}{\sum_{n=1}^{N} P(q_k^{(old)}|x_n, \Theta^{(old)})} \tag{3}$$

Code Implementation:

```
def update_covariances(data_points, posterior, means, covariance_type='full'):
    number_of_data_points, number_of_features = data_points.shape
    number_of_gaussians = posterior.shape[1]

    covariances = []
    sum_posteriors = np.sum(posterior, axis=0)

    for g in range(number_of_gaussians):
        # Compute the difference between each data point and the mean
        diff = data_points - means[g]

        # Weight each difference by the posterior probability for that Gaussian
        weighted_diff = diff * posterior[:, g][:, np.newaxis]

        # Compute the covariance matrix
        covariance_matrix = np.dot(weighted_diff.T, diff) / (sum_posteriors[g] + 1e-3)

        # If covariance_type is 'diag', set off-diagonal elements to zero
        if covariance_type == 'diag':
            covariance_matrix = np.diag(np.diag(covariance_matrix))

        covariances.append(covariance_matrix)

    return np.array(covariances)
```

Each covariance matrix $\Sigma_k^{(new)}$ is computed as the weighted sum of the outer product of the data point deviations from the mean, normalized by the sum of posterior probabilities for that Gaussian.

Specification: In the covariance update, the `covariance_type` parameter specifies how the covariance matrix is computed. If it is set to `'full'`, then a full covariance matrix is computed, capturing correlations between all the features. If it is set to `'diag'`, only the diagonal elements are kept, assuming that the features are uncorrelated. This simplifies computation and reduces the risk of overfitting. In the Experiment and Analysis section, we will compare both approaches to see which works best.

3. Updating the Priors

The prior probability of each Gaussian component is updated by averaging the posterior probabilities across all data points:

$$P(q_k^{(new)}|\Theta^{(new)}) = \frac{1}{N}\sum_{n=1}^{N} P(q_k^{(old)}|x_n, \Theta^{(old)}) \tag{4}$$

Code Implementation:

```
def update_priors(posterior):
    number_of_data_points, number_of_gaussians = posterior.shape

    # Calculate the number of data points assigned to each Gaussian (sum of the
        posterior)
    sum_posterior = np.sum(posterior, axis=0)

    # Compute the priors with a small value added to avoid division by zero
    priors = (sum_posterior + 1e-6) / posterior.shape[1]
    return priors
```

Each prior $P(q_k^{(new)})$ is updated as the normalized sum of posterior probabilities across all data points.

Training the GMM

Step 1: To train the GMM, we first initialize the parameters. The means come from K-means clustering to avoid randomness, with the number of clusters matching the number of Gaussians. The priors are set based on the fraction of points in each cluster. The covariances are calculated by looking at how the data in each cluster varies around its mean.

Step 2: Once initialized, the Expectation-Maximization (EM) algorithm iterates:

- **E-step**: Estimate the posterior probabilities for each data point belonging to each Gaussian.
- **M-step**: Update the priors, means, and covariances to maximize the likelihood, as explained earlier.

Step 3: This process repeats until convergence. If the difference in likelihood from the previous iteration is less than 10^{-8}, we assume the model has converged earlier than the maximum number of iterations, and the model is saved.

Classification and Evaluation

Classification

For classification, we use the **Maximum a Posteriori (MAP)** criterion, which selects the class with the highest posterior probability. Since all prior probabilities are equal, this simplifies to choosing the class with the highest **likelihood**. Given an input speech signal, the goal is to determine which trained class it belongs to.

Why Compute Log-Likelihood? In Gaussian Mixture Models (GMMs), the **likelihood** tells us how well the model explains the data. However, since likelihood values are often very small, multiplying many probabilities together can result in **numerical underflow**, meaning the values become too

small for the model to handle accurately.

To fix this, we compute the **log-likelihood** instead. Taking the logarithm converts multiplication into addition, making calculations more stable and preventing very small values from causing issues. The log-likelihood is given by:

$$\mathcal{L}(\Theta) = \log \sum_{k=1}^{K} P(q_k|X, \Theta)p(X|\Theta) \tag{5}$$

Code Implementation:

```
def compute_log_likelihood(data_points, priors, means, covariances):

    # Compute the likelihood for each data point and each Gaussian
    likelihood = np.zeros((data_points.shape[0], len(priors)))
    for g in range(len(priors)):
        likelihood[:, g] = priors[g] * multivariate_normal.pdf(
            data_points, mean=means[g], cov=covariances[g], allow_singular=True
        )
    return np.sum(np.log(np.sum(likelihood, axis=1) + 1e-6))  # Avoid log(0)
```

Evaluation Metrics

To measure how well our model performs, we use Accuracy, Precision, Recall and F1-score:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{6}$$

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}} \tag{7}$$

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}} \tag{8}$$

$$F1\text{-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{9}$$

Accuracy tells us how many predictions are correct overall, **precision** measures how many of the predicted positives are actually correct, **recall** measures how well the model identifies all actual positives, and **F1-score** balances precision and recall for a more reliable performance measure.

Experiments and Analysis

The data consists of multiple Gaussian distributions, so changing the number of Gaussians means adjusting how many we assume are present. This is why we use Gaussian Mixture Models (GMMs) instead of a single Gaussian. GMMs can better capture complex patterns by combining multiple distributions.

However, if we use too many Gaussians, there is a risk of overfitting , where the model learns noise instead of meaningful patterns in the data. I tuned the Gaussian Mixture Model (GMM) using the following parameters:

Experiment with covariance type: Full and different Gaussians

- **Maximum Iterations:** 150
- **Covariance Type:** Full (`'full'`)
- **Tolerance:** $1e^{-8}$ (Convergence threshold)
- **Number of Gaussians:** Exponentially increasing values: [1, 2, 4, 8, 16, 32]

Table 1: GMM Classification Performance for Different Gaussian Components

# Gaussians	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
1	83.87	86.45	83.87	83.10
2	85.48	86.81	85.48	85.33
4	85.48	86.37	85.48	85.37
8	79.03	81.04	79.03	78.86
16	74.19	79.07	74.19	73.96
32	77.42	81.42	77.42	77.10

Prediction Breakdown for Each Class (A written Confussion Matrix)

For 1 Gaussian:
- Blues (21 samples) : Predicted as Blues: 20, Classical: 1
- Reggae (20 samples) : Predicted as Blues: 4, Reggae: 12, Classical: 4
- Classical (21 samples) : Predicted as Blues: 1, Reggae: 0, Classical: 20

For 2 Gaussians:
- Blues (21 samples) : Predicted as Blues: 16, Reggae: 1, Classical: 4
- Reggae (20 samples) : Predicted as Blues: 2, Reggae: 16, Classical: 2
- Classical (21 samples) : Predicted as Blues: 0, Reggae: 0, Classical: 21

For 4 Gaussians:
- Blues (21 samples) : Predicted as Blues: 18, Reggae: 1, Classical: 2
- Reggae (20 samples) : Predicted as Blues: 2, Reggae: 15, Classical: 3
- Classical (21 samples) : Predicted as Blues: 1, Reggae: 0, Classical: 20

For 8 Gaussians:
- Blues (21 samples) : Predicted as Blues: 15, Reggae: 1, Classical: 5
- Reggae (20 samples) : Predicted as Blues: 3, Reggae: 14, Classical: 3
- Classical (21 samples) : Predicted as Blues: 1, Reggae: 0, Classical: 20

For 16 Gaussians:
- Blues (21 samples) : Predicted as Blues: 14, Reggae: 0, Classical: 7
- Reggae (20 samples) : Predicted as Blues: 4, Reggae: 12, Classical: 4
- Classical (21 samples) : Predicted as Blues: 1, Reggae: 0, Classical: 20

For 32 Gaussians:
- Blues (21 samples) : Predicted as Blues: 16, Reggae: 0, Classical: 5
- Reggae (20 samples) : Predicted as Blues: 4, Reggae: 12, Classical: 4
- Classical (21 samples) : Predicted as Blues: 1, Reggae: 0, Classical: 20

Experiment with Covariance type: Diagonal and different Gaussians

Changing the covariance type to **diagonal** means that each Gaussian component assumes the features are independent, so only the variances are considered, and the correlations between features are ignored.

- **Maximum Iterations:** 150
- **Covariance Type:** Diagonal (`'diag'`)
- **Tolerance:** $1e^{-8}$ (Convergence threshold)
- **Number of Gaussians:** Exponentially increasing values: [1, 2, 4, 8, 16, 32]

Table 2: GMM Classification Performance with Diagonal Covariance

# Gaussians	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
1	72.58	73.85	72.58	72.85
2	80.65	81.10	80.65	80.48
4	77.42	82.34	77.42	75.74
8	88.71	90.16	88.71	88.28
16	88.71	90.16	88.71	88.28
32	90.32	91.28	90.32	90.29

Prediction Breakdown for Each Class (A written Confussion Matrix)

Below is the confusion breakdown for different numbers of Gaussians with diagonal covariance :

For 1 Gaussian:
- Blues (21 samples) : Blues: 16, Reggae: 2, Classical: 3
- Reggae (20 samples) : Blues: 2, Reggae: 15, Classical: 3
- Classical (21 samples) : Blues: 7, Reggae: 0, Classical: 14

For 2 Gaussians:
- Blues (21 samples) : Blues: 17, Reggae: 3, Classical: 1
- Reggae (20 samples) : Blues: 1, Reggae: 14, Classical: 5
- Classical (21 samples) : Blues: 2, Reggae: 0, Classical: 19

For 4 Gaussians:
- Blues (21 samples) : Blues: 20, Reggae: 0, Classical: 1
- Reggae (20 samples) : Blues: 3, Reggae: 9, Classical: 8
- Classical (21 samples) : Blues: 2, Reggae: 0, Classical: 19

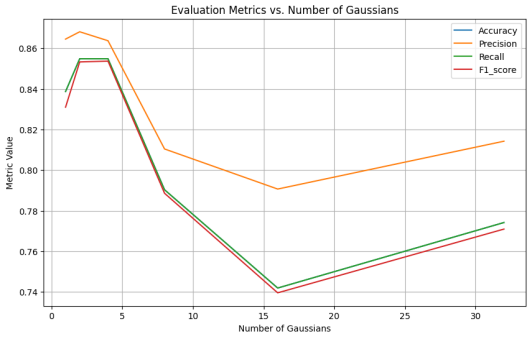
For 8 Gaussians:
- Blues (21 samples) : Blues: 20, Reggae: 0, Classical: 1
- Reggae (20 samples) : Blues: 3, Reggae: 14, Classical: 3
- Classical (21 samples) : Blues: 0, Reggae: 0, Classical: 21

For 16 Gaussians:
- Blues (21 samples) : Blues: 20, Reggae: 0, Classical: 1
- Reggae (20 samples) : Blues: 3, Reggae: 14, Classical: 3
- Classical (21 samples) : Blues: 0, Reggae: 0, Classical: 21

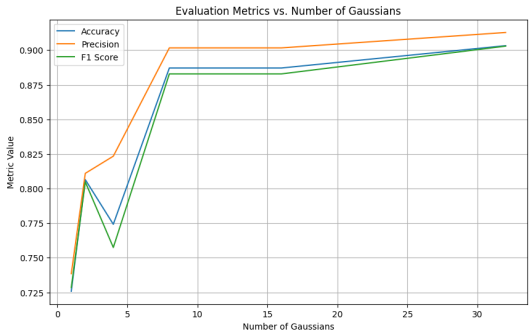
For 32 Gaussians:
- Blues (21 samples) : Blues: 20, Reggae: 0, Classical: 1
- Reggae (20 samples) : Blues: 3, Reggae: 16, Classical: 1

- Classical (21 samples) : Blues: 1, Reggae: 0, Classical: 20

Comparison of the Experiments



(a) Evaluation Metrics vs. Number of Gaussians (Full Covariance)



(b) Evaluation Metrics vs. Number of Gaussians (Diagonal Covariance)

Figure 1: Comparison of Evaluation Metrics for Full and Diagonal Covariance

Observations

Full Covariance: As the number of Gaussians increases, the evaluation metrics decrease, suggesting that the model starts to overfit(after 16 gaussians). The optimal performance is observed around 4 Gaussians, after which the model's ability to generalize declines. This indicates that while full covariance is beneficial for smaller models, its complexity causes a drop in performance when too many Gaussians are used.

Diagonal Covariance: In contrast, as the number of Gaussians increases, the evaluation metrics steadily improve, reaching the highest performance at 32 Gaussians, the maximum tested. This suggests that increasing the number of components benefits the model without causing overfitting.

Why is There an Opposite Behavior Between Full and Diagonal Covariance?

The key difference lies in how the two covariance types model feature relationships.

Full covariance attempts to model the dependencies between features, meaning it has significantly more parameters to estimate. As the number of Gaussians increases, this added complexity makes the model sensitive to overfitting, capturing noise instead of patterns. Consequently, generalization performance declines as the model becomes too specialized in the training data.

On the other hand, **diagonal covariance** assumes that features are independent, which reduces the number of parameters. This simplification makes training more stable and allows the model to handle more Gaussians without overfitting. The lower complexity helps the model generalize better, leading to improved performance as the number of Gaussians increases.

Specification: Dependencies between features describe how one feature relates to another within a dataset. More specifically, in music, tempo may correlate with energy, where faster tempos often indicate higher energy, and instrumentation may indicate genre. Full covariance captures these relationships, making the model more flexible but complex, while diagonal covariance assumes independence, simplifying training and reducing overfitting risk.

Conclusion

This project explored music genre classification using Gaussian Mixture Models (GMMs) with different covariance structures and varying numbers of Gaussian components. The results highlight a key trade-off between model complexity and generalization. Full covariance performs well when using a small number of Gaussians, with optimal results around 4 Gaussians, but as the number increases, overfitting occurs. In contrast, diagonal covariance, which assumes feature independence, remains stable and continues improving as more Gaussians are added, achieving its best performance at 32 Gaussians. This indicates that while full covariance is suitable for smaller models, its complexity limits scalability, whereas diagonal covariance provides a more efficient approach for larger models. These findings emphasize the importance of selecting an appropriate covariance type based on the number of Gaussians, balancing flexibility and generalization to optimize classification performance.