**Queen Mary University of London**

**Department of Electrical Engineering & Computer Science**

**ECS708P Machine Learning**

# Assignment 2 – Clustering and Mixture of Gaussians

**Hasan Emre Erdemoglu**

**3 December 2020**

**Table of Contents:**

## MoG Modelling Using the EM Algorithm

This report will implement MoG Modelling using Expectation Maximization algorithm on Peterson and Barney's dataset of vowel formant frequencies. For this assignment only the first two formants will be used in MoG modelling.

In the report each task will be handled separately. In the report the code will be given in full. Any additional code written or altered will be displayed in **bold** for clarification purposes.

### Task 1:

In this task the dataset will be loaded to the workspace and the phonemes will be displayed on a 2D plot with axes formed by fundamental frequencies F1 and F2. The following are the outputs given by *task_1.py*. Note that 3 clusters are already visible on figure on the left.

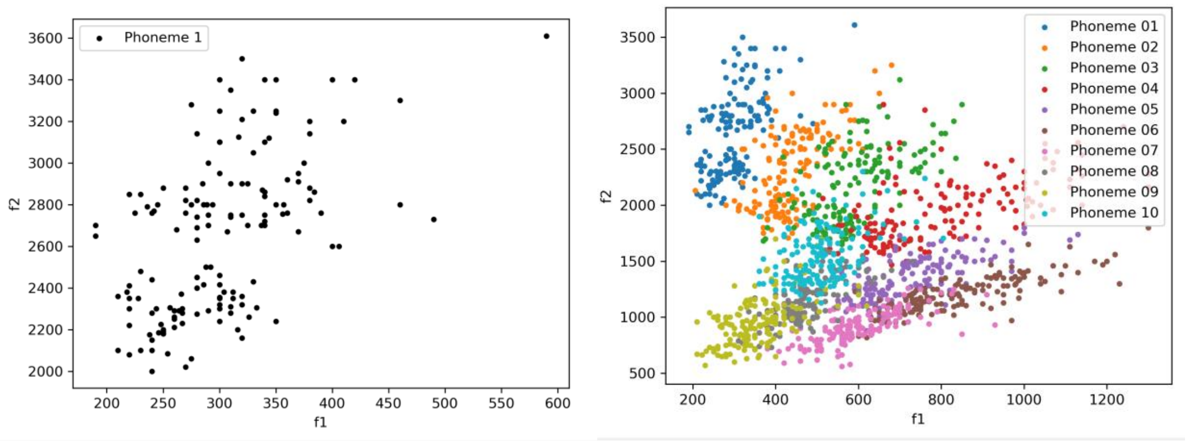


*Figure 1. 2D Plot of Phoneme 1 (Left) and All Phonemes (Right) using fundamental frequencies F1 and F2*

Additionally, the following statistic is also provided:

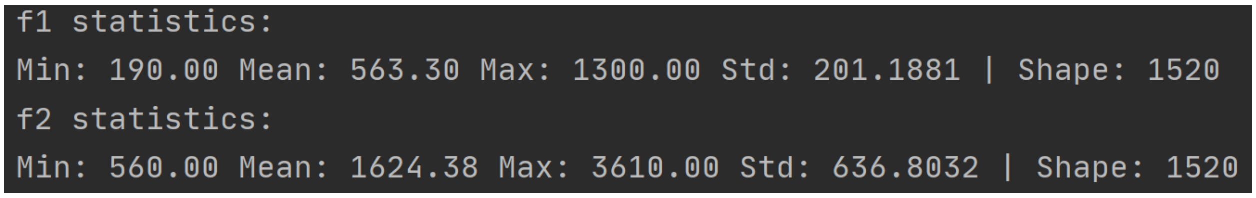

```
f1 statistics:
Min: 190.00 Mean: 563.30 Max: 1300.00 Std: 201.1881 | Shape: 1520
f2 statistics:
Min: 560.00 Mean: 1624.38 Max: 3610.00 Std: 636.8032 | Shape: 1520
```

*Figure 2. Task 1 fundamental frequency statistics for Phoneme 1*

## Code for Task 1:

```
from print_values import *
from plot_data_all_phonemes import *
from plot_data import *

# File that contains the data
data_npy_file = 'data/PB_data.npy'

# Loading data from .npy file
data = np.load(data_npy_file, allow_pickle=True)
data = np.ndarray.tolist(data)

# Make a folder to save the figures
figures_folder = os.path.join(os.getcwd(), 'figures')
if not os.path.exists(figures_folder):
    os.makedirs(figures_folder, exist_ok=True)

# Array that contains the phoneme ID (1-10) of each sample
phoneme_id = data['phoneme_id']

# frequencies f1 and f2
f1 = data['f1']
f2 = data['f2']
print('f1 statistics:')
print_values(f1)
print('f2 statistics:')
print_values(f2)

# Initialize array containing f1 & f2, of all phonemes.
X_full = np.zeros((len(f1), 2))
# ########################################
# Write your code here
# Store f1 in the first column of X_full, and f2 in the second column of X_full
X_full[:, 0] = f1
X_full[:, 1] = f2
```

```
# ########################################/
X_full = X_full.astype(np.float32)

# you can use the p_id variable, to store the ID of the chosen phoneme that will be used (e.g. phoneme 1, or phoneme 2)
p_id = 1

########################################
# Write your code here
# Create an array named "X_phoneme_1", containing only samples that belong to the chosen phoneme.
# The shape of X_phoneme_1 will be two-dimensional. Each row will represent a sample of the dataset,
# and each column will represent a feature (e.g. f1 or f2)
# Fill X_phoneme_1 with the samples of X_full that belong to the chosen phoneme
# To fill X_phoneme_1, you can leverage the phoneme_id array, that contains the ID of each sample of X_full

# Create array containing only samples that belong to phoneme 1
# X_phoneme_1 = np.zeros((np.sum(phoneme_id == p_id), 2))
X_phoneme_1 = X_full[phoneme_id == p_id, :]

########################################

# Plot array containing all phonemes

# Create a figure and a subplot
fig, ax1 = plt.subplots()
# plot the full dataset (f1 & f2, all phonemes)
plot_data_all_phonemes(X=X_full, phoneme_id=phoneme_id, ax=ax1)
# save the plotted dataset as a figure
plot_filename = os.path.join(os.getcwd(), 'figures', 'dataset_full.png')
plt.savefig(plot_filename)

################################################
# Plot array containing phoneme 1
# Create a figure and a subplot
_, ax2 = plt.subplots()
title_string = 'Phoneme 1'
```

```
# plot the samples of the dataset, belonging to phoneme 1 (f1 & f2, phoneme 1)

plot_data(X=X_phoneme_1, title_string=title_string, ax=ax2)

# save the plotted points of phoneme 1 as a figure

plot_filename = os.path.join(os.getcwd(), 'figures', 'dataset_phoneme_1.png')

plt.savefig(plot_filename)



# enter non-interactive mode of matplotlib, to keep figures open

plt.ioff()

plt.show()
```

**Task 2:**

This section will utilize *task2.py* given in the lab assignment. This script is used 4 times to construct MoG models for phonemes 1 and 2 for K=3 and K=6, respectively. Multiple repetitions of experiments were made to compare outputs.

The MoG model is trained using Expectation Maximization algorithm. The number of Gaussians fitted to the dataset is given by value K. Separate MoG models are trained for phonemes 1 and 2. The following algorithm shows Expectation Maximization used in MoG model. This will be explained in detail:

```
X = X_phoneme.copy()  # as dataset X, we will use only the samples of the chosen phoneme
N = X.shape[0]  # get number of samples
D = X.shape[1] # get dimensionality of our dataset
p = np.ones(k)/k  # common practice : GMM weights initially set as 1/k

# GMM means are picked randomly from data samples
random_indices = np.floor(N*np.random.rand(k))
random_indices = random_indices.astype(int)
mu = X[random_indices, :]  # shape kxD
s = np.zeros((k, D, D))  # shape kxDxD # covariance matrices
n_iter = 100 # number of iterations for the EM algorithm

# initialize covariances
for i in range(k):
    cov_matrix = np.cov(X.transpose())
    # initially set to fraction of data covariance
    s[i, :, :] = cov_matrix/k

# Initialize array Z that will get the predictions of each Gaussian on each sample
Z = np.zeros((N, k))  # shape Nxk

###############################
# run Expectation Maximization algorithm for n_iter iterations
for t in range(n_iter):
    # Do the E-step
    Z = get_predictions(mu, s, p, X)
    Z = normalize(Z, axis=1, norm='l1')
```

```
# Do the M-step:
for i in range(k):
    mu[i, :] = np.matmul(X.transpose(), Z[:, i]) / np.sum(Z[:, i])
    # We will fit Gaussian's with diagonal covariance matrices
    mu_i = mu[i, :]
    mu_i = np.expand_dims(mu_i, axis=1)
    mu_i_repeated = np.repeat(mu_i, N, axis=1)
    X_minus_mu = (X.transpose() - mu_i_repeated)**2
    res_1 = np.squeeze(np.matmul(X_minus_mu, np.expand_dims(Z[:, i], axis=1)))/np.sum(Z[:, i])
    s[i, :, :] = np.diag(res_1)
    p[i] = np.mean(Z[:, i])
```

The algorithm above first sets the weights and randomly picks GMM means from the dataset. Mean matrix has a shape k times D whereas covariance tensor has a shape of k by D time by D; where k is the number of Gaussians in the mixture model and D is the dimensionality of the dataset respectively. Covariance matrix is initialized for each Gaussian using the dataset X. GMM weights are initialized uniformly.

Since there are k Gaussians and N samples; a predictions matrix Z is formed with size N by k. After this setup, Expectation Maximization algorithm is applied for n_iter times.

For the expectation step; using the mean, covariance, GMM weights and the dataset; predictions are fetched. This is done by using *get_predictions.py* method. This method returns predictions Z for each Gaussian by the following formula. Normalization is required to make sure prediction vector Z holds up to the probability measure. Note that the equation below must be calculated for all samples:

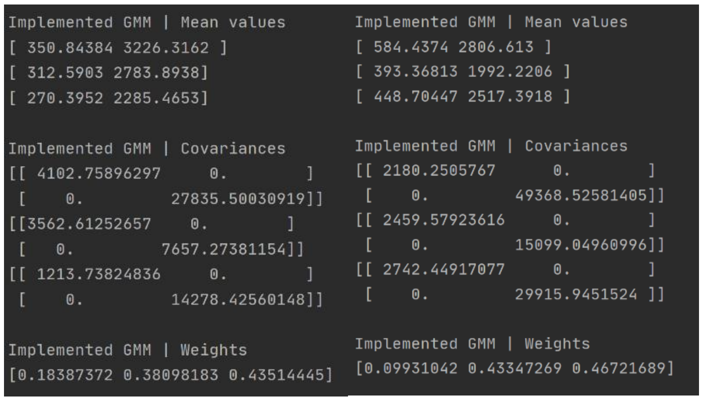
$$P_k(x_i \mid \theta) = \pi_k N(x_i|\mu_k, \Sigma_k) = \pi_k \frac{1}{(2\pi)^{D/2}\, \Sigma_k^{1/2}} e^{(-0.5(x-\mu_k)^T \Sigma_k^{-1}(x-\mu_k)}$$

In the maximization step, for each of the Gaussians in GMM model; weighted log-likelihood function is optimized for the mixture model: new means; covariances and mixture weights for the corresponding Gaussians are selected. Logarithm operation is used to simplify the exponential term in the Gaussian distribution, making the optimization easier. Taking the derivative with respective to weighted log-likelihood function for mean, standard deviation and GMM weights gives the equations in the M-step, written in the code above. The following are the update equations at (i+1)'th iteration:

$$\mu_k^{i+1} = \frac{\sum_n z_{n,k}\, x_n}{\sum_n z_{n,k}}, \Sigma_k^{i+1} = \frac{\sum_n z_{n,k} \left(x_n - \mu_k^{i+1}\right)\left(x_n - \mu_k^{i+1}\right)^T}{\sum_n z_{n,k}}, \pi_k^i = \frac{1}{n}\sum_n z_{n,k}$$

Note that in the maximization step; only diagonal entries are computed. For the covariance matrix, diagonal entries correspond to variances of individual dimensions. “res_1” estimates variances for F1 and F2 as a vector; which is later translated to covariance matrix; to be used in the GMM.

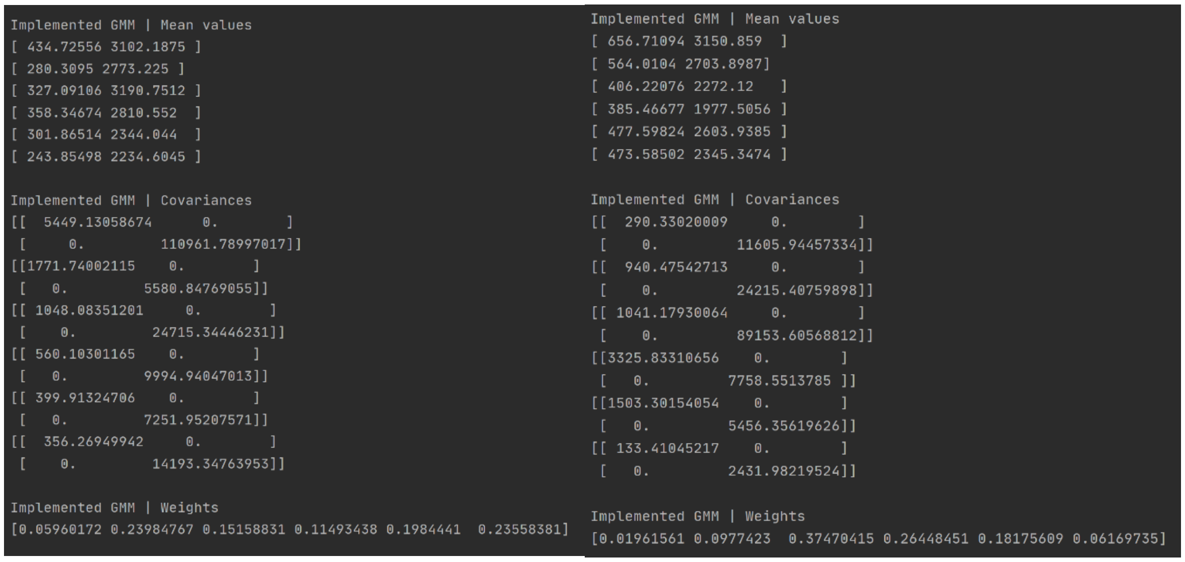
The outputs for phoneme 1 and 2 for K = 3 and K = 6 are given below. These values are also recorded within the code part of the assignment. All tests were run for 100 EM iterations. Note that the initial mean values are selected at random; so, when the same experiment restart, the Gaussian centers and label colors change. After 100 iterations, all independent experiments converge to similar layouts outputting very similar results.



```
Implemented GMM | Mean values
[ 350.84384 3226.3162 ]
[ 312.5903 2783.8938]
[ 270.3952 2285.4653]

Implemented GMM | Covariances
[[ 4102.75896297     0.        ]
 [    0.         27835.50030919]]
[[3562.61252657    0.        ]
 [   0.         7657.27381154]]
[[ 1213.73824836     0.        ]
 [    0.         14278.42560148]]

Implemented GMM | Weights
[0.18387372 0.38098183 0.43514445]
```

```
Implemented GMM | Mean values
[ 584.4374 2806.613 ]
[ 393.36813 1992.2206 ]
[ 448.70447 2517.3918 ]

Implemented GMM | Covariances
[[ 2180.2505767      0.        ]
 [    0.         49368.52581405]]
[[ 2459.57923616     0.        ]
 [    0.         15099.04960996]]
[[ 2742.44917077     0.        ]
 [    0.         29915.9451524 ]]

Implemented GMM | Weights
[0.09931042 0.43347269 0.46721689]
```

*Figure 3. Mean, Variance and GMM Weights for Phoneme 1 (Left) and Phoneme 2 (Right) - K=3*



```
Implemented GMM | Mean values
[ 434.72556 3102.1875 ]
[ 280.3095 2773.225 ]
[ 327.09106 3190.7512 ]
[ 358.34674 2810.552  ]
[ 301.86514 2344.044  ]
[ 243.85498 2234.6045 ]

Implemented GMM | Covariances
[[  5449.13058674      0.        ]
 [     0.         110961.78997017]]
[[1771.74002115    0.        ]
 [   0.         5580.84769055]]
[[ 1048.08351201     0.        ]
 [    0.         24715.34446231]]
[[ 560.10301165    0.        ]
 [   0.         9994.94047013]]
[[ 399.91324706    0.        ]
 [   0.         7251.95207571]]
[[  356.26949942     0.        ]
 [    0.         14193.34763953]]

Implemented GMM | Weights
[0.05960172 0.23984767 0.15158831 0.11493438 0.1984441  0.23558381]
```

```
Implemented GMM | Mean values
[ 656.71094 3150.859  ]
[ 564.0104 2703.8987]
[ 406.22076 2272.12   ]
[ 385.46677 1977.5056 ]
[ 477.59824 2603.9385 ]
[ 473.58502 2345.3474 ]

Implemented GMM | Covariances
[[  290.33020009     0.        ]
 [    0.         11605.94457334]]
[[  940.47542713     0.        ]
 [    0.         24215.40759898]]
[[ 1041.17930064     0.        ]
 [    0.         89153.60568812]]
[[3325.83310656    0.        ]
 [   0.         7758.5513785 ]]
[[1503.30154054    0.        ]
 [   0.         5456.35619626]]
[[ 133.41045217    0.        ]
 [   0.         2431.98219524]]

Implemented GMM | Weights
[0.01961561 0.0977423  0.37470415 0.26448451 0.18175609 0.06169735]
```

*Figure 4. Mean, Variance and GMM Weights for Phoneme 1 (Left) and Phoneme 2 (Right) - K=6*

The final Gaussian Models are given below:

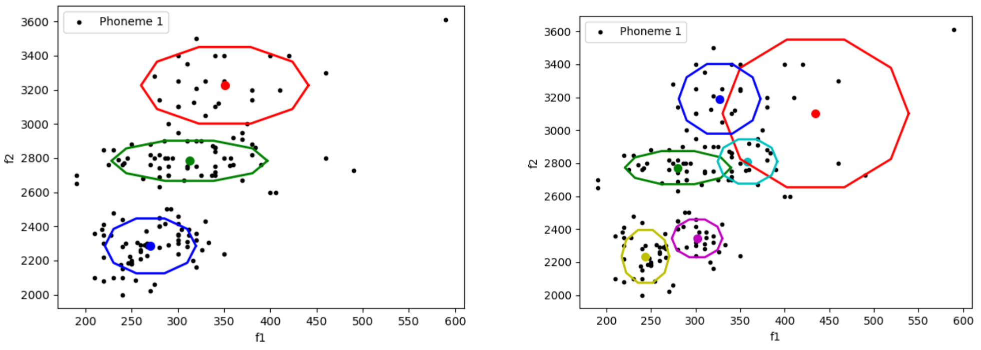


*Figure 5. Generated Gaussians Phoneme 1: K =3 (Left) and K=6 (Right)*

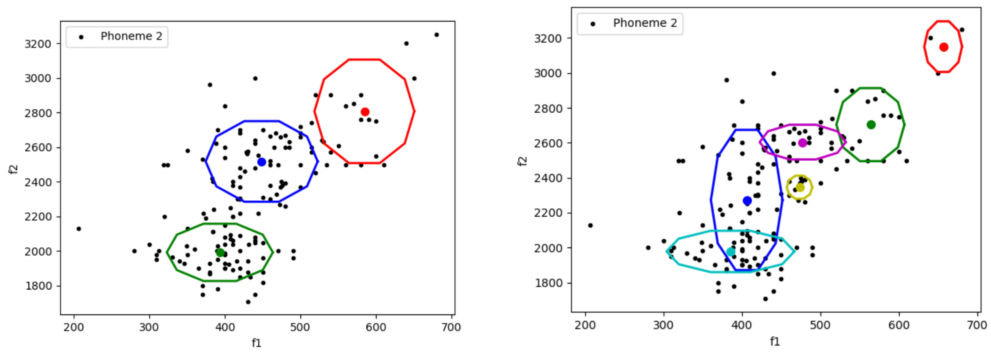


*Figure 6. Generated Gaussians Phoneme 2: K =3 (Left) and K=6 (Right)*

Again, note that these experiments were run multiple times for all of the Phoneme and K combinations. Even though clusters initialize from different places; they converge to similar shapes and layouts.

In **Code for Task 2** section; the code written or altered is shown in bold.

## Code for Task 2:

```
from plot_data import *
from sklearn.preprocessing import normalize
from get_predictions import *
from plot_gaussians import *

# File that contains the data
data_npy_file = 'data/PB_data.npy'

# Loading data from .npy file
data = np.load(data_npy_file, allow_pickle=True)
data = np.ndarray.tolist(data)

# Make a folder to save the figures
figures_folder = os.path.join(os.getcwd(), 'figures')
if not os.path.exists(figures_folder):
    os.makedirs(figures_folder, exist_ok=True)

# Array that contains the phoneme ID (1-10) of each sample
phoneme_id = data['phoneme_id']
# frequencies f1 and f2
f1 = data['f1']
f2 = data['f2']

# Initialize array containing f1 & f2, of all phonemes.
X_full = np.zeros((len(f1), 2))
# ########################################
# Write your code here
# Store f1 in the first column of X_full, and f2 in the second column of X_full
X_full[:, 0] = f1
X_full[:, 1] = f2
# ########################################/
X_full = X_full.astype(np.float32)

# We will train a GMM with k components, on a selected phoneme id which is stored in variable "p_id"
# number of GMM components
k = 3 # altered to 6 as well
# you can use the p_id variable, to store the ID of the chosen phoneme that will be used (e.g. phoneme 1, or phoneme 2)
p_id = 1 # altered to 2 as well

# ########################################
# Write your code here

# Create an array named "X_phoneme", containing only samples that belong to the chosen phoneme.
# The shape of X_phoneme will be two-dimensional. Each row will represent a sample of the dataset,
# and each column will represent a feature (e.g. f1 or f2)
# Fill X_phoneme with the samples of X_full that belong to the chosen phoneme
# To fill X_phoneme, you can leverage the phoneme_id array, that contains the ID of each sample of X_full
X_phoneme = X_full[phoneme_id == p_id, :]

# #########################################

# Plot array containing the chosen phoneme

# Create a figure and a subplot
fig, ax1 = plt.subplots()

title_string = 'Phoneme {}'.format(p_id)
# plot the samples of the dataset, belonging to the chosen phoneme (f1 & f2, phoneme 1 or 2)
plot_data(X=X_phoneme, title_string=title_string, ax=ax1)
# save the plotted points of phoneme 1 as a figure
plot_filename = os.path.join(os.getcwd(), 'figures', 'dataset_phoneme_{}.png'.format(p_id))
plt.savefig(plot_filename)

###############################################
# Train a GMM with k components, on the chosen phoneme

# as dataset X, we will use only the samples of the chosen phoneme
X = X_phoneme.copy()
# get number of samples
```

```
N = X.shape[0]
# get dimensionality of our dataset
D = X.shape[1]

# common practice : GMM weights initially set as 1/k
p = np.ones(k)/k
# GMM means are picked randomly from data samples
random_indices = np.floor(N*np.random.rand(k))
random_indices = random_indices.astype(int)
mu = X[random_indices, :]  # shape kxD
# covariance matrices
s = np.zeros((k, D, D))  # shape kxDxD
# number of iterations for the EM algorithm
n_iter = 100

# initialize covariances
for i in range(k):
    cov_matrix = np.cov(X.transpose())
    # initially set to fraction of data covariance
    s[i, :, :] = cov_matrix/k

# Initialize array Z that will get the predictions of each Gaussian on each sample
Z = np.zeros((N, k))  # shape Nxk

###############################
# run Expectation Maximization algorithm for n_iter iterations
for t in range(n_iter):
    print('Iteration {:03}/{:03}'.format(t+1, n_iter))

    # Do the E-step
    Z = get_predictions(mu, s, p, X)
    Z = normalize(Z, axis=1, norm='l1')

    # Do the M-step:
    for i in range(k):
        mu[i, :] = np.matmul(X.transpose(), Z[:, i]) / np.sum(Z[:, i])
        # We will fit Gaussian's with diagonal covariance matrices
        mu_i = mu[i, :]
        mu_i = np.expand_dims(mu_i, axis=1)
        mu_i_repeated = np.repeat(mu_i, N, axis=1)
        X_minus_mu = (X.transpose() - mu_i_repeated)**2
        res_1 = np.squeeze(np.matmul(X_minus_mu, np.expand_dims(Z[:, i], axis=1)))/np.sum(Z[:, i])
        s[i, :, :] = np.diag(res_1)
        p[i] = np.mean(Z[:, i])
    ax1.clear()
    # plot the samples of the dataset, belonging to the chosen phoneme (f1 & f2, phoneme 1 or 2)
    plot_data(X=X_phoneme, title_string=title_string, ax=ax1)
    # Plot gaussian's after each iteration
    plot_gaussians(ax1, 2*s, mu)
print('\nFinished.\n')

# save the trained GMM's plot as a figure
plot_filename = os.path.join(os.getcwd(), 'figures', 'GMM_phoneme_{}_k_{}.png'.format(p_id, k))
plt.savefig(plot_filename)

print('Implemented GMM | Mean values')
for i in range(k):
    print(mu[i])
print('')
print('Implemented GMM | Covariances')
for i in range(k):
    print(s[i, :, :])
print('')
print('Implemented GMM | Weights')
print(p)
print('')

# enter non-interactive mode of matplotlib, to keep figures open
plt.ioff()
plt.show()
```

```
# Create a dictionary to store the trained GMM's parameters
GMM_parameters = {'mu': mu, 's': s, 'p': p}


# Save the trained GMM's parameters in a numpy file
npy_filename = 'data/GMM_params_phoneme_{:02}_k_{:02}.npy'.format(p_id, k)
np.save(npy_filename, GMM_parameters)
```

**Task 3:**

This task will use the models that are saved in Task 2. Both phonemes 1 and 2 are used in this task. The following is the plot of all points:

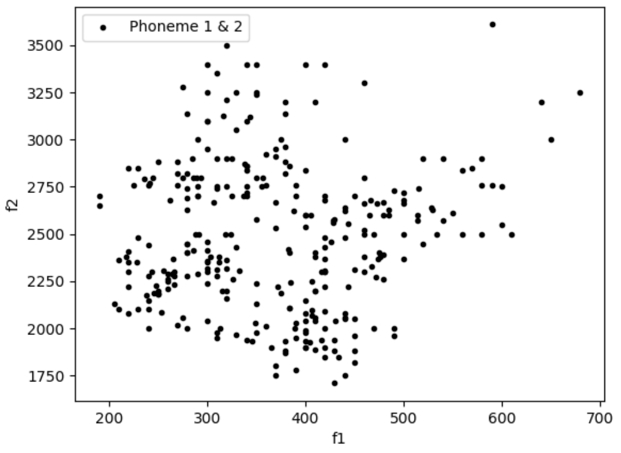


*Figure 7. Scatter plot for Phonemes 1 and 2*

The models can be loaded to the memory using the code segment below. Note that this is used both for phoneme 1 and 2 seperately.

```
phoneme1_model = 'data/GMM_params_phoneme_{:02}_k_{:02}.npy'.format(1, k)  # load the model

X = X_phonemes_1_2.copy()

model1_data = np.load(phoneme1_model, allow_pickle=True)

model1_data = np.ndarray.tolist(model1_data)

Z1 = get_predictions(model1_data['mu'], model1_data['s'], model1_data['p'], X)

pred1 = np.sum(Z1, axis=1)
```

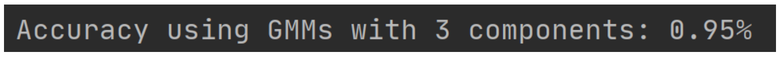
As the GMM training is done in previous task, using model parameters and get_predictions() method which the details are explained in Task 2; predictions on the datapoints can be taken. This method constructs GMM and uses the given set of phonemes to give predictions. It is also the expectation step of the expectation maximization algorithm.

Note that the models are separately trained for Phoneme 1 and 2, respectively. Here the dataset contains elements from both phoneme 1 and 2. As Z, for each model, contains likelihood of data points being in an GMM cluster given the parameters; both model's output's Z can be used to classify which class does the dataset points belong to. The following code segment is used for this part:

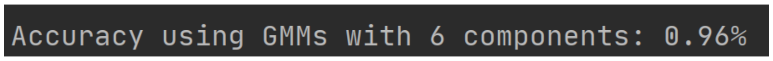
```
phoneme1_model = 'data/GMM_params_phoneme_{:02}_k_{:02}.npy'.format(1, k)  # load the model

X = X_phonemes_1_2.copy()

model1_data = np.load(phoneme1_model, allow_pickle=True)

model1_data = np.ndarray.tolist(model1_data)

Z1 = get_predictions(model1_data['mu'], model1_data['s'], model1_data['p'], X)

pred1 = np.sum(Z1, axis=1)

# Predictions on Phoneme 2 model:

Phoneme2_model = 'data/GMM_params_phoneme_{:02}_k_{:02}.npy'.format(2, k) # load the model

X = X_phonemes_1_2.copy()

Model2_data = np.load(phoneme1_model, allow_pickle=True)

Model2_data = np.ndarray.tolist(model1_data)

Z2 = get_predictions(model1_data['mu'], model1_data['s'], model1_data['p'], X)

pred2 = np.sum(Z2, axis=1)

# Accuracy calculations:

pred1_bigger_2 = np.ones(len(X)) * 2  # gives class 2

pred1_bigger_2[pred1 >= pred2] = 1  # assigns class 1

labels = phoneme_id[np.logical_or(phoneme_id == 1, phoneme_id == 2)]

accuracy = np.sum(pred1_bigger_2 == labels)/X.shape[0]
```

After returning the predictions on all clusters per model; the likelihoods are summed (pred1 and pred2). Classification is achieved by comparing the cumulative likelihoods of Z1 and Z2 from models that are pretrained on Phoneme 1 and Phoneme 2 models, respectively.

The following are the results outputted from **task_3.py** using k=3 and k=6 respectively.



```
Accuracy using GMMs with 3 components: 0.95%
```



```
Accuracy using GMMs with 6 components: 0.96%
```

*Figure 8. Accuracy of GMMs using different number of Gaussians*

As for this section, using K=6 gave slightly better results, at the cost of time complexity. K=3 is much faster than K=6 while training the models.

## Code for Task 3:

```
import numpy as np
import os
import matplotlib.pyplot as plt
from print_values import *
from plot_data_all_phonemes import *
from plot_data import *
import random
from sklearn.preprocessing import normalize
from get_predictions import *
from plot_gaussians import *

# File that contains the data
data_npy_file = 'data/PB_data.npy'

# Loading data from .npy file
data = np.load(data_npy_file, allow_pickle=True)
data = np.ndarray.tolist(data)

# Make a folder to save the figures
figures_folder = os.path.join(os.getcwd(), 'figures')
if not os.path.exists(figures_folder):
    os.makedirs(figures_folder, exist_ok=True)

# Array that contains the phoneme ID (1-10) of each sample
phoneme_id = data['phoneme_id']
# frequencies f1 and f2
f1 = data['f1']
f2 = data['f2']

# Initialize array containing f1 & f2, of all phonemes.
X_full = np.zeros((len(f1), 2))
# ########################################
# Write your code here
# Store f1 in the first column of X_full, and f2 in the second column of X_full
```

**X_full[:, 0] = f1**

**X_full[:, 1] = f2**

```
# ########################################
X_full = X_full.astype(np.float32)

# number of GMM components
```

**k = 6 # also k=3 is used**

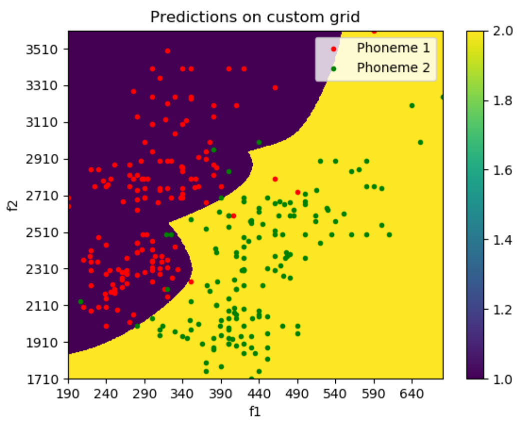
```
# ########################################
# Write your code here
# Create an array named "X_phonemes_1_2", containing only samples that belong to
# phoneme 1 and samples that belong to phoneme 2.
# The shape of X_phonemes_1_2 will be two-dimensional. Each row will represent a sample of the dataset, and each column
# will represent a feature (e.g. f1 or f2)
# Fill X_phonemes_1_2 with the samples of X_full that belong to the chosen phonemes
# To fill X_phonemes_1_2, you can leverage the phoneme_id array, that contains the ID of each sample of X_full
```

**X_phonemes_1_2 = X_full[np.logical_or(phoneme_id == 1, phoneme_id == 2), :]**

```
# ########################################/

# Plot array containing the chosen phonemes
# Create a figure and a subplot
fig, ax1 = plt.subplots()

title_string = 'Phoneme 1 & 2'
# plot the samples of the dataset, belonging to the chosen phoneme (f1 & f2, phoneme 1 & 2)
plot_data(X=X_phonemes_1_2, title_string=title_string, ax=ax1)
# save the plotted points of phoneme 1 as a figure
plot_filename = os.path.join(os.getcwd(), 'figures', 'dataset_phonemes_1_2.png')
plt.savefig(plot_filename)

# ########################################
# Write your code here
# Get predictions on samples from both phonemes 1 and 2, from a GMM with k components, pretrained on phoneme 1
# Get predictions on samples from both phonemes 1 and 2, from a GMM with k components, pretrained on phoneme 2
# Compare these predictions for each sample of the dataset, and calculate the accuracy,
# and store it in a scalar variable named "accuracy"


# Predictions on Phoneme 1 model:
phoneme1_model = 'data/GMM_params_phoneme_{:02}_k_{:02}.npy'.format(1, k)  # load the model
X = X_phonemes_1_2.copy()
model1_data = np.load(phoneme1_model, allow_pickle=True)
model1_data = np.ndarray.tolist(model1_data)
Z1 = get_predictions(model1_data['mu'], model1_data['s'], model1_data['p'], X)
pred1 = np.sum(Z1, axis=1)


# Predictions on Phoneme 2 model:
phoneme1_model = 'data/GMM_params_phoneme_{:02}_k_{:02}.npy'.format(2, k) # load the model
X = X_phonemes_1_2.copy()
model1_data = np.load(phoneme1_model, allow_pickle=True)
model1_data = np.ndarray.tolist(model1_data)
Z2 = get_predictions(model1_data['mu'], model1_data['s'], model1_data['p'], X)
pred2 = np.sum(Z2, axis=1)


# Accuracy calculations:
pred1_bigger_2 = np.ones(len(X)) * 2  # gives class 2
pred1_bigger_2[pred1 >= pred2] = 1  # assigns class 1


labels = phoneme_id[np.logical_or(phoneme_id == 1, phoneme_id == 2)]
accuracy = np.sum(pred1_bigger_2 == labels)/X.shape[0]
# ########################################/


print('Accuracy using GMMs with {} components: {:.2f}%'.format(k, accuracy))


################################################
# enter non-interactive mode of matplotlib, to keep figures open
plt.ioff()
plt.show()
```

**Task 4:**

This Task is very similar to Part 3. The following results are achieved after running this section. Details will be given below:



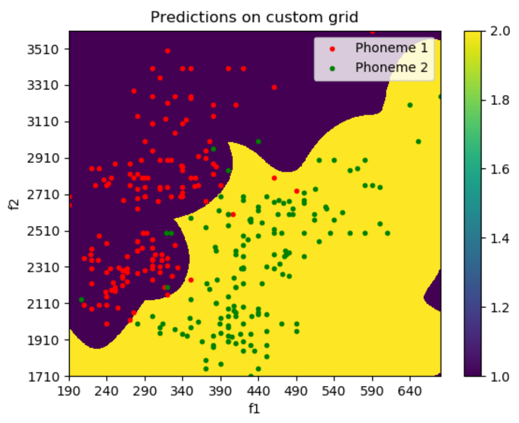


*Figure 9. Predictions on Grid K=3 (Left), K=6 (Right)*

In task 3; classification was made based on the datapoints shown above. The grid boundaries above also prove that both K=3 and K=6 are very accurate models. With Occam's razor; picking K=3 as the best model could be better; in terms of balancing computational performance and accuracy.

The code below creates a linear spaced vector for f1 and f2 given the minimum and maximum values that F1 and F2 can take; given Phonemes 1 and 2. Mesh grid is used to generate a grid of points to classify. Flattening is used to utilize vector notation and operations.

```
ax_f1 = np.linspace(min_f1, max_f1, N_f1)

ax_f2 = np.linspace(min_f2, max_f2, N_f2)

xx, yy = np.meshgrid(ax_f1, ax_f2)


x = xx.flatten()

y = yy.flatten()


samples = np.stack((x, y)).transpose()  # fix dimensions
```

Instead of actual dataset, this time the grid is used to have predictions using the pre-trained models. Everything in this point is the same with Task 3. Instead of using an accuracy metric; the predictions are reshaped back into the original meshgrid; using an classification matrix M; later to be displayed on a plot along with actual datapoints with their respective classes.

```
# Accuracy calculations:
pred1_bigger_2 = np.ones(len(S2)) * 2  # gives class 2
pred1_bigger_2[pred1 >= pred2] = 1  # assigns class 1
M = pred1_bigger_2.reshape(N_f2, N_f1)
```

Note that this classification matrix is already been displayed by Figure 9. This matrix constructs the background of the scatterplot figure. Additionally, the plot expected the datapoints X to be sorted as this assumption is used while assigning color to individual datapoints. This problem is solved by altering given code in the following way:

```
## Fix the coloring issue here. initial plot assumes that the samples are ordered.
# N_samples = int(X.shape[0]/2)
# plt.scatter(X[:N_samples, 0] - min_f1, X[:N_samples, 1] - min_f2, marker='.', color='red', label='Phoneme 1')
# plt.scatter(X[N_samples:, 0] - min_f1, X[N_samples:, 1] - min_f2, marker='.', color='green', label='Phoneme 2')
# Change above to this
targets = phoneme_id[np.isin(phoneme_id, [1, 2])]
X1 = X[targets == 1]
X2 = X[targets == 2]
plt.scatter(X1[:, 0] - min_f1, X1[:, 1] - min_f2, marker='.', color='red', label='Phoneme 1')
plt.scatter(X2[:, 0] - min_f1, X2[:, 1] - min_f2, marker='.', color='green', label='Phoneme 2')
```

## Code for Task 4:

```
import numpy as np

import os

import matplotlib.pyplot as plt

from print_values import *

from plot_data_all_phonemes import *

from plot_data import *

import random

from sklearn.preprocessing import normalize

from get_predictions import *

from plot_gaussians import *


# File that contains the data

data_npy_file = 'data/PB_data.npy'


# Loading data from .npy file

data = np.load(data_npy_file, allow_pickle=True)

data = np.ndarray.tolist(data)


# Make a folder to save the figures

figures_folder = os.path.join(os.getcwd(), 'figures')

if not os.path.exists(figures_folder):

    os.makedirs(figures_folder, exist_ok=True)


# Array that contains the phoneme ID (1-10) of each sample

phoneme_id = data['phoneme_id']

# frequencies f1 and f2

f1 = data['f1']

f2 = data['f2']


# Initialize array containing f1 & f2, of all phonemes.

X_full = np.zeros((len(f1), 2))

# ########################################

# Write your code here

# Store f1 in the first column of X_full, and f2 in the second column of X_full
```

```
X_full[:, 0] = f1

X_full[:, 1] = f2

# ########################################/

X_full = X_full.astype(np.float32)


# number of GMM components

k = 3 # also k=6 is tried


# ########################################

# Write your code here


# Create an array named "X_phonemes_1_2", containing only samples that belong to

# phoneme 1 and samples that belong to phoneme 2.

# The shape of X_phonemes_1_2 will be two-dimensional. Each row will represent a sample of the dataset,

# and each column will represent a feature (e.g. f1 or f2)

# Fill X_phonemes_1_2 with the samples of X_full that belong to the chosen phonemes

# To fill X_phonemes_1_2, you can leverage the phoneme_id array, that contains the ID of each sample of X_full


X_phonemes_1_2 = X_full[np.logical_or(phoneme_id == 1, phoneme_id == 2), :]


# ########################################


# as dataset X, we will use only the samples of phoneme 1 and 2

X = X_phonemes_1_2.copy()


min_f1 = int(np.min(X[:, 0]))

max_f1 = int(np.max(X[:, 0]))

min_f2 = int(np.min(X[:, 1]))

max_f2 = int(np.max(X[:, 1]))

N_f1 = max_f1 - min_f1

N_f2 = max_f2 - min_f2

print('f1 range: {}-{} | {} points'.format(min_f1, max_f1, N_f1))  # N_f1

print('f2 range: {}-{} | {} points'.format(min_f2, max_f2, N_f2))  # N_f2


# ########################################
# Write your code here

# Create a custom grid of shape N_f1 x N_f2
# The grid will span all the values of (f1, f2) pairs,
# between [min_f1, max_f1] on f1 axis, and between [min_f2, max_f2] on f2 axis
ax_f1 = np.linspace(min_f1, max_f1, N_f1)
ax_f2 = np.linspace(min_f2, max_f2, N_f2)
xx, yy = np.meshgrid(ax_f1, ax_f2)

x = xx.flatten()
y = yy.flatten()

samples = np.stack((x, y)).transpose()  # fix dimensions

# Then, classify each point [i.e., each (f1, f2) pair] of that grid,
# to either phoneme 1, or phoneme 2, using the two trained GMMs
# Predictions on Phoneme 1 model:
phoneme1_model = 'data/GMM_params_phoneme_{:02}_k_{:02}.npy'.format(1, k)  # load the model
S1 = samples.copy()
model1_data = np.load(phoneme1_model, allow_pickle=True)
model1_data = np.ndarray.tolist(model1_data)

Z1 = get_predictions(model1_data['mu'], model1_data['s'], model1_data['p'], S1)
pred1 = np.max(Z1, axis=1)

# Predictions on Phoneme 2 model:
phoneme2_model = 'data/GMM_params_phoneme_{:02}_k_{:02}.npy'.format(2, k)  # load the model
S2 = samples.copy()
model2_data = np.load(phoneme2_model, allow_pickle=True)
model2_data = np.ndarray.tolist(model2_data)
Z2 = get_predictions(model2_data['mu'], model2_data['s'], model2_data['p'], S2)
pred2 = np.sum(Z2, axis=1)

# Accuracy calculations:
pred1_bigger_2 = np.ones(len(S2)) * 2  # gives class 2
```

```
pred1_bigger_2[pred1 >= pred2] = 1  # assigns class 1

M = pred1_bigger_2.reshape(N_f2, N_f1)

# Do predictions, using GMM trained on phoneme 1, on custom grid
# Do predictions, using GMM trained on phoneme 2, on custom grid

# Compare these predictions, to classify each point of the grid
# Store these prediction in a 2D numpy array named "M", of shape N_f2 x N_f1
# (the first dimension is f2 so that we keep f2 in the vertical axis of the plot)
# M should contain "0.0" in the points that belong to phoneme 1 and "1.0" in the points that belong to phoneme 2
# ########################################/

################################################
# Visualize predictions on custom grid

# Create a figure
# fig = plt.figure()
fig, ax = plt.subplots()

# use aspect='auto' (default is 'equal'), to force the plotted image to be square, when dimensions are unequal
plt.imshow(M, aspect='auto')

# set label of x axis
ax.set_xlabel('f1')
# set label of y axis
ax.set_ylabel('f2')

# set limits of axes
plt.xlim((0, N_f1))
plt.ylim((0, N_f2))

# set range and strings of ticks on axes
x_range = np.arange(0, N_f1, step=50)
x_strings = [str(x+min_f1) for x in x_range]
```

```
plt.xticks(x_range, x_strings)
y_range = np.arange(0, N_f2, step=200)
y_strings = [str(y+min_f2) for y in y_range]
plt.yticks(y_range, y_strings)

# set title of figure
title_string = 'Predictions on custom grid'
plt.title(title_string)

# add a color bar
plt.colorbar()

## Fix the coloring issue here. initial plot assumes that the samples are ordered.
# N_samples = int(X.shape[0]/2)
# plt.scatter(X[:N_samples, 0] - min_f1, X[:N_samples, 1] - min_f2, marker='.', color='red', label='Phoneme 1')
# plt.scatter(X[N_samples:, 0] - min_f1, X[N_samples:, 1] - min_f2, marker='.', color='green', label='Phoneme 2')

targets = phoneme_id[np.isin(phoneme_id, [1, 2])]
X1 = X[targets == 1]
X2 = X[targets == 2]
plt.scatter(X1[:, 0] - min_f1, X1[:, 1] - min_f2, marker='.', color='red', label='Phoneme 1')
plt.scatter(X2[:, 0] - min_f1, X2[:, 1] - min_f2, marker='.', color='green', label='Phoneme 2')

# add legend to the subplot
plt.legend()

# save the plotted points of the chosen phoneme, as a figure
plot_filename = os.path.join(os.getcwd(), 'figures', 'GMM_predictions_on_grid.png')
plt.savefig(plot_filename)

# ################################################
# enter non-interactive mode of matplotlib, to keep figures open
plt.ioff()
plt.show()
```

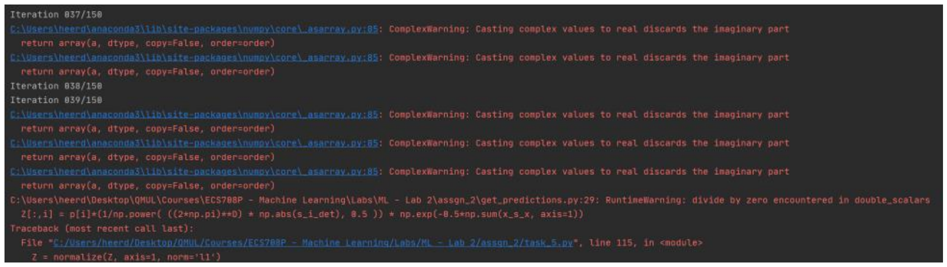
**Task 5:**

In the task 1 to 4; all models were trained on covariance matrices where non-diagonal entries were zero. This is done by using np.diag() option to just work on variances ignoring covariance terms. Outputs of covariances matrices in Task 2 is an example to this.

This implementation is putting a constraint on the covariance matrix, forcing all non-diagonal entries to zero. Such diagonal matrix will always be invertible, so this implementation circumvents the singularity problem.

This diagonal matrix also implies that F1 and F2 is linearly independent from each other. Although this is a strong assumption it holds quite well. Using the entire covariance matrix (as used in Task 5) in Task 2, in some runs it is seen that the covariance matrix became singular; causing GM model to fail. In many cases training hold up well.

In Task 5, additional feature vector F1+F2 is added to the dataset. This will third feature linearly dependent on the first two features, therefore it will cause singularity issues as shown below:

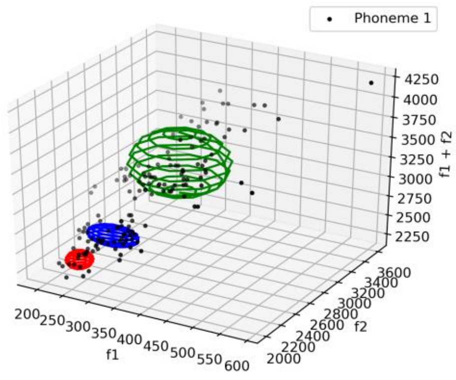


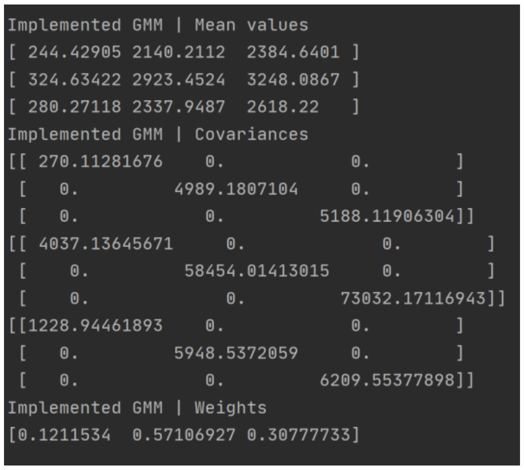
```
Iteration 037/150
C:\Users\heerd\anaconda3\lib\site-packages\numpy\core\_asarray.py:85: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
C:\Users\heerd\anaconda3\lib\site-packages\numpy\core\_asarray.py:85: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
Iteration 038/150
Iteration 039/150
C:\Users\heerd\anaconda3\lib\site-packages\numpy\core\_asarray.py:85: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
C:\Users\heerd\anaconda3\lib\site-packages\numpy\core\_asarray.py:85: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
C:\Users\heerd\anaconda3\lib\site-packages\numpy\core\_asarray.py:85: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
C:\Users\heerd\Desktop\QMUL\Courses\ECS708P - Machine Learning\Labs\ML - Lab 2\assgn_2\get_predictions.py:29: RuntimeWarning: divide by zero encountered in double_scalars
  Z[:,i] = p[i]*(1/np.power( ((2*np.pi)**D) * np.abs(s_i_det), 0.5 )) * np.exp(-0.5*np.sum(x_s_x, axis=1))
Traceback (most recent call last):
  File "C:/Users/heerd/Desktop/QMUL/Courses/ECS708P - Machine Learning/Labs/ML - Lab 2/assgn_2/task_5.py", line 115, in <module>
    Z = normalize(Z, axis=1, norm='l1')
```

*Figure 10. Problems in training due to singularity. Crashed.*

The reason for the crash is that the inverse of the covariance matrix does not exist, so Python uses complex domain to solve this problem. It causes many issues on training such as the notation shown above regarding ignoring imaginary part of matrices or divide by zero. It also states that the matrix is singular as well. If the attributes of the dataset was linearly independent, then it would result with a diagonal covariance matrix which is always invertible.

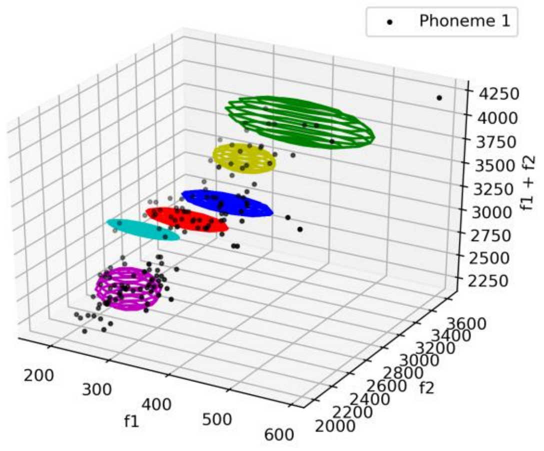
If task 2's diagonal covariance enforcement approach is used in Task 5; the singularity problem is fixed, and the following results are seen for K=3 below. Similar results are visible for K=6 as well.

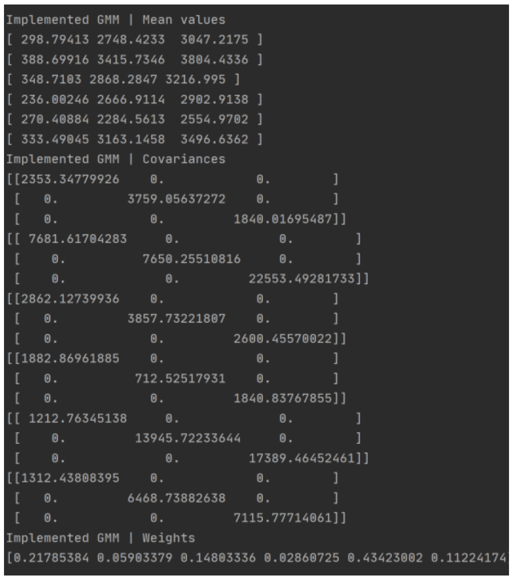




```
Implemented GMM | Mean values
[ 244.42905 2140.2112  2384.6401 ]
[ 324.63422 2923.4524  3248.0867 ]
[ 280.27118 2337.9487  2618.22   ]
Implemented GMM | Covariances
[[ 270.11281676    0.            0.        ]
 [   0.         4989.1807104     0.        ]
 [   0.            0.         5188.11906304]]
[[ 4037.13645671     0.             0.        ]
 [    0.         58454.01413015     0.        ]
 [    0.             0.         73032.17116943]]
[[1228.94461893    0.            0.        ]
 [   0.         5948.5372059     0.        ]
 [   0.            0.         6209.55377898]]
Implemented GMM | Weights
[0.1211534  0.57106927 0.30777733]
```

*Figure 11. Use of Diagonal Entries of Covariance Matrix to Avoid Singularity Issues (K=3, Phoneme 1)*





```
Implemented GMM | Mean values
[ 298.79413 2748.4233  3047.2175 ]
[ 388.69916 3415.7346  3804.4336 ]
[ 348.7103 2868.2847 3216.995 ]
[ 236.00246 2666.9114  2902.9138 ]
[ 270.40884 2284.5613  2554.9702 ]
[ 333.49045 3163.1458  3496.6362 ]
Implemented GMM | Covariances
[[2353.34779926    0.            0.        ]
 [   0.         3759.05637272    0.        ]
 [   0.            0.         1840.01695487]]
[[ 7681.61704283     0.             0.        ]
 [    0.          7650.25510816     0.        ]
 [    0.             0.         22553.49281733]]
[[2862.12739936    0.            0.        ]
 [   0.         3857.73221807    0.        ]
 [   0.            0.         2600.45570022]]
[[1882.86961885    0.            0.        ]
 [   0.          712.52517931    0.        ]
 [   0.            0.         1840.83767855]]
[[ 1212.76345138     0.             0.        ]
 [    0.         13945.72233644     0.        ]
 [    0.             0.         17389.46452461]]
[[1312.43808395    0.            0.        ]
 [   0.         6468.73882638    0.        ]
 [   0.            0.         7115.77714061]]
Implemented GMM | Weights
[0.21785384 0.05903379 0.14803336 0.02860725 0.43423002 0.11224174
```

*Figure 12. Use of Diagonal Entries of Covariance Matrix to Avoid Singularity Issues (K=6, Phoneme 1)*

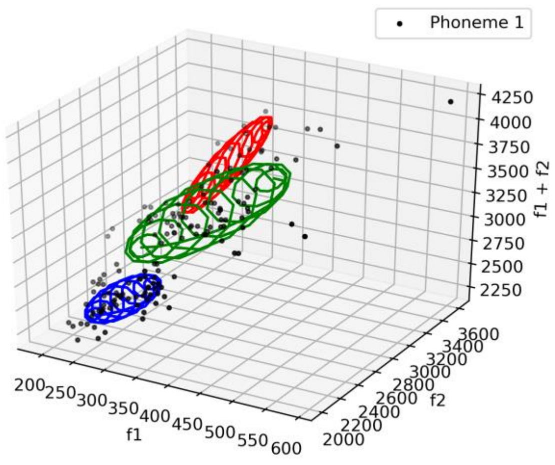
Note that in the Figure above Gaussian clusters do not fit to the dataset very well (as in Green one). From the covariance matrix it looks like F1, F2 and F1+F2 is linearly independent but it is not given F1+F2 is artificially created by using F1 and F2.  Also notice that in Figures above; all Gaussians are parallel to x and y axes and do not have any skewness on Z direction. This is because of independence assumption between all three axes. This is wrong; the model does not take this dependency into account. If the third dimension was ignored; and projection to F1-F2 axis was taken; the results will be very similar to the results in Task 2.

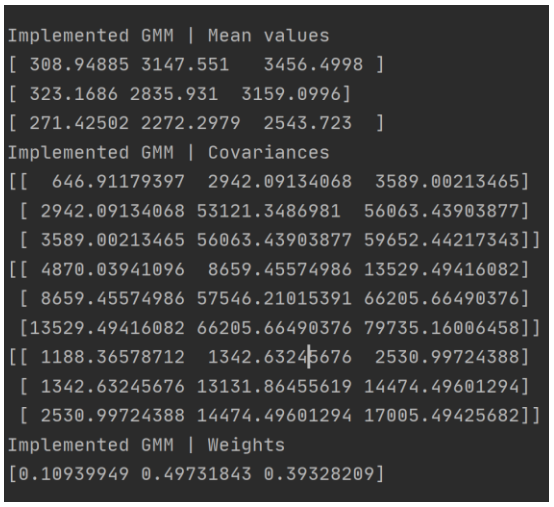
Another way of dealing with singularity is to regularize covariance by adding regularization to the covariance matrix. This approach builds upon the approach which Task 5 uses. The following is the implementation:

```
s[i, :, :] = np.matmul(term_1, term_2) / np.sum(Z[:, i])

 ############################################

 # Write your code here

 # Suggest ways of overcoming the singularity

 s[i, :, :] += 0.001 * np.identity(D)

 # ############################################

 p[i] = np.mean(Z[:, i])
```

The covariance matrix is regularized by adding an additional diagonal matrix with the same dimension as the original covariance matrix. For diagonal matrix, an identity matrix is used. For the diagonal entry factors a small amount of 0.001 is used. Different regularization parameters for each of the attributes is also possible.

In the EM procedure even if some of the variances turn out to be zero; this small term will make sure that all diagonal entries are non-zero so that an inverse could be taken. After implementing this term, the following results were obtained:

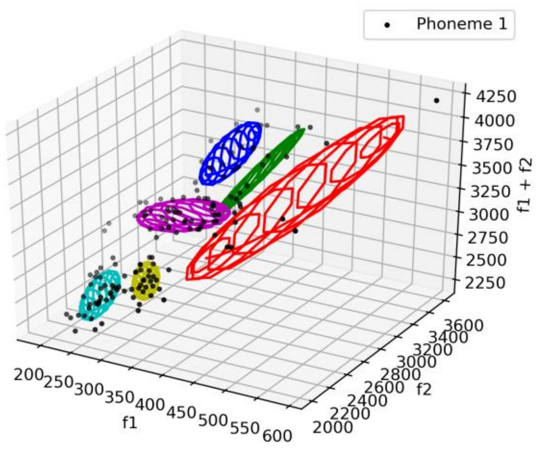


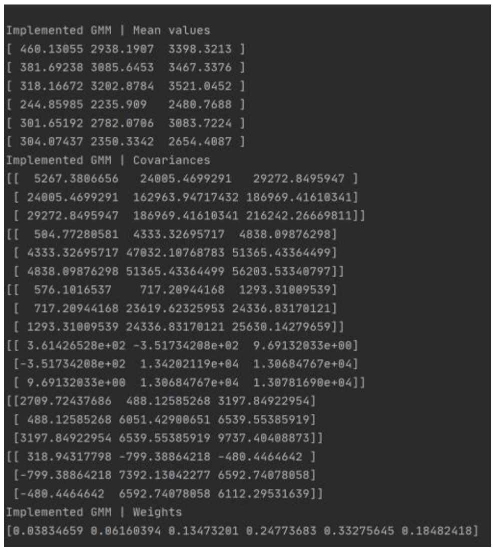


```
Implemented GMM | Mean values
[ 308.94885 3147.551   3456.4998 ]
[ 323.1686 2835.931  3159.0996]
[ 271.42502 2272.2979  2543.723  ]
Implemented GMM | Covariances
[[  646.91179397  2942.09134068  3589.00213465]
 [ 2942.09134068 53121.3486981  56063.43903877]
 [ 3589.00213465 56063.43903877 59652.44217343]]
[[ 4870.03941096  8659.45574986 13529.49416082]
 [ 8659.45574986 57546.21015391 66205.66490376]
 [13529.49416082 66205.66490376 79735.16006458]]
[[ 1188.36578712  1342.63245676  2530.99724388]
 [ 1342.63245676 13131.86455619 14474.49601294]
 [ 2530.99724388 14474.49601294 17005.49425682]]
Implemented GMM | Weights
[0.10939949 0.49731843 0.39328209]
```

*Figure 13. Use of Diagonal Regularization of Covariance Matrix to Avoid Singularity Issues (K=3, reg_factor = 0.001, Phoneme 1)*

For K=6 the results are shown below:





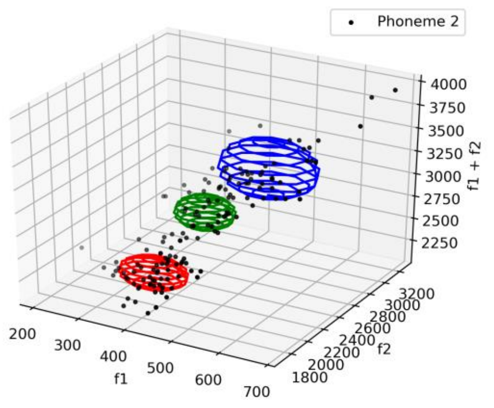
```
Implemented GMM | Mean values
[ 460.13055 2938.1907  3398.3213 ]
[ 381.69238 3085.6453  3467.3376 ]
[ 318.16672 3202.8784  3521.0452 ]
[ 244.85985 2235.909   2480.7688 ]
[ 301.65192 2782.0706  3083.7224 ]
[ 304.07437 2350.3342  2654.4087 ]
Implemented GMM | Covariances
[[  5267.3806656   24005.4699291   29272.8495947 ]
 [ 24005.4699291  162963.94717432 186969.41610341]
 [ 29272.8495947  186969.41610341 216242.26669811]]
[[  504.77280581  4333.32695717  4838.09876298]
 [ 4333.32695717 47032.10768783 51365.43364499]
 [ 4838.09876298 51365.43364499 56203.53340797]]
[[  576.1016537    717.20944168  1293.31009539]
 [  717.20944168 23619.62325953 24336.83170121]
 [ 1293.31009539 24336.83170121 25630.14279659]]
[[ 3.61426528e+02 -3.51734208e+02  9.69132033e+00]
 [-3.51734208e+02  1.34202119e+04  1.30684767e+04]
 [ 9.69132033e+00  1.30684767e+04  1.30781690e+04]]
[[2709.72437686  488.12585268 3197.84922954]
 [ 488.12585268 6051.42900651 6539.55385919]
 [3197.84922954 6539.55385919 9737.40408873]]
[[ 318.94317798 -799.38864218 -480.4464642 ]
 [-799.38864218 7392.13042277 6592.74078058]
 [-480.4464642  6592.74078058 6112.29531639]]
Implemented GMM | Weights
[0.03834659 0.06160394 0.13473201 0.24773683 0.33275645 0.18482418]
```

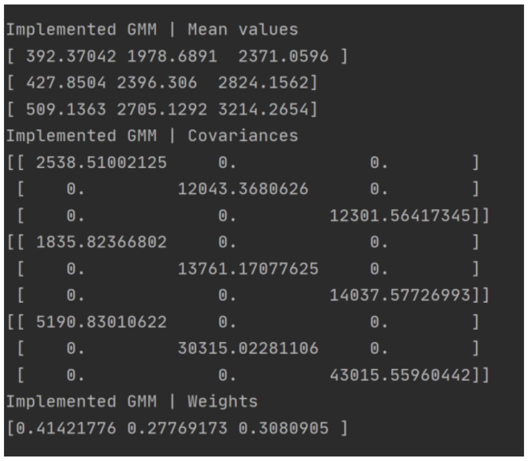
*Figure 14. Use of Diagonal Regularization of Covariance Matrix to Avoid Singularity Issues (K=6, reg_factor = 0.001, Phoneme 1)*

Note the differences in Figures 11, 12 and 13, 14. This time the Gaussians are also spread along the linearly dependent attribute F1+F2 and the covariance matrices are not diagonal anymore. The training on 150 iterations never give complex domain answer; divide by zero or other errors related to singularity. Without using the regularization the code never completed execution. As a small parameter is always forced to the diagonal entries, the variance of individual attributes was never reached to zero; hence the inverse of the covariance matrix existed.

Visually this regularization parameter forces the Gaussians to have at least a small thickness on all dimensions so that the multivariate Gaussian do not collapse on any of the dimensions. If some dimensions collapse; this will cause rank deficiency in covariance matrix. If a matrix become rank deficient in EM steps; it will not have an inverse and as it cannot recover back to stable state, the training will fail. Enforcing a small thickness using this method prevents this situation from happening.

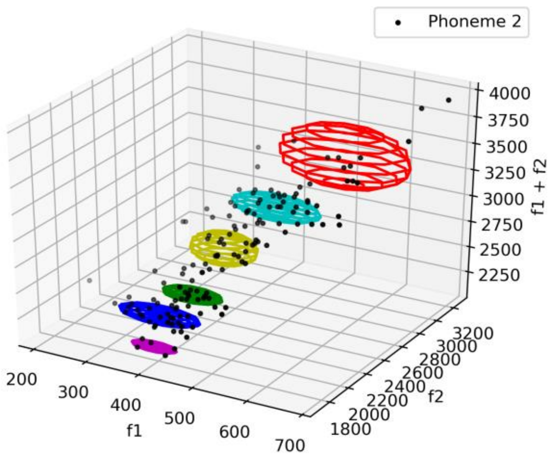
For the sake of completeness, the results of the discussion held above is also tried for Phoneme 2. The same discussion and similar results will follow.

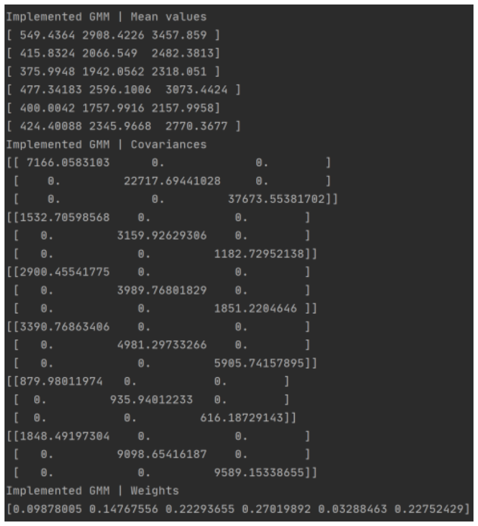




```
Implemented GMM | Mean values
[ 392.37042 1978.6891  2371.0596 ]
[ 427.8504 2396.306  2824.1562]
[ 509.1363 2705.1292 3214.2654]
Implemented GMM | Covariances
[[ 2538.51002125     0.              0.        ]
 [    0.         12043.3680626      0.        ]
 [    0.             0.         12301.56417345]]
[[ 1835.82366802     0.              0.        ]
 [    0.         13761.17077625     0.        ]
 [    0.             0.         14037.57726993]]
[[ 5190.83010622     0.              0.        ]
 [    0.         30315.02281106     0.        ]
 [    0.             0.         43015.55960442]]
Implemented GMM | Weights
[0.41421776 0.27769173 0.3080905 ]
```

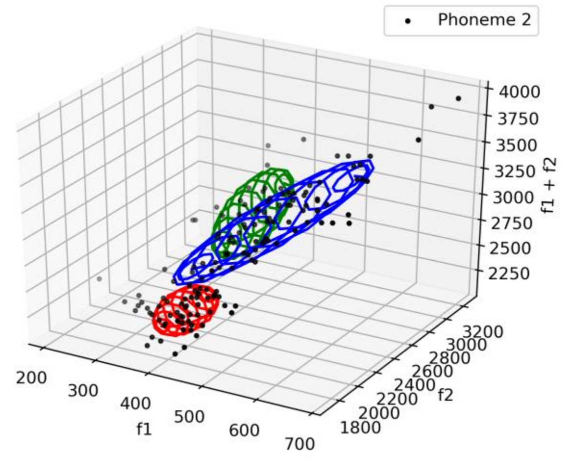
*Figure 15. Use of Diagonal Entries of Covariance Matrix to Avoid Singularity Issues (K=3, Phoneme 2)*

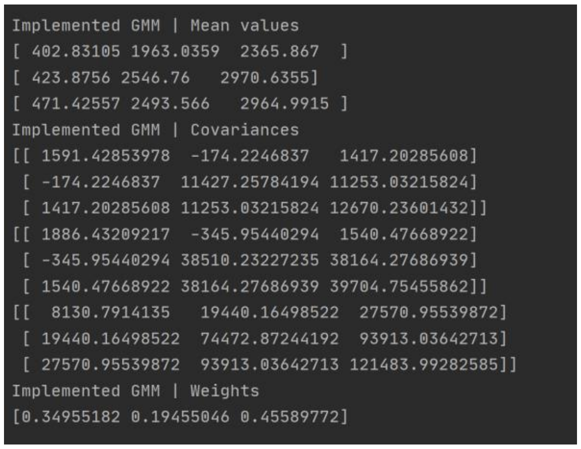




```
Implemented GMM | Mean values
[ 549.4364 2908.4226 3457.859 ]
[ 415.8324 2066.549  2482.3813]
[ 375.9948 1942.0562 2318.051 ]
[ 477.34183 2596.1006  3073.4424 ]
[ 400.0042 1757.9916 2157.9958]
[ 424.40088 2345.9668  2770.3677 ]
Implemented GMM | Covariances
[[ 7166.0583103      0.              0.        ]
 [    0.         22717.69441028     0.        ]
 [    0.             0.         37673.55381702]]
[[1532.70598568    0.            0.        ]
 [   0.         3159.92629306    0.        ]
 [   0.            0.         1182.72952138]]
[[2900.45541775    0.            0.        ]
 [   0.         3989.76801829    0.        ]
 [   0.            0.         1851.2204646 ]]
[[3390.76863406    0.            0.        ]
 [   0.         4981.29733266    0.        ]
 [   0.            0.         5905.74157895]]
[[879.98011974   0.            0.        ]
 [  0.         935.94012233   0.        ]
 [  0.           0.         616.18729143]]
[[1848.49197304    0.            0.        ]
 [   0.         9098.65416187    0.        ]
 [   0.            0.         9589.15338655]]
Implemented GMM | Weights
[0.09878005 0.14767556 0.22293655 0.27019892 0.03288463 0.22752429]
```

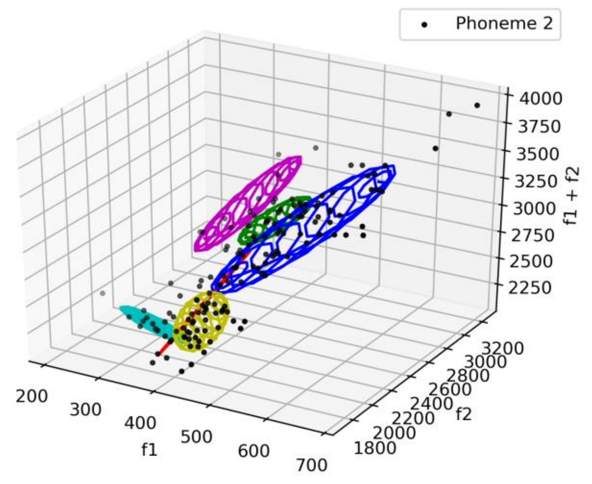
*Figure 16. Use of Diagonal Entries of Covariance Matrix to Avoid Singularity Issues (K=6, Phoneme 2)*

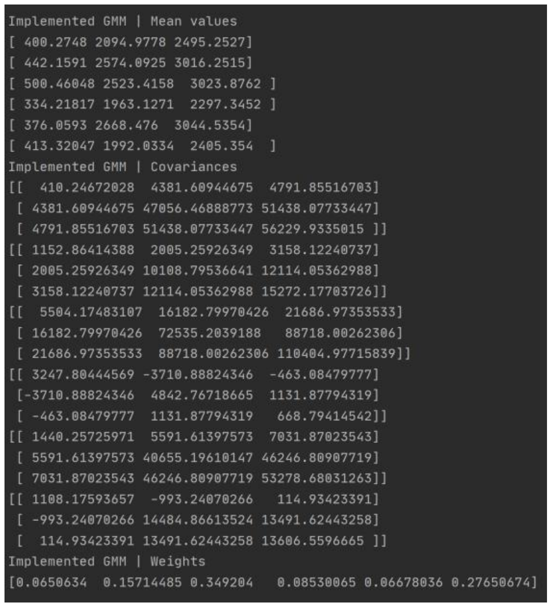




```
Implemented GMM | Mean values
[ 402.83105 1963.0359  2365.867  ]
[ 423.8756 2546.76   2970.6355]
[ 471.42557 2493.566   2964.9915 ]
Implemented GMM | Covariances
[[ 1591.42853978  -174.2246837   1417.20285608]
 [ -174.2246837  11427.25784194 11253.03215824]
 [ 1417.20285608 11253.03215824 12670.23601432]]
[[ 1886.43209217  -345.95440294  1540.47668922]
 [ -345.95440294 38510.23227235 38164.27686939]
 [ 1540.47668922 38164.27686939 39704.75455862]]
[[  8130.7914135   19440.16498522  27570.95539872]
 [ 19440.16498522  74472.87244192  93913.03642713]
 [ 27570.95539872  93913.03642713 121483.99282585]]
Implemented GMM | Weights
[0.34955182 0.19455046 0.45589772]
```

*Figure 17. Use of Diagonal Regularization of Covariance Matrix to Avoid Singularity Issues (K=3, reg_factor = 0.001, Phoneme 2)*





```
Implemented GMM | Mean values
[ 400.2748 2094.9778 2495.2527]
[ 442.1591 2574.0925 3016.2515]
[ 500.46048 2523.4158  3023.8762 ]
[ 334.21817 1963.1271  2297.3452 ]
[ 376.0593 2668.476  3044.5354]
[ 413.32047 1992.0334  2405.354  ]
Implemented GMM | Covariances
[[  410.24672028  4381.60944675  4791.85516703]
 [ 4381.60944675 47056.46888773 51438.07733447]
 [ 4791.85516703 51438.07733447 56229.9335015 ]]
[[ 1152.86414388  2005.25926349  3158.12240737]
 [ 2005.25926349 10108.79536641 12114.05362988]
 [ 3158.12240737 12114.05362988 15272.17703726]]
[[  5504.17483107  16182.79970426  21686.97353533]
 [ 16182.79970426  72535.2039188   88718.00262306]
 [ 21686.97353533  88718.00262306 110404.97715839]]
[[ 3247.80444569 -3710.88824346  -463.08479777]
 [-3710.88824346  4842.76718665  1131.87794319]
 [ -463.08479777  1131.87794319   668.79414542]]
[[ 1440.25725971  5591.61397573  7031.87023543]
 [ 5591.61397573 40655.19610147 46246.80907719]
 [ 7031.87023543 46246.80907719 53278.68031263]]
[[ 1108.17593657  -993.24070266   114.93423391]
 [ -993.24070266 14484.86613524 13491.62443258]
 [  114.93423391 13491.62443258 13606.5596665 ]]
Implemented GMM | Weights
[0.0650634  0.15714485 0.349204   0.08530065 0.06678036 0.27650674]
```

*Figure 18. Use of Diagonal Regularization of Covariance Matrix to Avoid Singularity Issues (K=6, reg_factor = 0.001, Phoneme 2)*

## Code for Task 5:

```
import numpy as np
import os
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
from print_values import *
from plot_data_all_phonemes import *
from plot_data_3D import *
import random
from sklearn.preprocessing import normalize
from get_predictions import *
from plot_gaussians import *

# File that contains the data
data_npy_file = 'data/PB_data.npy'

# Loading data from .npy file
data = np.load(data_npy_file, allow_pickle=True)
data = np.ndarray.tolist(data)

# Make a folder to save the figures
figures_folder = os.path.join(os.getcwd(), 'figures')
if not os.path.exists(figures_folder):
    os.makedirs(figures_folder, exist_ok=True)

# Array that contains the phoneme ID (1-10) of each sample
phoneme_id = data['phoneme_id']
# frequencies f1 and f2
f1 = data['f1']
f2 = data['f2']

# Initialize array containing f1, f2 & f1+f2, of all phonemes.
X_full = np.zeros((len(f1), 3))
# ########################################
# Write your code here
# Store f1 in the first column of X_full, f2 in the second column of X_full and f1+f2 in the third column of X_full
```

```
X_full[:, 0] = f1
X_full[:, 1] = f2
X_full[:, 2] = f1 + f2
# ########################################
X_full = X_full.astype(np.float32)

# We will train a GMM with k components, on a selected phoneme id which is stored in variable "p_id"

# id of the phoneme that will be used (e.g. 1, or 2)
p_id = 1 #  also tried p_id = 2
# number of GMM components
k = 3 # also tried k=6
# ########################################
# Write your code here

# Create an array named "X_phoneme", containing only samples that belong to the chosen phoneme.
# The shape of X_phoneme will be two-dimensional. Each row will represent a sample of the dataset,
# and each column will represent a feature (e.g. f1 or f2 or f1+f2)
# Fill X_phoneme with the samples of X_full that belong to the chosen phoneme
# To fill X_phoneme, you can leverage the phoneme_id array, that contains the ID of each sample of X_full
X_phoneme = X_full[phoneme_id == p_id, :]
# ########################################
# ###############################################
# Plot array containing the chosen phoneme

# Create a figure and a subplot
fig = plt.figure()
ax1 = plt.axes(projection='3d')

title_string = 'Phoneme {}'.format(p_id)
# plot the samples of the dataset, belonging to the chosen phoneme (f1 & f2, phoneme 1 or 2)
plot_data_3D(X=X_phoneme, title_string=title_string, ax=ax1)
# save the plotted points of phoneme 1 as a figure
plot_filename = os.path.join(os.getcwd(), 'figures', 'dataset_3D_phoneme_{}.png'.format(p_id))
plt.savefig(plot_filename)

################################################
# Train a GMM with k components, on the chosen phoneme
# as dataset X, we will use only the samples of the chosen phoneme
X = X_phoneme.copy()

# get number of samples
N = X.shape[0]
# get dimensionality of our dataset
D = X.shape[1]

# common practice : GMM weights initially set as 1/k
p = np.ones(k) / k
# GMM means are picked randomly from data samples
random_indices = np.floor(N * np.random.rand(k))
random_indices = random_indices.astype(int)
mu = X[random_indices, :]  # shape kxD
# covariance matrices
s = np.zeros((k, D, D))  # shape kxDxD
# number of iterations for the EM algorithm
n_iter = 150

# initialize covariances
for i in range(k):
    cov_matrix = np.cov(X.transpose())
    # initially set to fraction of data covariance
    s[i, :, :] = cov_matrix / k

# Initialize array Z that will get the predictions of each Gaussian on each sample
Z = np.zeros((N, k))  # shape Nxk

# ###############################
# run Expectation Maximization algorithm for n_iter iterations
for t in range(n_iter):
    # print('****************************************')
    print('Iteration {:03}/{:03}'.format(t + 1, n_iter))

    # Do the E-step
    Z = get_predictions(mu, s, p, X)
    Z = normalize(Z, axis=1, norm='l1')
    # Do the M-step:
    for i in range(k):
        mu[i, :] = np.matmul(X.transpose(), Z[:, i]) / np.sum(Z[:, i])

        ###################################################
        # We will fit Gaussian's with full covariance matrices:
        mu_i = mu[i, :]
        mu_i = np.expand_dims(mu_i, axis=1)
        mu_i_repeated = np.repeat(mu_i, N, axis=1)

        term_1 = X.transpose() - mu_i_repeated
        term_2 = np.repeat(np.expand_dims(Z[:, i], axis=1), D, axis=1) * term_1.transpose()
        s[i, :, :] = np.matmul(term_1, term_2) / np.sum(Z[:, i])
        #########################################
        # Write your code here
        # Suggest ways of overcoming the singularity
        s[i, :, :] += 0.001 * np.identity(D)
        # #########################################
        p[i] = np.mean(Z[:, i])
    ax1.clear()
    # plot the samples of the dataset, belonging to the chosen phoneme (f1, f2, f1+f2 | phoneme 1 or 2)
    plot_data_3D(X=X, title_string=title_string, ax=ax1)
    # Plot gaussian's after each iteration
    plot_gaussians(ax1, 2 * s, mu)
print('\nFinished.\n')
print('Implemented GMM | Mean values')
for i in range(k):
    print(mu[i])
print('Implemented GMM | Covariances')
for i in range(k):
    print(s[i, :, :])
print('Implemented GMM | Weights')
print(p)
print('')
# enter non-interactive mode of matplotlib, to keep figures open
plt.ioff()
plt.show()
```