GMM_Music_Classification

March 3, 2022

1 GMM for Music and Speech Audio Classification

Uses Gaussian Mixture Models and EM Algorithm to classify music and speech audio files.

Libraries used: 1) Numpy - for numerical computations such as fft(), dot operator 2) Scipy - to read the .wav file and find the likelihood of data points 3) Matplotlib - to plot the spectogram

[1]:
```
import numpy as np

from scipy.io import wavfile
from scipy import signal
from scipy.stats import multivariate_normal

import matplotlib.pyplot as plt
import os
import random
```

1.0.1 Question 3) a)

1.0.2 Function read_audio()

Reads the .wav file and returns the sample_rate and the wav_file.

[2]:
```
def read_audio(folder, input):
    sample_rate, wav_file = wavfile.read('speech_music_classification/
↪'+folder+input)

    #print("Sample rate", sample_rate)

    length = len(wav_file)

    #print(length, wav_file)

    time_frame = length / sample_rate

    #To plot the audio wave
    #time = [i/sample_rate for i in range(len(wav_file))]
    #plt.plot(time, wav_file)
    #plt.show()
```

```
    return sample_rate, wav_file
```

1.0.3 Function fft()

Window size = 25 ms = 25 / 1000 * 16000 = 400 samples

Shift = 10 ms = 10 / 1000 * 16000 = 160 samples

For each window, computes 64 point magnitude FFT and retains the first 32 dimensions in each window, apply log of the magnitude of the FFT.

Returns the spectogram of dimension 32 x 2998.

[3]:
```
def fft(sample_rate, wav_file):
    length = len(wav_file)

    start = 0
    window_size = 25 * sample_rate // 1000

    shift = 10 * sample_rate // 1000

    #print("Windows size", window_size)
    #print("Shift", shift)

    i = 0
    while start + window_size <= length:
        fft = np.abs(np.fft.fft(wav_file[start:start+window_size], axis= 0 ,␣
↪n=64)[:32])
        with np.errstate(divide='ignore'):
            fft = np.log(fft)
        fft[np.isneginf(fft)]=0

        if start == 0:
            spectogram = fft
        else:
            spectogram = np.vstack((spectogram, fft))

        start += shift
        i+=1

    return spectogram
```

1.0.4 Function get_input_vector()

Reads the audio files in train dataset and performs fft on it to get the respective spectogram.

Returns the spectogram of all audio files in the given folder as a combined input vector.

[4]:
```python
def get_input_vector(path, folder_name):
    input_feature = np.zeros((0, 32))

    for dirname, _, filenames in os.walk(path):
        for filename in filenames:
            sample_rate, audio_wav = read_audio(folder_name, filename)
            audio_spec = fft(sample_rate, audio_wav)

            #plot_spectogram(audio_spec)

            input_feature = np.vstack((input_feature, audio_spec))
    return input_feature
```

Gets the input vector for the music files in train folder

[5]:
```python
music_feature = get_input_vector("speech_music_classification/train/music",␣
 ↪'train/music/')

print("Shape of Music Audio input ", music_feature.shape)
```

```
Shape of Music Audio input  (119920, 32)
```

Gets the input vector for the speech files in train folder

[6]:
```python
speech_feature = get_input_vector("speech_music_classification/train/speech",␣
 ↪'train/speech/')

print("Shape of Speech Audio input ", speech_feature.shape)
```

```
Shape of Speech Audio input  (119920, 32)
```

1.0.5 Class KMeans

Attributes

1) n_clusters : Number of clusters
2) centroid: Centroid of each cluster
3) clusters: Data points in each cluster
4) max_iter: Maximum iteration limit
5) tol: Error tolerance

Function fit(X) Performs KMeans clustering algorithm on the given input data.

Computes the centroid and data points present on each cluster.

[7]:
```python
class KMeans():
    def __init__(self, n_clusters, max_iter, tol = 0.001):
        self.n_clusters = n_clusters
        self.max_iter   = max_iter
        self.tol        = tol
```

```
    # X - shape (n_samples, n_features)
    def fit(self, X):

        self.centroid = [np.random.rand(X.shape[1], X.shape[1]) for i in␣
↪range(self.n_clusters)]

        data_points = random.sample(range(X.shape[0]), self.n_clusters)

        for i in range(self.n_clusters):
            self.centroid[i] = X[data_points[i]]

        epsilon = self.tol + 1
        for i in range(self.max_iter):

            if epsilon < self.tol:
                break

            self.clusters = [[] for i in range(self.n_clusters)]

            for sample in X:
                idx = np.argmin(np.array([np.linalg.norm(self.centroid[k] -␣
↪sample) for k in range(self.n_clusters)]))
                self.clusters[idx].append(sample)

            previous_centroid = self.centroid[:]

            epsilon = 0
            for j in range(self.n_clusters):

                if len(self.clusters[j]) == 0:
                    continue

                self.centroid[j] = np.mean(np.array(self.clusters[j]), axis = 0)

                err = abs(self.centroid[j] - previous_centroid[j])
                epsilon = max(epsilon, sum(err))
```

1.0.6 Class GMM

Attributes

1) n_mixtures: Number of Gaussian Mixture Components
2) covariance_type: Type of covariance matrix
3) clusters: Data points in each cluster
4) max_iter: Maximum iteration limit
5) tol: Error tolerance

Function initialize(mean, sigma) Initilizes the EM Iteration algorithm by setting mean and sigma to those obtained by KMean algorithm.

And \$α_l = 1 / M \$, where \$ l ∈ $1, \ldots, M$\$

Function fit(X) Performs the EM iteration algorithm on the given data to find the parameter for the GMM

Plots the log-likelihood for each EM iteration.

Function expectation_step(X) Performs the Expectation step in EM algorithm

Calculates the likelihood of the data points on each of the mixture components

Function maximization_step(X) Performs the Maximization step in EM algorithm

Updates alpha, mean and covariance based on the following equations:

\$ α_l^{new} = σ$_{i=1}^{N} P(l|x_i, \theta^n)_{\overline{N\$}}$

\$ μ_l^{new} = σ$_{i=1}^{N} x_i P(l|x_i, \theta^n)_{\overline{\sigma_{i=1}^{N} P(l|x_i,\theta^n)\$}}$

\$ σ_l^{new} = σ$_{i=1}^{N} P(l|x_i, \theta^n)(x_i - \mu_l^{new})(x_i - \mu_l^{new})^T{}_{\overline{\sigma_{i=1}^{N} P(l|x_i,\theta^n)\$}}$

Function predict_likelihood(X) Estimates the likelihood of the given data points based on the current parameters of GMM

Function calc_log_likelihood(X) Returns the log likelihood of the GMM on the current iteration

[8]:
```
class GMM:
    def __init__(self, n_mixtures, covariance_type, max_iter = 40, tol = 0.0001):
        self.n_mixtures = n_mixtures
        self.max_iter = max_iter
        self.tol = tol
        self.covariance_type = covariance_type

    def initialize(self, mean, sigma):
        self.alpha = [(1/self.n_mixtures) for i in range(self.n_mixtures)]
        self.mean  = mean
        self.sigma = sigma

    def expectation_step(self, X):
        self.likelihood_data = self.predict_likelihood(X)

    def maximization_step(self, X):
        self.alpha = np.mean(self.predict_likelihood(X), axis=0)

        for i in range(self.n_mixtures):
            like_prob = self.likelihood_data[:, i]
```

```
            num = X * like_prob.reshape((X.shape[0], 1))
            self.mean[i]  = np.sum(num, axis = 0) / np.sum(like_prob)

            cov = like_prob.reshape((1, X.shape[0])) * (X - self.mean[i]).T
            cov = cov.dot((X - self.mean[i]))
            cov /= np.sum(like_prob)
            self.sigma[i] = cov

            if self.covariance_type == 'diag':
                self.sigma = transform_to_diagonal_matrix(self.sigma)

    def predict_likelihood(self, X):
        likelihood = np.zeros( (X.shape[0], self.n_mixtures))

        for i in range(self.n_mixtures):
            distribution = multivariate_normal(mean=self.mean[i], cov=self.
↪sigma[i])
            likelihood[:,i] = distribution.pdf(X)

        likelihood_gmm = likelihood * self.alpha
        total_likelihood = likelihood_gmm.sum(axis=1)[:, np.newaxis]
        likelihood_gmm = likelihood_gmm / total_likelihood
        return likelihood_gmm

    def calc_log_likelihood(self, X):
        likelihood = np.zeros( (X.shape[0], self.n_mixtures) )
        for i in range(self.n_mixtures):
            distribution = multivariate_normal(mean=self.mean[i], cov=self.
↪sigma[i])
            likelihood[:,i] = distribution.pdf(X)

        numerator = likelihood * self.alpha
        log_like = np.log(np.sum(numerator, axis=1))
        log_like = np.sum(log_like)
        return log_like

    # X - n_samples, n_features
    def fit(self, X):

        log_likelihood = []
        last = self.max_iter

        for i in range(self.max_iter):
            self.expectation_step(X)
            self.maximization_step(X)

            log_likelihood.append(self.calc_log_likelihood(X))
```

```
            if i > 2 and abs(abs(log_likelihood[-1]) - abs(log_likelihood[-2]))␣
↪< self.tol:
                last = i+1
                break

        plt.scatter([i for i in range(last)], log_likelihood, marker="o",␣
↪color='g', linewidths=5)
        plt.xlabel('Number of Iteration')
        plt.ylabel('Log Likelihood')
        plt.title('Log Likelihood for each EM iteration')
        plt.show()
```

1.0.7 Question 3) b) i)

Number of Mixtures = 2

Covariance Matrix Type = Diagonal

[9]:
```
n_mixtures_ = 2
covariance_type = 'diag'
```

Performing KMeans clustering on Train Music feature data with 2 clusters

[10]:
```
k = KMeans(n_clusters = n_mixtures_, max_iter = 10)

k.fit(music_feature)
```

1.0.8 Function transform_to_diagonal_matrix(matrix)

Returns a matrix with diagonal elements of the given matrix

[11]:
```
def transform_to_diagonal_matrix(matrix):
    for i in range(len(matrix)):
        diag = np.einsum('ii->i', matrix[i])
        save = diag.copy()
        matrix[i][...] = 0
        matrix[i] = np.diag(save)
    return matrix
```

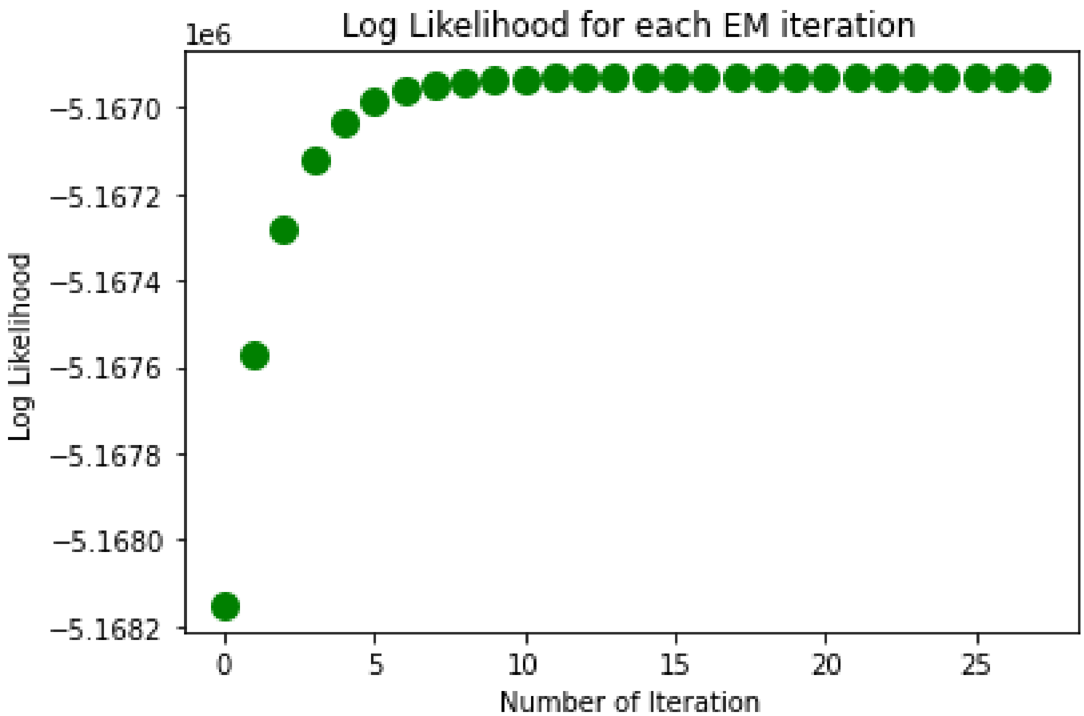
Calculates mean and covariance from the KMeans parameters

[12]:
```
mu_k_music = [k.centroid[0], k.centroid[1]]
sigma_k_music = [np.cov(np.array(k.clusters[i]).T) for i in range(n_mixtures_)]

if covariance_type == 'diag':
    transform_to_diagonal_matrix(sigma_k_music)
```

Finding the parameters of GMM for the given data

[13]:
```
gmm_music = GMM(n_mixtures = n_mixtures_, covariance_type=covariance_type)
```

```
gmm_music.initialize(mu_k_music[:], sigma_k_music[:])
gmm_music.fit(music_feature)
```



Performing KMeans clustering on Train Speech feature data with 2 clusters

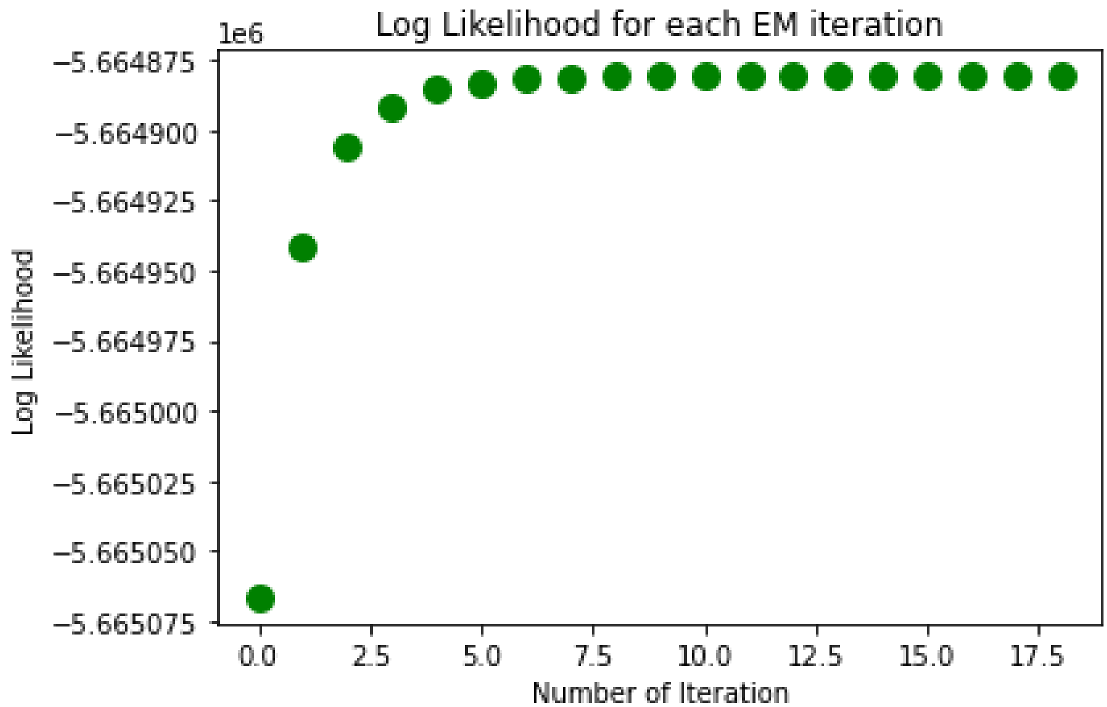
[14]:
```
k = KMeans(n_clusters = n_mixtures_, max_iter = 10)

k.fit(speech_feature)
```

Calculates mean and covariance from the KMeans parameters

[15]:
```
mu_k = [k.centroid[0], k.centroid[1]]
sigma_k = [np.cov(np.array(k.clusters[i]).T) for i in range(n_mixtures_)]

if covariance_type == 'diag':
    transform_to_diagonal_matrix(sigma_k)
```

Finding the parameters of GMM for the given data

[16]:
```
gmm_speech = GMM(n_mixtures = n_mixtures_, covariance_type=covariance_type)

gmm_speech.initialize(mu_k[:], sigma_k[:])
gmm_speech.fit(speech_feature)
```



1.0.9 Function misclassification_rate()

Returns the percentage of misclassification done by the built classifiers

[17]:
```
def misclassification_rate(gmm_speech, gmm_music):
    error_count = 0
    folders = ['speech_', 'music_']
    for j in range(2):
        for i in range(1, 24):
            feature = np.zeros((0, 32))
            sample_rate, clean_wav = read_audio('test/', folders[j]+str(i)+'.
↪wav')
            clean_spec = fft(sample_rate, clean_wav)

            feature = np.vstack((feature, clean_spec))

            likelihood_speech = gmm_speech.predict_likelihood(feature)
            alpha_speech = gmm_speech.alpha
            likelihood_gmm_speech = likelihood_speech * alpha_speech
            likelihood_gmm_speech = np.sum(likelihood_gmm_speech, axis=1)
            mean_likelihood_speech = np.mean(likelihood_gmm_speech)

            likelihood_music = gmm_music.predict_likelihood(feature)
            alpha_music = gmm_music.alpha
```

```
            likelihood_gmm_music = likelihood_music * alpha_music
            likelihood_gmm_music = np.sum(likelihood_gmm_music, axis=1)
            mean_likelihood_music = np.mean(likelihood_gmm_music)

            if j == 0 and mean_likelihood_speech < mean_likelihood_music:
                error_count +=1
            elif j == 1 and mean_likelihood_speech > mean_likelihood_music:
                error_count += 1

    print("Misclassification rate: ", error_count/48 * 100, "%")
```

Calculates Misclassification Rate for the given data

[18]:
```
misclassification_rate(gmm_speech, gmm_music)
```

```
Misclassification rate:  18.75 %
```

1.0.10 Question 3) b) ii)

Number of Mixtures = 2

Covariance Matrix Type = Full

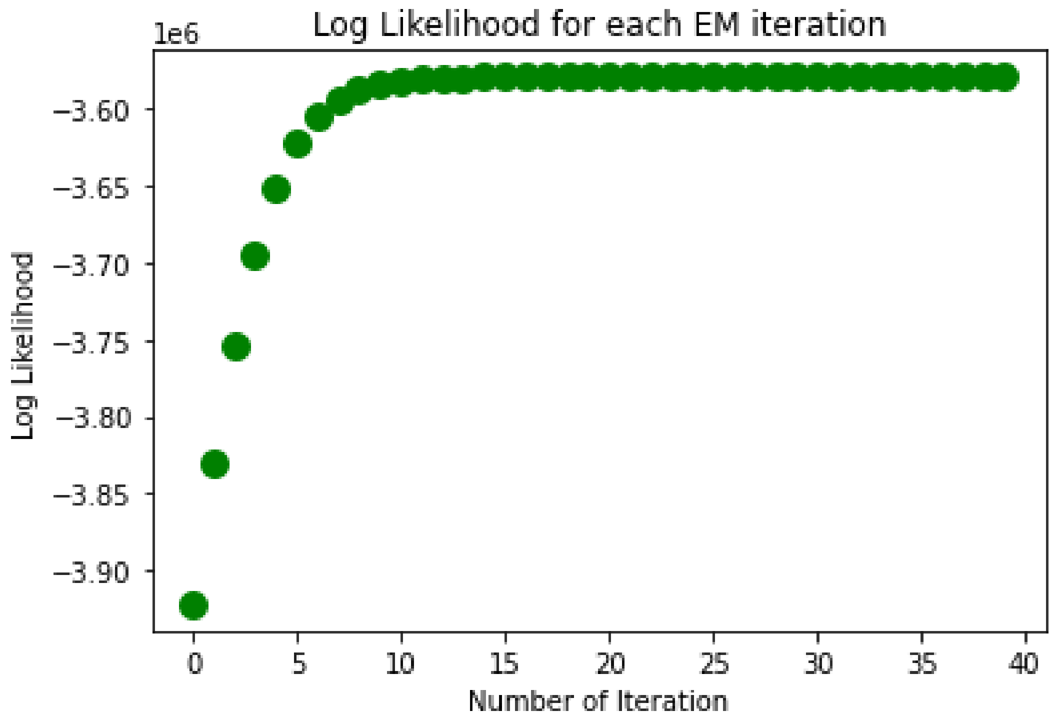
[19]:
```
n_mixtures_ = 2
covariance_type = 'full'
```

Calculates mean and covariance from the KMeans parameters

[20]:
```
mu_k_music = [k.centroid[0], k.centroid[1]]
sigma_k_music = [np.cov(np.array(k.clusters[i]).T) for i in range(n_mixtures_)]

if covariance_type == 'diag':
    transform_to_diagonal_matrix(sigma_k_music)
```

Finding the parameters of GMM for the given data

[21]:
```
gmm_music = GMM(n_mixtures = n_mixtures_, covariance_type=covariance_type)

gmm_music.initialize(mu_k_music[:], sigma_k_music[:])
gmm_music.fit(music_feature)
```

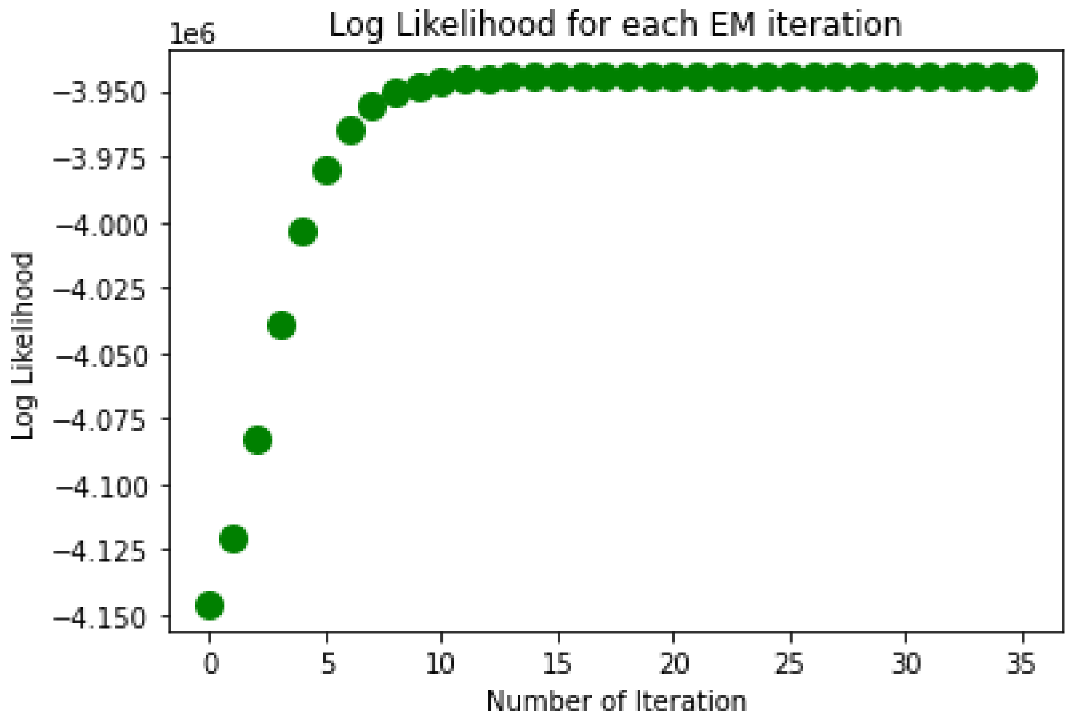


Calculates mean and covariance from the KMeans parameters

[22]:
```
mu_k = [k.centroid[0], k.centroid[1]]
sigma_k = [np.cov(np.array(k.clusters[i]).T) for i in range(n_mixtures_)]

if covariance_type == 'diag':
    transform_to_diagonal_matrix(sigma_k)
```

Finding the parameters of GMM for the given data

[23]:
```
gmm_speech = GMM(n_mixtures = n_mixtures_, covariance_type=covariance_type)

gmm_speech.initialize(mu_k[:], sigma_k[:])
gmm_speech.fit(speech_feature)
```



Calculates Misclassification Rate for the given data

[24]:
```
misclassification_rate(gmm_speech, gmm_music)
```

```
Misclassification rate:  60.41666666666664 %
```

1.0.11 Question 3) b) iii)

Number of Mixtures = 5

Covariance Matrix Type = Diagonal

[25]:
```
n_mixtures_ = 5
covariance_type = 'diag'
```

Performing KMeans clustering on Train Music feature data with 5 clusters

[26]:
```
k = KMeans(n_clusters = n_mixtures_, max_iter = 10)

k.fit(music_feature)
```

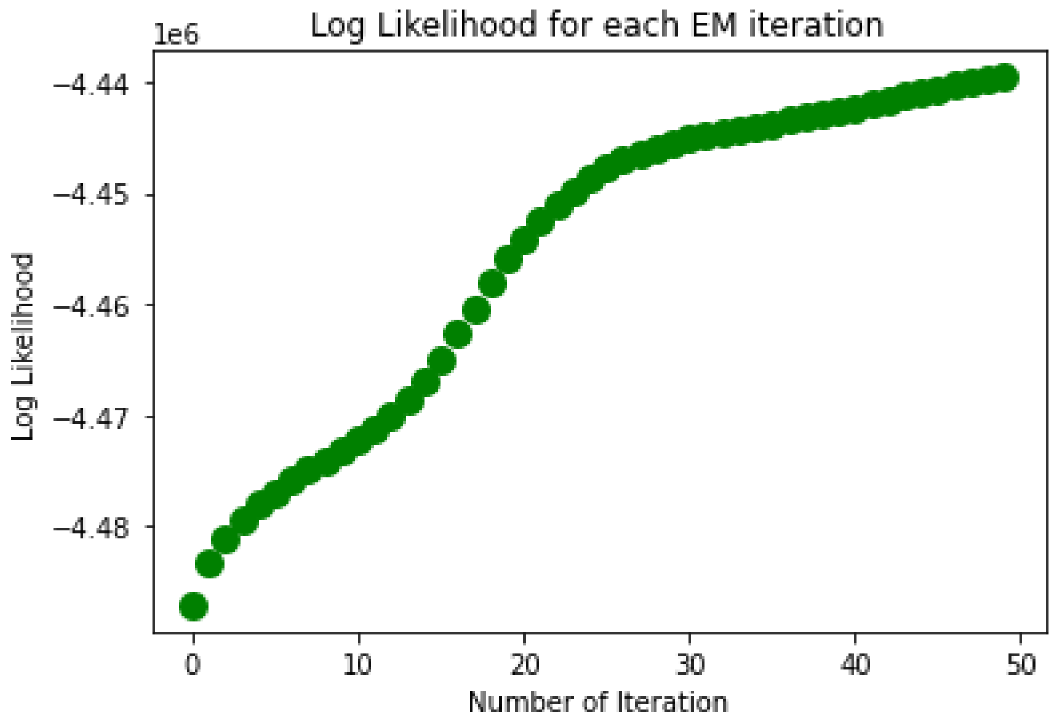
Calculates mean and covariance from the KMeans parameters

[27]:
```
mu_k_music = [k.centroid[i] for i in range(n_mixtures_)]
sigma_k_music = [np.cov(np.array(k.clusters[i]).T) for i in range(n_mixtures_)]

if covariance_type == 'diag':
```

```
    transform_to_diagonal_matrix(sigma_k_music)
```

Finding the parameters of GMM for the given data

[28]:
```
gmm_music = GMM(n_mixtures = n_mixtures_, max_iter = 50,␣
 ↪covariance_type=covariance_type)

gmm_music.initialize(mu_k_music[:], sigma_k_music[:])
gmm_music.fit(music_feature)
```



Performing KMeans clustering on Train Speech feature data with 5 clusters

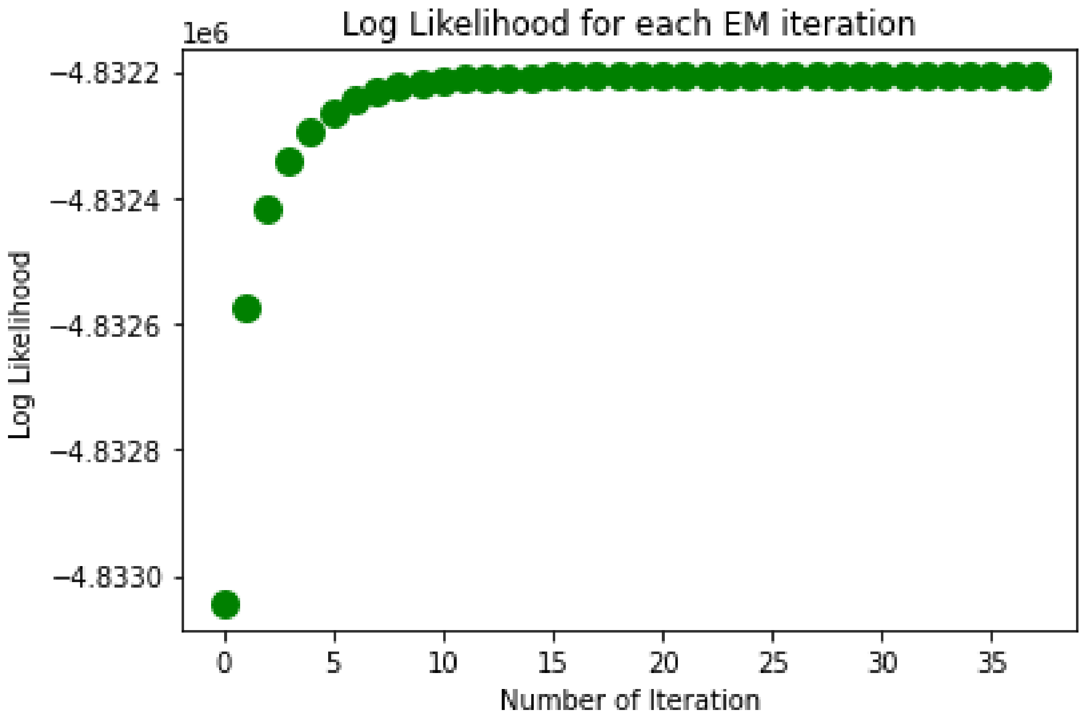
[29]:
```
k = KMeans(n_clusters = n_mixtures_, max_iter = 10)
k.fit(speech_feature)
```

Calculates mean and covariance from the KMeans parameters

[30]:
```
mu_k = [k.centroid[i] for i in range(n_mixtures_)]
sigma_k = [np.cov(np.array(k.clusters[i]).T) for i in range(n_mixtures_)]

if covariance_type == 'diag':
    transform_to_diagonal_matrix(sigma_k)
```

Finding the parameters of GMM for the given data

[31]:
```
gmm_speech = GMM(n_mixtures = n_mixtures_, covariance_type=covariance_type)

gmm_speech.initialize(mu_k[:], sigma_k[:])
gmm_speech.fit(speech_feature)
```



Calculates Misclassification Rate for the given data

[32]:
```
misclassification_rate(gmm_speech, gmm_music)
```

```
Misclassification rate:  16.666666666666664 %
```

1.0.12 Question 3) b) iii)

Number of Mixtures = 5

Covariance Matrix Type = Full

[33]:
```
n_mixtures_ = 5
covariance_type = 'full'
```

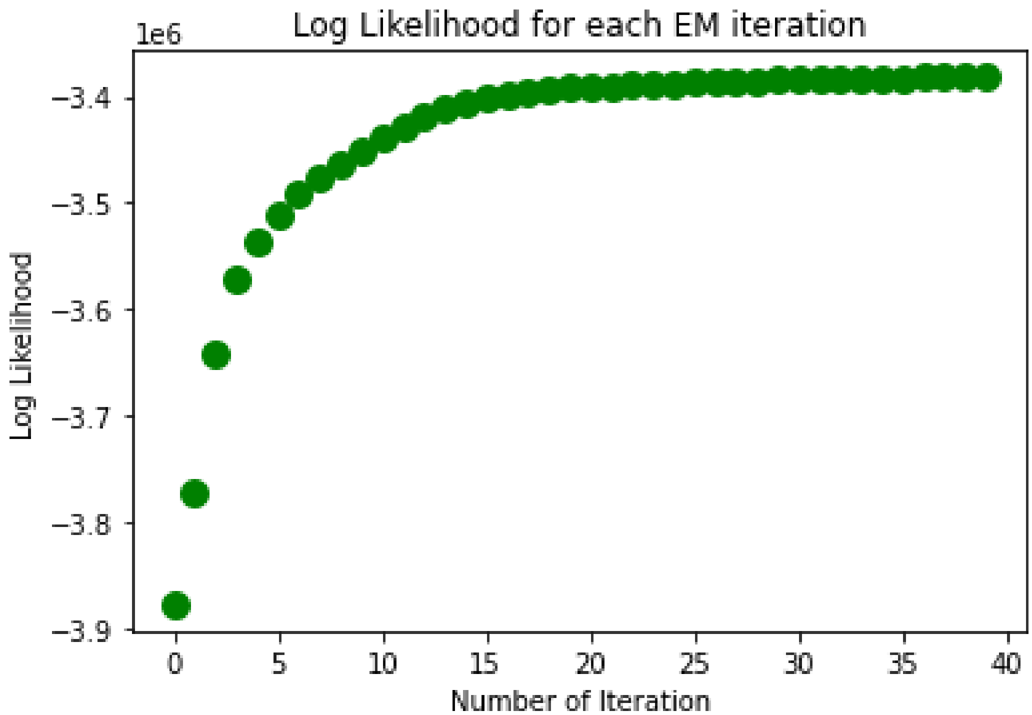
Calculates mean and covariance from the KMeans parameters

[34]:
```
mu_k_music = [k.centroid[i] for i in range(n_mixtures_)]
sigma_k_music = [np.cov(np.array(k.clusters[i]).T) for i in range(n_mixtures_)]

if covariance_type == 'diag':
```

```
    transform_to_diagonal_matrix(sigma_k_music)
```

Finding the parameters of GMM for the given data

[35]:
```
gmm_music = GMM(n_mixtures = n_mixtures_, covariance_type=covariance_type)

gmm_music.initialize(mu_k_music[:], sigma_k_music[:])
gmm_music.fit(music_feature)
```

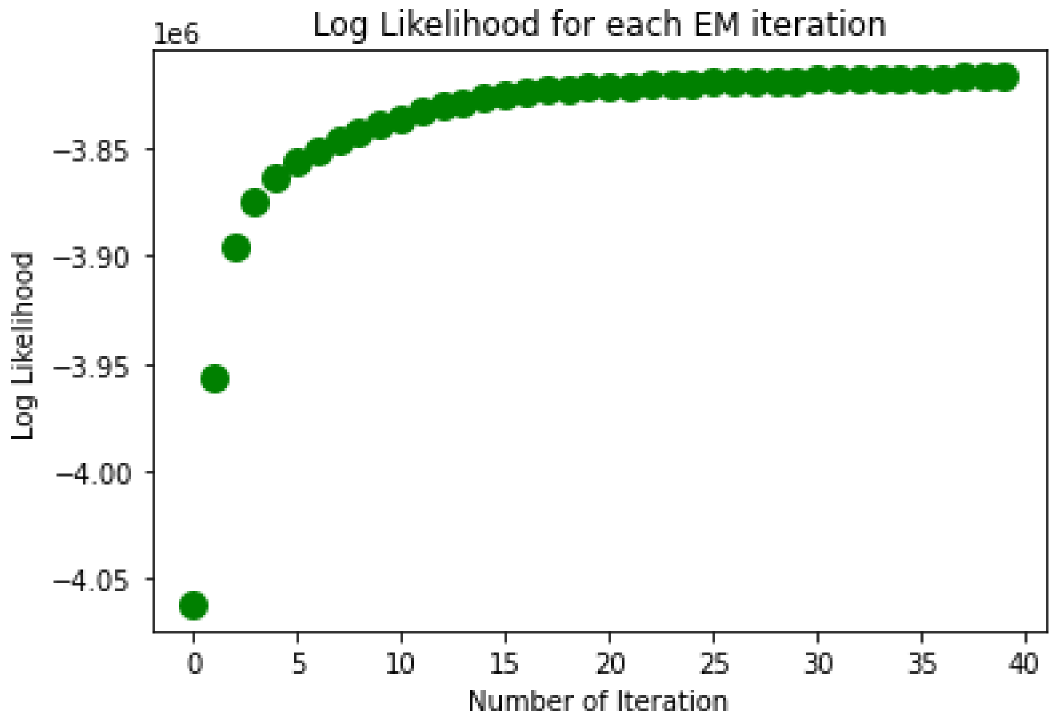


Calculates mean and covariance from the KMeans parameters

[36]:
```
mu_k = [k.centroid[i] for i in range(n_mixtures_)]
sigma_k = [np.cov(np.array(k.clusters[i]).T) for i in range(n_mixtures_)]

if covariance_type == 'diag':
    transform_to_diagonal_matrix(sigma_k)
```

Finding the parameters of GMM for the given data

[37]:
```
gmm_speech = GMM(n_mixtures = n_mixtures_, covariance_type=covariance_type)

gmm_speech.initialize(mu_k[:], sigma_k[:])
gmm_speech.fit(speech_feature)
```



Calculates Misclassification Rate for the given data

[38]:
```
misclassification_rate(gmm_speech, gmm_music)
```

```
Misclassification rate  41.666666666666667 %
```

1.0.13 Question 3) c)

Error Rate

Error Rate	2 Mixture Component GMM	5 Mixture Component GMM
Diagonal Covariance	18.75 %	16.66 %
Full Covariance	60.41 %	41.67 %

1.0.14 Question 3) d)

From the results obtained, it is clear that, 5 mixture component based GMM models perform better than their corresponding 2 mixture component based GMM models.

Conclusion: For the given problem, Increase in Number of Mixtures, Decrease in Error Rate

From the results obtained, it is clear that, diagonal covariance based GMM models perform better than their corresponding full coviariance based GMM models.

Conclusion: For the given problem, Diagonal covariance GMMs perform better than Full covariance GMMs