Implementing GMM

March 3, 2022

[1]:
```python
import numpy as np
from scipy.io import wavfile
import scipy.io
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
import  os
import math
import random
```

0.1 Computing Spectrograms of the audio files

Observations from analysis of audio files

- All the music files had a sampling frequency of 16K samples per second
- The total number of samples in each audio files was common for all (480000 samples)

Number of samples in 1s = 16K

Number of samples in 25ms = $16K * 0.025$ = 400 samples

The shift between successive windows = $10ms$

Number of samples to shift moving from one window to next = $16K * 0.010$ = 160 samples

Number of windows Without adding any amounts of padding at the end so the left end of the window should move only upto $480000 - 400 = 479600$

We will be shifting by 160 samples each time.

The total number of windows/frames = $\frac{479600}{160} = 2997.5 = (approx)2998$

[2]:
```python
class Spectrogram:
    def __init__(self,folder_path=None):
        self.name=folder_path
        self.num_frames = None
        self.data_matrix = None

    def plot_time_series_graph(self,wav_file):
        time = np.linspace(0,length,wav_file.shape[0])
        plt.plot(time, wav_file, label="channel")
        plt.xlabel("time (s)")
```

```
        plt.ylabel("Amplitude")
        plt.show()

    # Computing the number of frames in the spectrogram
    def compute_num_frames(self,audiofile):
        starting=0
        increment = 160
        window_size = 400
        self.num_frames = 0
        while(starting+window_size <= len(audiofile)):
            self.num_frames+=1
            starting += increment

    # Compute the spectrogram from the data
    def compute_spectrogram(self,audiofile):

        starting = 0
        increment = 160
        window_size = 400

        if(self.num_frames is None):
            self.compute_num_frames(audiofile)
        spectrogram = None
        while(starting+window_size < len(audiofile)):
            current_window = np.array(audiofile[starting : starting +␣
↪window_size])
            #Perform the required transformation
            fft_transform = np.log(np.abs(np.fft.fft(current_window,n=64)[:32].
↪reshape(32,1))+1e-10)
            if spectrogram is None:
                spectrogram = fft_transform
            else:
                spectrogram = np.append(spectrogram,fft_transform,axis = 0)
            starting += increment
        return spectrogram

    # Preparing the Data Matrix for training the size of dataset would be 32 *␣
↪(Total frames taken from all audio)
    def compute_data_matrix(self):

        for file in os.listdir(self.name):
            sample_rate, wav_file = wavfile.read(self.name+"/"+file)
            spectrogram = self.compute_spectrogram(wav_file).reshape((32,-1))
            if( self.data_matrix is None):
                self.data_matrix = spectrogram
            else:
                self.data_matrix = np.append(self.data_matrix,spectrogram,axis=1)
```

```
        #self.plot_spectrogram(spectrogram)
        #print(self.original_data_matrix.shape)


    def plot_spectrogram(self,spectrogram):
        time = np.linspace(0,self.num_frames,num=self.num_frames)
        freq = np.linspace(0,32,32)
        plt.pcolormesh(time,freq,spectrogram)
        plt.xlabel("Time")
        plt.ylabel("Frequency")
        plt.show()
```

0.2 KMeans Clustering for finding Initial Data centers

[3]:
```
# Helper class for KMeans clustering Algorithm
# Reference : https://pythonprogramming.net/
↪k-means-from-scratch-machine-learning-tutorial/

class KMeans:

    def __init__(self,k,data_matrix,max_iter=10, remove_non_diagonal=False):
        self.k = k
        self.n = data_matrix.shape[0]
        self.data_matrix = data_matrix
        self.max_iter = max_iter
        self.remove_non_diagonal = remove_non_diagonal
        self.centers = {}
        self.covs = {}
        self.labels = {}

    # Compute the cluster centers accoring to the KMeans Algorithm
    def compute_means(self):
        for i in range(self.k):
            self.centers[i] = self.data_matrix[i]

        for i in range(self.max_iter):
            for j in range(self.k):
                self.labels[j] = []
            for p in range(len(self.data_matrix)):
                current_point = self.data_matrix[p]

                # Get the label of the cluster to which the point belongs
                label = np.argmin(np.array([np.linalg.norm(current_point-self.
↪centers[c]) for c in range(self.k)]))
                self.labels[label].append(self.data_matrix[p])
```

```
            # Compute the new centers by taking the mean out of all points␣
↪assigned to a cluster
            new_centers = {}
            for j in range(self.k):
                mean_vector = np.mean(np.array(self.labels[j]),axis=0)
                new_centers[j] = mean_vector
            self.centers = new_centers

    # Compute the covariance of data points
    # present in each cluster
    # after finding the cluster centers

    def compute_covariance(self):
        for i in range(self.k):
            new_covs = np.cov(np.array(self.labels[i]).transpose())

            # Remove Non Diagonal entries if needed
            if(self.remove_non_diagonal):
                diagonal_elements =new_covs.diagonal()
                new_covariance_matrix = np.eye(new_covs[i].shape[0])
                for j in range(self.k):
                    new_covariance_matrix[j][j] = diagonal_elements[j]
                new_covs= new_covariance_matrix
            self.covs[i] = new_covs
```

1 Class for training the GMM Model

In a GMM Model we are interested in the probability

$P(X_i|\Theta) = \sum_{k=1}^{K} \alpha_k \mathbb{N}(X_i|\Theta_k)$

The parameters α and μ_k ,Σ_k is determined by using the EM Algorithm where in the E step we determine

$p(Z_i = l|X_i, \Theta_n) = \frac{\alpha_l \mathbb{N}(X_i|Z_i=l,\Theta_n)}{\sum_{k=1}^{K} \alpha_k \mathbb{N}(X_i|Z_i=l,\Theta_n)}$

where we assign soft values of memebership of data point to a cluster

The M step the parameters are updated such that

$α{new}^{l} = 1_{N∑{N}{n=1}P(Z_i=l|X_i,Θ_n)$

$\mu_l^{new} = \frac{\sum_N^{n=1} X_i P(Z_i=l|X_i,\Theta_n)}{\sum_N^{n=1} P(Z_i=l|X_i,\Theta_n)}$

$\mu_l^{new} = \frac{\sum_N^{n=1} P(Z_i=l|X_i,\Theta_n)(X_i-\mu_l^{new})(X_i-\mu_l^{new})^T}{\sum_N^{n=1} P(Z_i=l|X_i,\Theta_n)}$

Here the initial values of the mean and covariances are determined using KMeans Algorithm

alpha is taken to be 1/K where K is the number of mixture components

[4]:
```
class GMM:

    def __init__(self,K,data,skip_non_diagonal_entries=False,iterations=10):
        self.K = K
        self.means = None
        self.covs = None
        self.alpha = {}
        for i in range(self.K):
            self.alpha[i] = 1.0/K

        self.clip_nondiag_cov = skip_non_diagonal_entries
        self.data = data
        self.num_iterations = iterations
        self.likelihoods = None
        self.plot_points = []

    #Initialize the mean covariance matrices of the clusters
    def initialize(self,centers,cov_matrix):
        self.means = centers
        self.covs = cov_matrix

    # Computing the posterior probability of belonging to a cluster given a␣
↪datapoint
    # and model parameters
    def compute_posterior(self,X):

        probability = np.zeros((X.shape[0],self.K))

        # Denominator term
        for i in range(self.K):
            normal_fn = multivariate_normal(mean=self.means[i],cov=self.covs[i])
            for j in range(X.shape[0]):
                pdf_value = normal_fn.pdf(X[j])
                probability[j][i] = pdf_value* self.alpha[i]

        denominator = np.sum(probability,axis=1)

        # Computing the posterior probability
        for i in range(X.shape[0]):
            probability[i] = probability[i]/denominator[i]

        return probability

    # Computing the Log Likelihood
```

```python
    def compute_total_likelihood(self,X):
        probability = np.zeros((X.shape[0],self.K))
        for i in range(self.K):
            normal_fn = multivariate_normal(mean=self.means[i],cov=self.covs[i])
            for j in range(X.shape[0]):
                pdf_value = normal_fn.pdf(X[j])
                probability[j][i] = pdf_value* self.alpha[i]

        denominator = np.sum(probability,axis=1)
        for i in range(X.shape[0]):
            probability[i] = probability[i]/denominator[i]

        for i in range(X.shape[0]):
            for j in range(self.K):
                probability[i][j] = probability[i][j]*self.alpha[j]

        summation_on_mixture = np.log(np.sum(probability,axis=1))
        likelihood= np.sum(summation_on_mixture,axis=0)
        return likelihood

    # Expectation step in EM Algorithm
    def E_step(self):
        self.likelihoods = self.compute_posterior(self.data)

    # Update step in EM Algorithm
    def M_step(self):

        # Compute the new values of weights assigned for each gaussian
        new_alpha = {}
        mean_alpha = np.mean(self.likelihoods,axis=0)
        for i in range(self.K):
            new_alpha[i] = mean_alpha[i]


        denominator = np.sum(self.likelihoods,axis=0)

        # Compute the new mean vectors for the next iteration
        new_centers= {}
        # computing the terms in the numerator
        for i in range(self.K):
            logits = self.likelihoods[:,i]
            sum_vector = None
            for j in range(self.data.shape[0]):
                if sum_vector is None:
                    sum_vector = logits[j]*self.data[j]
                else:
                    sum_vector += logits[j]* self.data[j]
```

```
            new_centers[i]=sum_vector
        for i in range(self.K):
            new_centers[i] = new_centers[i]/denominator[i]


        # Compute the new covariance matrix for the next iteration

        new_covs = {}
        # computing the terms in the numerator
        for i in range(self.K):
            logits = self.likelihoods[:,i]
            sum_matrix = None
            for j in range(self.data.shape[0]):
                if sum_matrix is None:
                    vec = (self.data[j]-new_centers[i]).reshape((-1,1))
                    sum_matrix = logits[j]*(np.matmul(vec,np.transpose(vec)))
                else:
                    sum_matrix += logits[j]*(np.matmul(np.transpose(vec),vec))

            new_covs[i]=sum_matrix

        for i in range(self.K):
            new_covs[i] = new_covs[i]/denominator[i]
            # Stripping of Non Diagonal Entries
            if(self.clip_nondiag_cov):
                diagonal_elements =new_covs[i].diagonal()
                new_covariance_matrix = np.eye(new_covs[i].shape[0])
                for j in range(self.K):
                    new_covariance_matrix[j][j] = diagonal_elements[j]
                new_covs[i]= new_covariance_matrix

            if(np.linalg.matrix_rank(new_covs[i])!= 32):
                while(np.linalg.matrix_rank(new_covs[i])!= 32):
                    new_covs[i] = new_covs[i] + np.eye(32)*1

        # Update for next iteration
        self.alpha = new_alpha
        self.centers= new_centers
        self.covs = new_covs

    # The training Loop in the EM Algorithm
    def train(self):
        print("GMM Training")
        for iter in range(1,self.num_iterations+1):

            loglikelihoods_value = self.compute_total_likelihood(self.data)
```

```
            print("Iteration" + str(iter) +"  "+ str(loglikelihoods_value))
            self.plot_points.append(loglikelihoods_value)
            self.E_step()
            self.M_step()

    # Plotting the log likelihood as a function of EM Algorithm Iteration
    def likelihood_fn(self):
        plt.xlabel("Iterations count")
        plt.ylabel("Log Likelihood")
        for i in range(len(self.plot_points)):
            plt.scatter(i,self.plot_points[i],c='r')
        plt.show()
```

1.1 Preparing the Datasets for Training and Testing

[5]:
```
# Prepare the datasets
speech_train = Spectrogram("speech_music_classification/train/speech")
music_train = Spectrogram("speech_music_classification/train/music")

speech_train.compute_data_matrix()
music_train.compute_data_matrix()
```

[6]:
```
speech_test = Spectrogram("speech_music_classification/train/speech")
music_test = Spectrogram("speech_music_classification/train/music")
```

[7]:
```
# Helper function for checking accuracy
def accuracy_test(gmm_speech,gmm_music,dataset_type='speech'):

    directory = 'speech_music_classification/test'
    accuracy = 0
    total = 0
    spectrogram_class = Spectrogram()
    for file in os.listdir(directory):

        true_label = 0 if dataset_type in file else 1
        total+=1
        sample_rate, wav_file = wavfile.read(directory+"/"+file)
        spectrogram = spectrogram_class.compute_spectrogram(wav_file).
 ↪reshape((32,-1))

        # Compute probability
        prob1 = gmm_speech.compute_posterior(spectrogram.transpose())
        prob2 = gmm_music.compute_posterior(spectrogram.transpose())

        # Holds the probability of each frame being belonging to a class
        prob1_average = np.zeros((prob1.shape[0],1))
```

```
        prob2_average = np.zeros((prob2.shape[0],1))

        for i in range(prob2.shape[0]):
            prob_speech = 0
            prob_music = 0
            for j in range(gmm_speech.K):
                prob_speech = prob_speech + prob1[i][j]*gmm_speech.alpha[j]
                prob_music = prob_music + prob2[i][j]*gmm_music.alpha[j]
            prob1_average[i] = prob_speech
            prob2_average[i] = prob_music

        # Average over the probabilities of all the frames in audio file
        prob_speech = np.mean(prob1_average,axis=0)
        prob_music = np.mean(prob2_average,axis=0)

        # Assigning to the argmax the predicted label
        predicted_class = 0 if prob_speech >= prob_music else 1
        if(predicted_class == true_label):
            accuracy+=1

    print("Correctly classified = " + str(accuracy))
    print("Total files = " + str(total))
    accuracy = float(accuracy)/total
    print("Accuracy of the GMM Predictions on test data = " + str(accuracy*100))
```

[8]:
```
# Setting the parameters for Iteration count
kmeans_iteration_limit = 6
gmm_iteration_count = 15
```

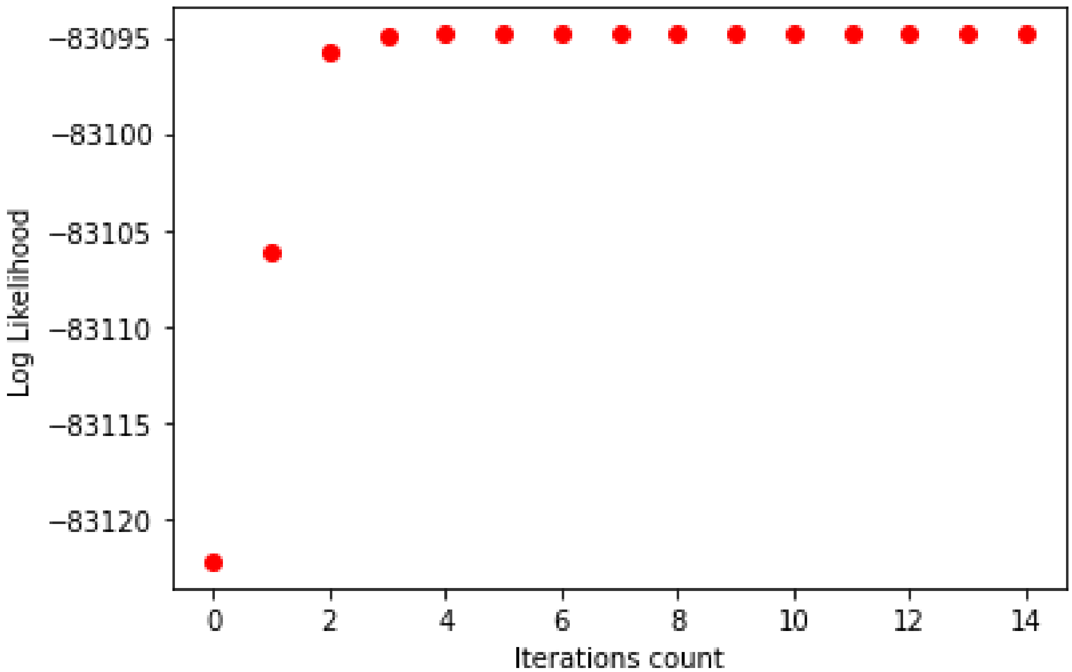
2 2 Mixture Gaussian Diagonal Covariance

[9]:
```
k_i_speech = KMeans(2,speech_train.data_matrix.
 ↪transpose(),kmeans_iteration_limit,remove_non_diagonal=True)
k_i_speech.compute_means()
k_i_speech.compute_covariance()
```

[10]:
```
gmm_i_speech = GMM(2,speech_train.data_matrix.
 ↪transpose(),True,gmm_iteration_count)
gmm_i_speech.initialize(k_i_speech.centers,k_i_speech.covs)
gmm_i_speech.train()
gmm_i_speech.likelihood_fn()
```

```
GMM Training
Iteration1  -83122.20989274865
Iteration2  -83106.11487290719
Iteration3  -83095.72079375514
```

```
Iteration4  -83094.90861610003
Iteration5  -83094.83429937912
Iteration6  -83094.82663973875
Iteration7  -83094.82583955045
Iteration8  -83094.82575583678
Iteration9  -83094.8257470775
Iteration10  -83094.82574616118
Iteration11  -83094.82574606537
Iteration12  -83094.82574605512
Iteration13  -83094.82574605382
Iteration14  -83094.82574605383
Iteration15  -83094.8257460538
```

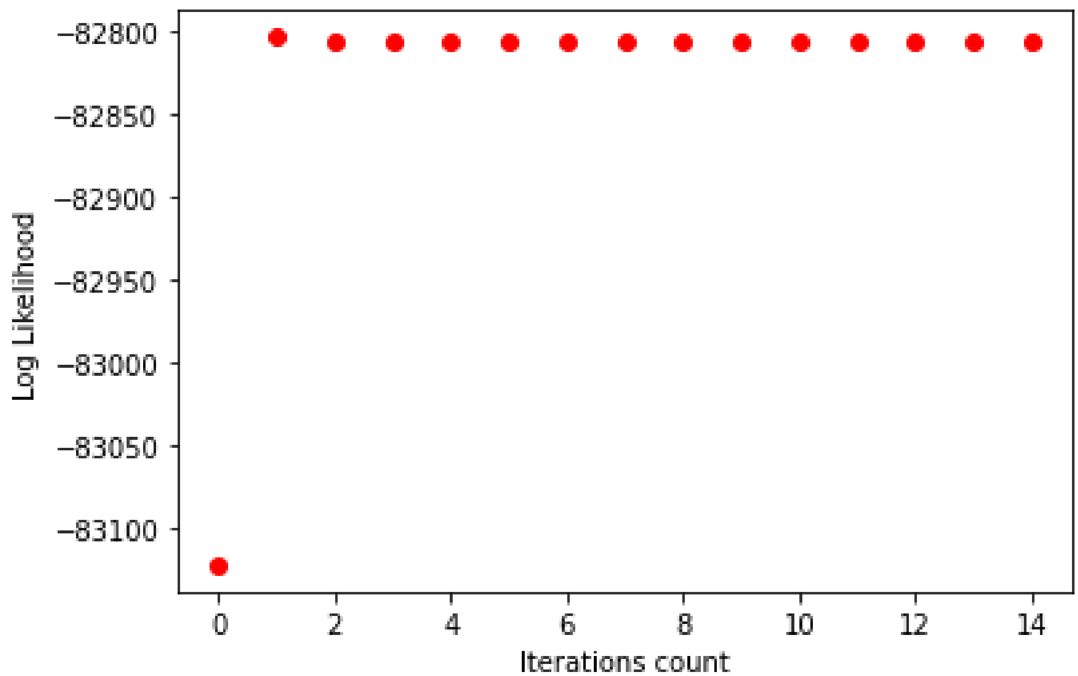


[11]:
```
k_i_music = KMeans(2,music_train.data_matrix.
 ↪transpose(),kmeans_iteration_limit,remove_non_diagonal=True)
k_i_music.compute_means()
k_i_music.compute_covariance()
```

[12]:
```
gmm_i_music = GMM(2,music_train.data_matrix.transpose(),True,gmm_iteration_count)
gmm_i_music.initialize(k_i_music.centers,k_i_music.covs)
gmm_i_music.train()
gmm_i_music.likelihood_fn()
```

```
GMM Training
Iteration1  -83122.20989274865
```

```
Iteration2  -82803.26901465219
Iteration3  -82805.32534540647
Iteration4  -82805.37531846325
Iteration5  -82805.38000960398
Iteration6  -82805.38045386915
Iteration7  -82805.38049593879
Iteration8  -82805.38049992235
Iteration9  -82805.38050029961
Iteration10  -82805.38050033535
Iteration11  -82805.38050033864
Iteration12  -82805.38050033872
Iteration13  -82805.38050033874
Iteration14  -82805.38050033877
Iteration15  -82805.3805003388
```



[13]:
```
accuracy_test(gmm_i_speech,gmm_i_music)
```

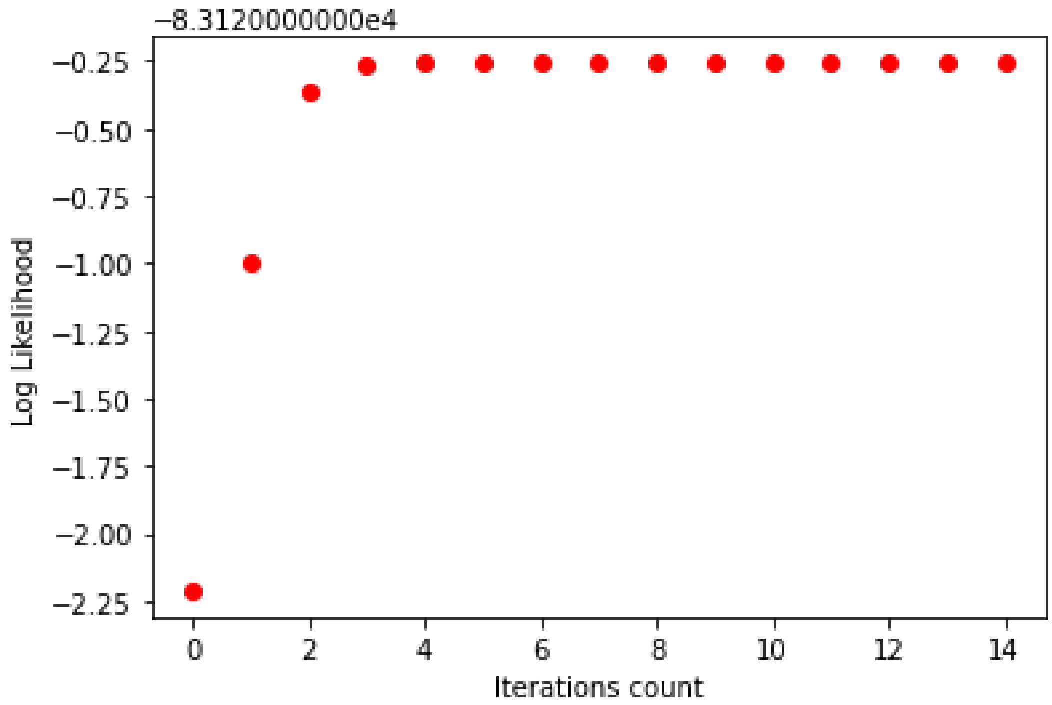
```
Correctly classified = 40
Total files = 48
Accuracy of the GMM Predictions on test data = 83.33333333333334
```

3 2 mixture Gaussian Full Covariance

[14]:
```
k_ii_speech = KMeans(2,speech_train.data_matrix.
 ↪transpose(),kmeans_iteration_limit,remove_non_diagonal=False)
k_ii_speech.compute_means()
k_ii_speech.compute_covariance()
```

[15]:
```
gmm_ii_speech = GMM(2,speech_train.data_matrix.
 ↪transpose(),False,gmm_iteration_count)
gmm_ii_speech.initialize(k_ii_speech.centers,k_ii_speech.covs)
gmm_ii_speech.train()
gmm_ii_speech.likelihood_fn()
```

```
GMM Training
Iteration1  -83122.20989274865
Iteration2  -83120.99862376199
Iteration3  -83120.36364427106
Iteration4  -83120.26652164238
Iteration5  -83120.25933612559
Iteration6  -83120.25869087211
Iteration7  -83120.25863161906
Iteration8  -83120.25862616638
Iteration9  -83120.2586256443
Iteration10  -83120.25862561827
Iteration11  -83120.25862561405
Iteration12  -83120.25862561371
Iteration13  -83120.25862561355
Iteration14  -83120.25862561351
Iteration15  -83120.25862561355
```

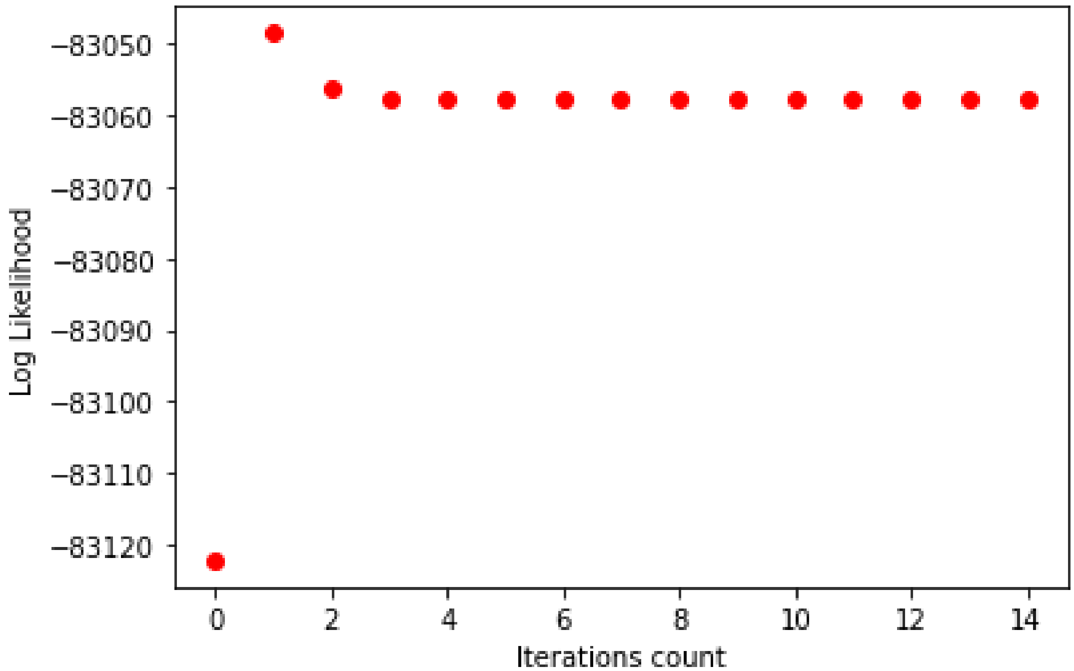


[16]:
```
k_ii_music = KMeans(2,music_train.data_matrix.
  ↪transpose(),kmeans_iteration_limit,remove_non_diagonal=False)
k_ii_music.compute_means()
k_ii_music.compute_covariance()
```

[17]:
```
gmm_ii_music = GMM(2,music_train.data_matrix.
  ↪transpose(),False,gmm_iteration_count)
gmm_ii_music.initialize(k_ii_music.centers,k_ii_music.covs)
gmm_ii_music.train()
gmm_ii_music.likelihood_fn()
```

```
GMM Training
Iteration1  -83122.20989274865
Iteration2  -83048.51079647487
Iteration3  -83056.33387379738
Iteration4  -83057.56959489791
Iteration5  -83057.7237151911
Iteration6  -83057.73445543497
Iteration7  -83057.73518287214
Iteration8  -83057.73523204833
Iteration9  -83057.73523537233
Iteration10  -83057.73523559688
Iteration11  -83057.73523561237
Iteration12  -83057.73523561332
```

```
Iteration13  -83057.73523561323
Iteration14  -83057.73523561348
Iteration15  -83057.73523561348
```



[18]:
```
accuracy_test(gmm_ii_speech,gmm_ii_music)
```

```
Correctly classified = 27
Total files = 48
Accuracy of the GMM Predictions on test data = 56.25
```

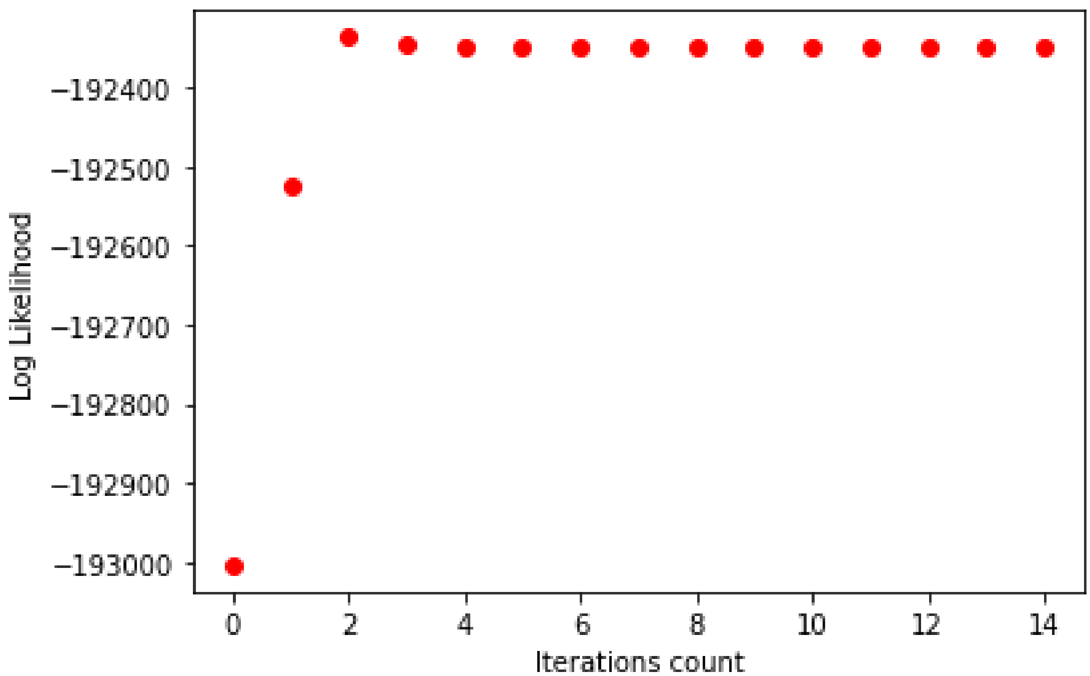
4 5 mixture Gaussian Diagonal Covariance

[19]:
```
k_iii_speech = KMeans(5,speech_train.data_matrix.
 ↪transpose(),kmeans_iteration_limit,remove_non_diagonal=True)
k_iii_speech.compute_means()
k_iii_speech.compute_covariance()
```

[20]:
```
gmm_iii_speech = GMM(5,speech_train.data_matrix.
 ↪transpose(),True,gmm_iteration_count)
gmm_iii_speech.initialize(k_iii_speech.centers,k_iii_speech.covs)
gmm_iii_speech.train()
gmm_iii_speech.likelihood_fn()
```

```
GMM Training
Iteration1  -193003.79445909726
```

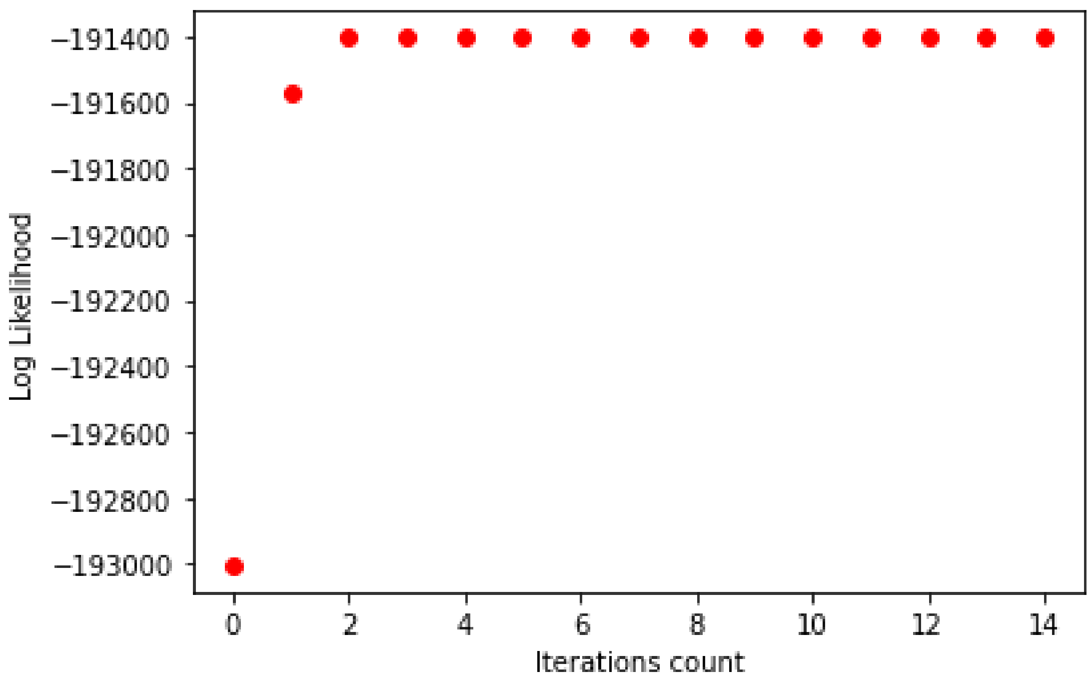
```
Iteration2  -192525.1244459133
Iteration3  -192336.18078816903
Iteration4  -192346.62654893086
Iteration5  -192349.47876707264
Iteration6  -192350.15011428678
Iteration7  -192350.30620427165
Iteration8  -192350.34274367883
Iteration9  -192350.3513586448
Iteration10  -192350.35339957464
Iteration11  -192350.353884443
Iteration12  -192350.35399981032
Iteration13  -192350.3540272816
Iteration14  -192350.35403382572
Iteration15  -192350.35403538527
```



[21]:
```
k_iii_music = KMeans(5,music_train.data_matrix.
 ↪transpose(),kmeans_iteration_limit,remove_non_diagonal=True)
k_iii_music.compute_means()
k_iii_music.compute_covariance()
```

[22]:
```
gmm_iii_music = GMM(5,music_train.data_matrix.
 ↪transpose(),True,gmm_iteration_count)
gmm_iii_music.initialize(k_iii_music.centers,k_iii_music.covs)
gmm_iii_music.train()
gmm_iii_music.likelihood_fn()
```

```
GMM Training
Iteration1  -193003.79445909726
Iteration2  -191570.78945734003
Iteration3  -191404.44853250636
Iteration4  -191403.55549843112
Iteration5  -191403.52462655108
Iteration6  -191403.5253766211
Iteration7  -191403.5258871991
Iteration8  -191403.52598721322
Iteration9  -191403.5260027139 4
Iteration10  -191403.52600489388
Iteration11  -191403.52600518553
Iteration12  -191403.52600522415
Iteration13  -191403.5260052294
Iteration14  -191403.52600522953
Iteration15  -191403.52600522962
```



[23]:
```
accuracy_test(gmm_iii_speech,gmm_iii_music)
```

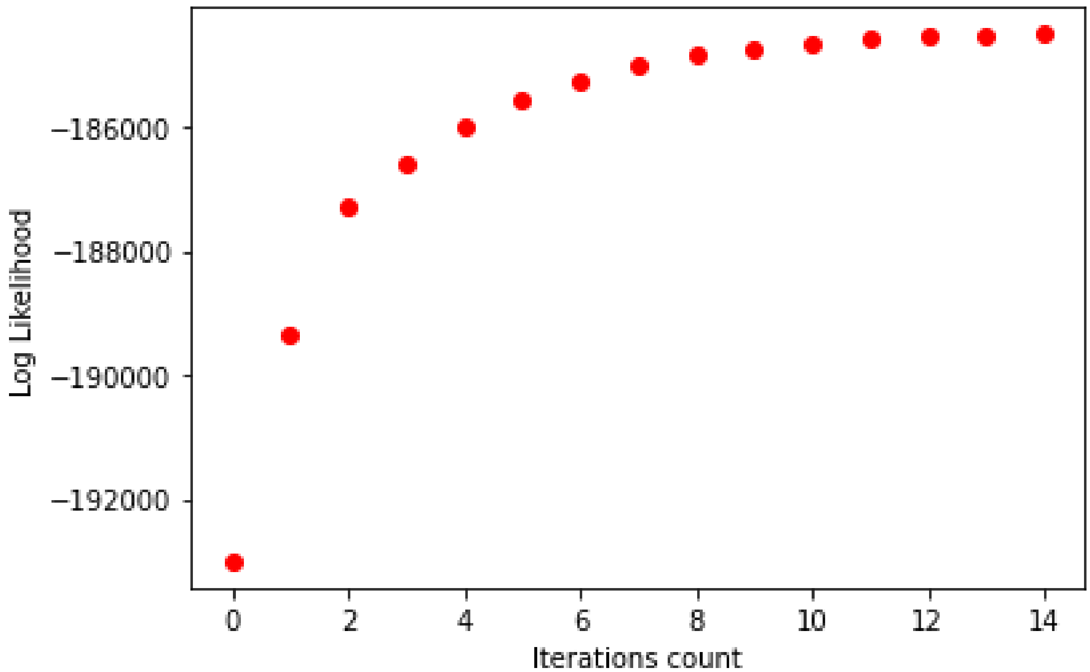
```
Correctly classified = 32
Total files = 48
Accuracy of the GMM Predictions on test data = 66.66666666666666
```

5 5 mixture Gaussian Full Covariance

[24]:
```
k_iv_speech = KMeans(5,speech_train.data_matrix.
  ↪transpose(),kmeans_iteration_limit,remove_non_diagonal=False)
k_iv_speech.compute_means()
k_iv_speech.compute_covariance()
```

[25]:
```
gmm_iv_speech = GMM(5,speech_train.data_matrix.
  ↪transpose(),False,gmm_iteration_count)
gmm_iv_speech.initialize(k_iv_speech.centers,k_iv_speech.covs)
gmm_iv_speech.train()
gmm_iv_speech.likelihood_fn()
```

```
GMM Training
Iteration1  -193003.79445909726
Iteration2  -189338.5541192052
Iteration3  -187285.63559388215
Iteration4  -186584.5634945222
Iteration5  -186013.42739394872
Iteration6  -185572.08442673727
Iteration7  -185246.28373146942
Iteration8  -185011.74958000946
Iteration9  -184845.27095740603
Iteration10  -184728.07022512288
Iteration11  -184645.977114116
Iteration12  -184588.65922643358
Iteration13  -184548.7229890435
Iteration14  -184520.93589449147
Iteration15  -184501.61997664248
```

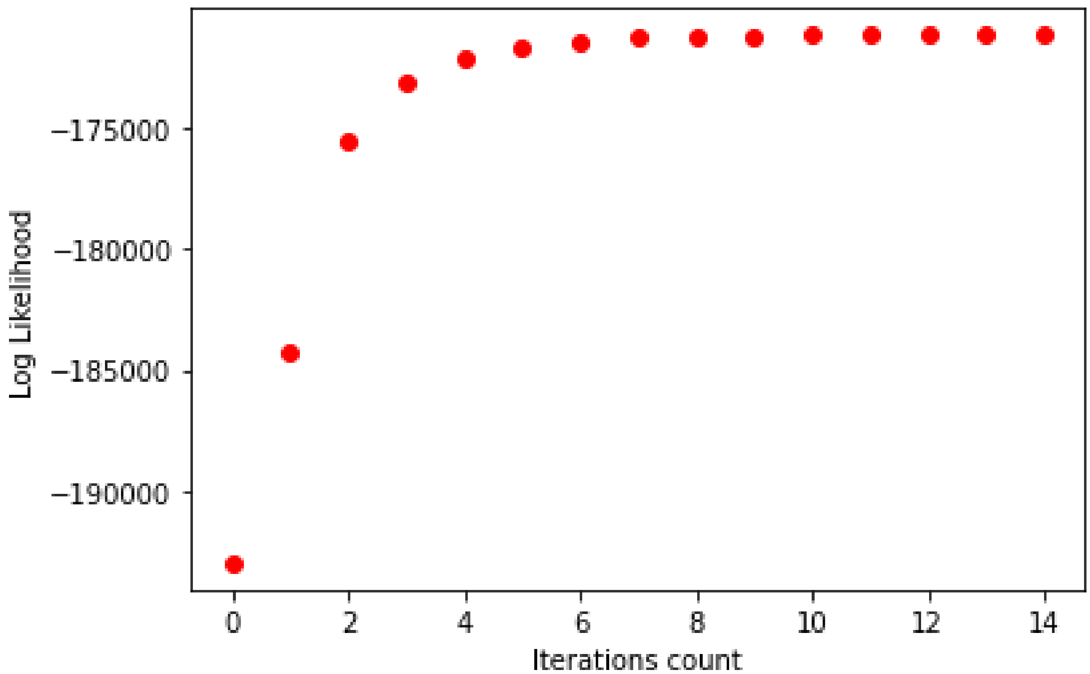


[26]:
```
k_iv_music = KMeans(5,music_train.data_matrix.
 ↪transpose(),kmeans_iteration_limit,remove_non_diagonal=False)
k_iv_music.compute_means()
k_iv_music.compute_covariance()
```

[27]:
```
gmm_iv_music = GMM(5,music_train.data_matrix.
 ↪transpose(),False,gmm_iteration_count)
gmm_iv_music.initialize(k_iv_music.centers,k_iv_music.covs)
gmm_iv_music.train()
gmm_iv_music.likelihood_fn()
```

```
GMM Training
Iteration1  -193003.79445909726
Iteration2  -184303.9517579474
Iteration3  -175568.0794464079
Iteration4  -173155.68166448345
Iteration5  -172127.8585550698
Iteration6  -171643.36049752237
Iteration7  -171402.86401053195
Iteration8  -171280.27003159418
Iteration9  -171216.9094585535
Iteration10  -171183.92621749875
Iteration11  -171166.69159167344
Iteration12  -171157.66826796025
Iteration13  -171152.9391509156
```

```
Iteration14  -171150.4592804641
Iteration15  -171149.15850776382
```



[28]:
```
accuracy_test(gmm_iv_speech,gmm_iv_music)
```

```
Correctly classified = 24
Total files = 48
Accuracy of the GMM Predictions on test data = 50.0
```

6 Summary

	Diagonal Covariance	*Full Covariance*
2 Mixture	*83.34*	*56.25*
5 Mixture	*66.66*	*50.0*

6.1 Conclusions

- *The error rate increased on increasing the number of gaussians*
- *Fixing the number of mixture components, the Diagonal Covariance mxiture models perform better than the full covariance models*

[ ]: