

# Machine Learning Assignment 2: Clustering and MoG

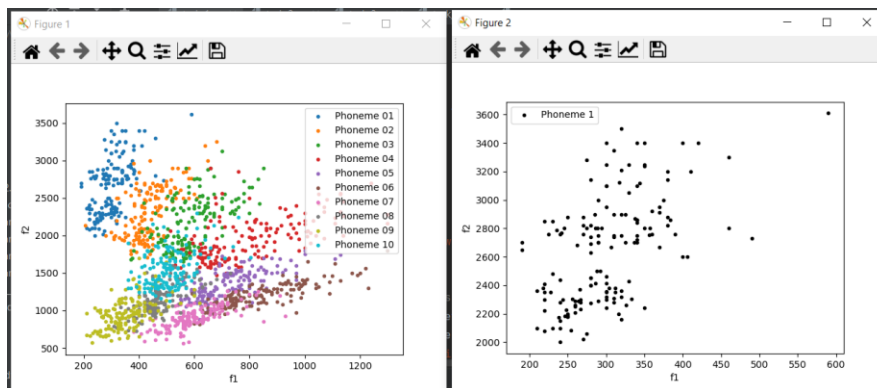
## Q1: Task 1

Load the dataset to your workspace. We will only use the dataset for F1 and F2, arranged into a 2D matrix where the first column will be F1 and the second column will be F2. Produce a plot of F1 against F2. (You should be able to spot some clusters already in this scatter plot.).

Include in your report the corresponding lines of your code and the plot. [20 points]

The dataset is loaded into the workspace for this task, and the phonemes are displayed on a 2D plot with axes formed by the fundamental frequencies F1 and F2. The following are the results of task 1.py. Three clusters are visible in the figure on the right.

### #Outputs:



The below statistic is also obtained:

```
task_1
C:\Users\jahnv\anaconda3\python.exe "C:/Users/jahnv/OneDrive/Desktop/ML/Assignment 2/assign_2/assgn_2/task_1.py"
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
[240. 280. 390. ... 500. 740. 660.]
[2280. 2400. 2030. ... 1140. 1850. 1830.]
```

### #CODE:

```
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 *

# 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
print(X_full[:, 0])
print(X_full[:, 1])

#####/
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 = 2

#####
# Write your code here

# Create an array named "X_phoneme_1", containing only samples that belong to the chosen phoneme.

X_phoneme_1 = np.zeros((np.sum(phoneme_id==1), 2))
# 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==1), 2))
# X_phoneme = ...
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
fig, 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()
```

## **Q2: Task 2**

**Train the data for phonemes 1 and 2 with MoGs. You are provided with python files task 2.py and gaussians.py. Specifically, you are required to:**

- 1. Look at the task 2.py code and understand what it is calculating. Pay particular attention to the initialisation of the means and covariances (also note that it is only estimating diagonal covariances).**
  - 2. Generate a dataset X phoneme 1 that contains only the F1 and F2 for the first phoneme.**
  - 3. Run task 2.py on the dataset using K=3 Gaussians (run the code a number of times and note the differences.)**
- Save your MoG model: this should comprise the variables mu, s and p.**
- 4. Run task 2.py on the dataset using K=6**
  - 5. Repeat steps 2-4 for the second phoneme.**

**Include in your report the lines of code you wrote, and results that illustrate the learnt models. [20 points]**

This section will make use of task2.py, which was provided in the lab assignment. This script is used four times to build MoG models for phonemes 1 and 2 with K=3 and K=6, respectively. To compare results, experiments were repeated several times. The Expectation Maximization algorithm is used to train the MoG model. Value K denotes the number of Gaussian fits to the dataset. For phonemes 1 and 2, separate MoG models are trained.

The following algorithm demonstrates Expectation Maximization, which is used in the MoG model. The EM algorithm first determines the weights and then selects GMM means at random from the dataset. The shape of the mean matrix is k times D, whereas the shape of the covariance tensor is k by D times D, where k is the number of Gaussian in the mixture model and D is the dimensionality of the dataset. The dataset X is used to initialise the covariance matrix for each Gaussian. GMM weights are uniformly initialised. Because there are k Gaussians and N samples, a predictions matrix Z of size N by k is formed. Following this, the Expectation Maximization algorithm is used for n\_iter times.

Predictions are fetched for the expectation step(E-step) using the mean, covariance, GMM weights, and the dataset. This is accomplished through the use of the get predictions.py method. The following formula returns predictions Z for each Gaussian. Normalization is required to ensure that prediction vector Z is consistent with the probability measure. It is important to note that the following equation must be calculated for all samples:

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

In the maximisation step (M-step) , a weighted log-likelihood function is optimised for the mixture model for each of the Gaussians in the GMM model: new means, covariances, and mixture weights for the corresponding Gaussians are chosen. The logarithm operation is used to simplify the exponential term in the Gaussian distribution, allowing for easier optimization. Taking the derivative of the weighted log-likelihood function for

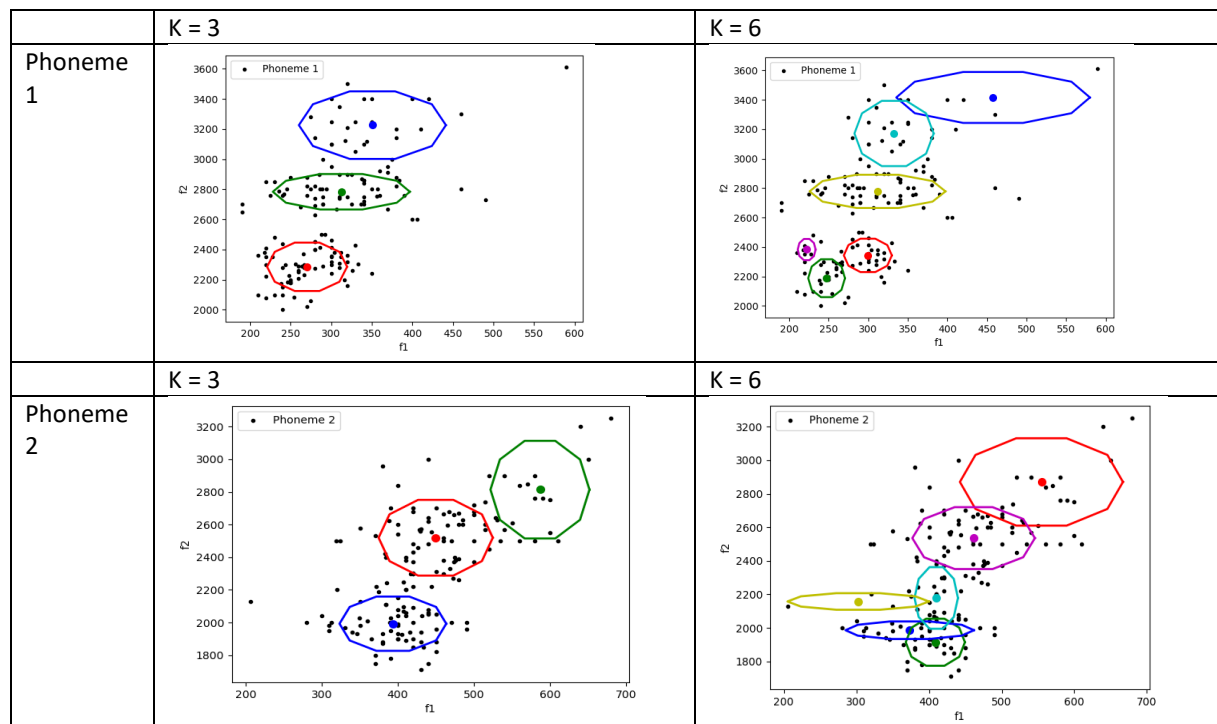
mean, standard deviation, and GMM weights yields the equations in the M-step, which are written in the code above. The update equations at (i+1)'th iteration are as follows:

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

It should be noted that only diagonal entries are computed during the maximisation step. Diagonal entries in the covariance matrix correspond to variances of individual dimensions. "res 1" calculates the variances for F1 and F2 as a vector, which is then converted to a covariance matrix for use in the GMM. The phoneme 1 and 2 outputs for K = 3 and K = 6 are shown below. All tests were carried out for a total of 100 EM iterations. Because the initial mean values are chosen at random, the Gaussian centres and label colours change when the experiment is restarted. After 100 iterations, all independent experiments converge to similar layouts with very similar outcomes.

## #Outputs for Task 2:

### A) Gaussian Models are as shown below:



### B) Mean, Variance and GMM Weights for Phoneme 1 and Phoneme 2:

	K = 3	K = 6
Phoneme 1	<pre> 105 106 Implemented GMM   Mean values 107 [ 270.3952 2285.4653] 108 [ 312.59125 2783.898 ] 109 [ 350.8446 3226.3394] 110 111 Implemented GMM   Covariances 112 [[ 1213.73843494 0. ] 113 [ 0. 14278.4202995 ]] 114 [[3562.59743765 0. ] 115 [ 0. 7657.84897245]] 116 [[ 4102.875375 0. ] 117 [ 0. 27829.54221422]] 118 119 Implemented GMM   Weights 120 [0.43514434 0.38099528 0.18386038] 121 </pre> <p>Page 3 of 3</p>	<pre> 114 Implemented GMM   Covariances 115 [[ 460.06868572 0. ] 116 [ 0. 7115.95326276]] 117 [[ 269.03919204 0. ] 118 [ 0. 9147.77853499]] 119 [[ 7474.67888023 0. ] 120 [ 0. 16556.37235094]] 121 [[ 1251.73391388 0. ] </pre> <p>Page 3 of 4</p> <hr/> <pre> File - task_2 122 [ 0. 27163.5923426 ] 123 [[ 64.00260235 0. ] 124 [ 0. 2870.80419161]] 125 [[3684.10188282 0. ] 126 [ 0. 7051.58873667]] 127 128 Implemented GMM   Weights 129 [0.21249032 0.17626387 0.02607654 0.17220193 0. 0.4534794 0.3676194 ] </pre>
	K = 3	K = 6

Phoneme 2	106 Implemented GMM   Mean values 107 [ 449.67615 2519.6865 ] 108 [ 586.56354 2814.1814 ] 109 [ 393.45288 1993.0826 ] 110 111 Implemented GMM   Covariances 112 [[ 2809.33998756 0. ]] 113 [ 0. 29720.28652606]] 114 [[ 2111.52137597 0. ]] 115 [ 0. 49424.41053787]] 116 [[ 2454.90873186 0. ]] 117 [ 0. 15263.72205429]] 118 119 Implemented GMM   Weights 120 [0.46995344 0.09487908 0.43516748] 121 Page 3 of 3	106 Implemented GMM   Mean values 107 [ 554.76935 2870.7134 ] 108 [ 408.69656 1914.5243 ] 109 [ 373.4518 1986.1808] 110 [ 409.94376 2178.7542 ] 111 [ 461.4054 2535.671 ] 112 [ 302.45483 2157.7102 ] 113 114 Implemented GMM   Covariances 115 [[ 6300.92134696 0. ]] 116 [ 0. 37530.16060906]] 117 [[ 828.8816996 0. ]] 118 [ 0. 10876.03844741]] 119 [[3908.1582197 0. ]] 120 [ 0. 1514.23514878]] 121 [[ 446.97218189 0. ]] Page 3 of 4	
		File - task_2 122 [ 0. 18668.35159018]] 123 [[ 3560.46642424 0. ]] 124 [ 0. 18839.07048577]] 125 [[4739.04950988 0. ]] 126 [ 0. 1348.79576843]] 127 128 Implemented GMM   Weights 129 [0.09711308 0.19006832 0.1214562 0.15954052 0.41065881 0.02116306] 	

### #CODE:

```

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)
# 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
k = 6 (the k value was altered for k = 3 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 (the p_id value was altered for phoneme 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 = np.zeros((np.sum(phoneme_id==p_id), 2))

# X_phoneme = ...
X_phoneme = X_full[phoneme_id==p_id,:]
print(X_phoneme[0])
#####/

# 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 Gaussians 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 gaussians 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 = {}
GMM_parameters['mu'] = mu
GMM_parameters['s'] = s

```

```
GMM_parameters['p'] = p
```

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

### Q3: Task 3

Use the 2 MoGs (K=3) learnt in task 2 to build a classifier to discriminate between phonemes 1 and 2. Classify using the Maximum Likelihood (ML) criterion (feel free to hack parts from the MoG code in task 2.py so that you calculate the likelihood of a data vector for each of the two MoG models) and calculate the misclassification error. Remember that a classification under the ML compares  $p(x; \theta_1)$ , where  $\theta_1$  are the parameters of the MoG learnt for the first phoneme, with  $p(x; \theta_2)$ , where  $\theta_2$  are the parameters of the MoG learnt for the second phoneme.

Repeat this for K = 6 and compare the results.

Include in your report the lines of the code that you wrote, explanations of what the code does and comment on the differences on the classification performance [20 points]

For both phoneme 1 and phoneme 2, the models from task 2 will be utilized in the task 3. The code segment below can be used to load the models into memory. Note that this is used separately for phonemes 1 and 2. The GMM training is carried out using model parameters and `get_predictions()` method that were explained in detail in Task 2. The method constructs GMM and utilizes the provided phonemes to give predictions which is also the expectation step of expectation maximization algorithm.

The models are trained separately for both Phoneme 1 and Phoneme 2. The dataset contains elements of both phoneme 1 and phoneme 2. In Z, each model contains likelihood of data points in a GMM cluster of given parameters. The outputs of Z of each model is used for the class classification of datasets. The necessary code for this has been highlighted by 'bold' in the below #CODE section for Task 3.

The likelihoods are added (pred1 and pred2) after returning of predictions on all clusters per model. The comparison of likelihoods – Z1 and Z2 is beneficial in achieving the classification from the models that have been previously trained on the Phoneme 1 and 2 respectively. The outputs have been shown in the Accuracy and Mis-classification error individually for both K = 3 and 6.

Accuracy and Mis-classification									
K = 3					K = 6				
File - task_3 (1)									
1	C:\Users\jahnv\anaconda3\python.exe "C:/Users/jahnv/OneDrive/Desktop/ML/Assignment 2/assign_2/assgn_2/task_3.py"								
2		precision	recall	f1-score	support				
3									
4	phoneme 1	0.94	0.96	0.95	152				
5	phoneme 2	0.96	0.94	0.95	152				
6									
7	accuracy			0.95	304				
8	macro avg	0.95	0.95	0.95	304				
9	weighted avg	0.95	0.95	0.95	304				
10									
11	confusion matrix:								
12	[[146 6]								
13	[ 9 143]]								
14	Accuracy using GMMs with 3 components: 95.07%								
15	Mis-classification error using GMMs with 3 components : 4.93%								
16									

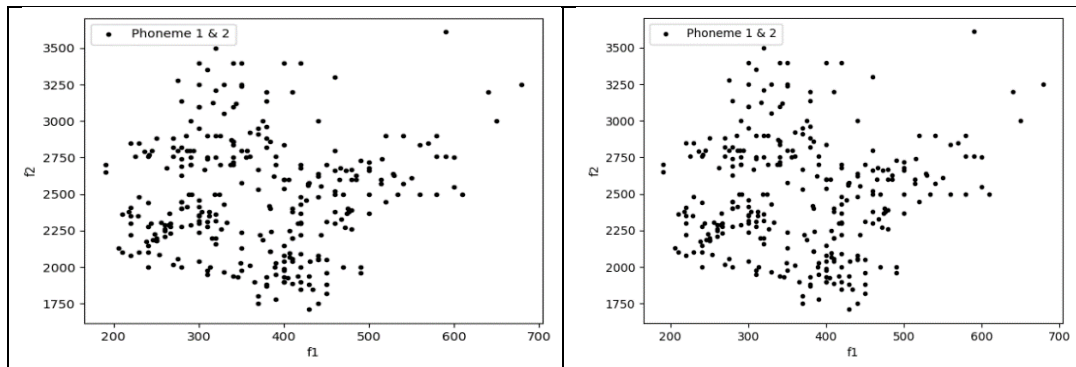
File - task_3 (1)									
1	C:\Users\jahnv\anaconda3\python.exe "C:/Users/jahnv/OneDrive/Desktop/ML/Assignment 2/assign_2/assgn_2/task_3.py"								
2		precision	recall	f1-score	support				
3									
4	phoneme 1	0.95	0.97	0.96	152				
5	phoneme 2	0.97	0.95	0.96	152				
6									
7	accuracy			0.96	304				
8	macro avg	0.96	0.96	0.96	304				
9	weighted avg	0.96	0.96	0.96	304				
10									
11	confusion matrix:								
12	[[147 5]								
13	[ 7 145]]								
14	Accuracy using GMMs with 6 components: 96.05%								
15	Mis-classification error using GMMs with 6 components : 3.95%								
16									

As it can be seen from the outputs that K = 6 gave better results as compared to K = 3 for accuracy. It can also be observed that the K = 3 is faster at training models than K = 6.

### #Output for Task 3:

K = 3	K = 6
Scatter plots for phonemes 1 and 2	





### #CODE:

```
import numpy as np
import os
import time
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 *
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn import metrics

# 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)

def load_data(k):
    # File that contain trained model
    data_npy_phoneme_01 = 'data/GMM_params_phoneme_01_k_0' + str(k) + '.npy'
    data_npy_phoneme_02 = 'data/GMM_params_phoneme_02_k_0' + str(k) + '.npy'

    # Loading data from .npy file
    model_phoneme_01 = np.ndarray.tolist(np.load(data_npy_phoneme_01, allow_pickle=True))
    model_phoneme_02 = np.ndarray.tolist(np.load(data_npy_phoneme_02, allow_pickle=True))
    return model_phoneme_01, model_phoneme_02

def get_pred(X, k, model1_weights, model2_weights):
    predClass = []
    N = X.shape[0]
    Z01 = np.zeros((N, k))
    Z02 = np.zeros((N, k))

    # get predictions on X from model1
    Z01 = get_predictions(model1_weights['mu'], model1_weights['s'], model1_weights['p'], X)
    #Z01 = normalize(Z01, axis=1, norm='l1')
    Z01 = Z01.astype(np.float32)
    Z01Sum = np.sum(Z01, axis=1)
```

```

# get predictions on X from model2
Z02 = get_predictions(model2_weights['mu'], model2_weights['s'], model2_weights['p'], X)
#Z02 = normalize(Z02, axis=1, norm='l1')
Z02 = Z02.astype(np.float32)
Z02Sum = np.sum(Z02, axis=1)

# if sum of probabilities of any model is less than other mark it
for z1, z2 in zip(Z01Sum, Z02Sum):
    if z1 > z2:
        predClass.append(0.0)
    else:
        predClass.append(1.0)

y_pred = np.array(predClass)
return y_pred

# 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
#k = 6 (the k value was altered for k = 3 as well)
#####
# 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 = np.zeros((np.sum(phoneme_id==2)+np.sum(phoneme_id==1), 2))
X_phonemes_1_2 = np.concatenate((X_full[phoneme_id==1,:],X_full[phoneme_id==2,:]), axis=0)
## X_phonemes_1_2 =
y_true = np.concatenate((np.zeros((np.sum(phoneme_id==1))),np.ones((np.sum(phoneme_id==2))))),axis =0)

#####/

```

```

# 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"

X = X_phonemes_1_2.copy()
# get number of samples
N = X.shape[0]
# get dimensionality of our dataset
D = X.shape[1]

model_phoneme_01, model_phoneme_02 = load_data(k)
y_pred = get_pred(X, k, model_phoneme_01, model_phoneme_02)

# print(y_pred)
accuracy = accuracy_score(y_true, y_pred)

target_names = ['phoneme 1', 'phoneme 2']
report = print(classification_report(y_true, y_pred, target_names = target_names))

confusion = metrics.confusion_matrix(y_true, y_pred)

print("confusion matrix: ")
print(confusion)
#[row, column]
TP = confusion[1, 1]
TN = confusion[0, 0]
FP = confusion[0, 1]
FN = confusion[1, 0]
classification_error = (FP + FN) / float(TP + TN + FP + FN)

#####/

print('Accuracy using GMMs with {} components: {:.2f}%'.format(k, accuracy*100))
print('Mis-classification error using GMMs with {} components: {:.2f}%'.format(k, classification_error*100))
#####
# enter non-interactive mode of matplotlib, to keep figures open
plt.ioff()
plt.show()

```

#### Q4: Task 4:

Create a grid of points that spans the two datasets. Classify each point in the grid using one of your classifiers. That is, create a classification matrix,  $M$ , whose elements are either 1 or 2.  $M(i, j)$  is 1 if the point  $x_1$  is classified as belonging to phoneme 1, and is 2 otherwise.  $x_1$  is a vector whose elements are between the minimum and the maximum value of  $F_1$  for the first two phonemes, and  $x_2$  similarly for  $F_2$ . Display the classification matrix. Include the lines of code in your report, comment them, and display the classification matrix. [20 points]

As seen above, the classification is based on the datapoints in Task 3 and above provided grid boundaries proved to be accurate for both  $K=3$  and  $K=6$  models. For better balancing of computational performance along with accuracy, the  $K=3$  could be the best model with Occam's razor. The Mesh grid is used for generation of grid for classification of points and flattening is implemented for utilization of vector notation and operations. The necessary code has been provided below for linear spaced vector  $f_1$  and  $f_2$  such that  $F_1$  and  $F_2$  can take minimum and maximum values(Phonemes 1 and 2):

```
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()
```

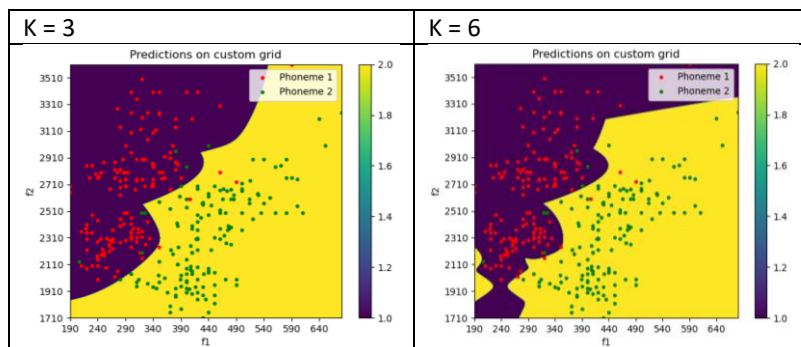
```
samples = np.stack((x, y)).transpose() # fix dimensions
```

#### #Outputs for Task 4:

File - task\_4

```
1 C:\Users\jahnv\anaconda3\python.exe C:/Users/jahnv/
  Downloads/ML_assignment2_190573735/assgn_2/task_4.py
2 f1 range: 190-680 | 490 points
3 f2 range: 1710-3610 | 1900 points
4
```

#### Custom grid predictions for $K=3$ and $K=6$



#### #CODE:

```
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
#k = 6 (the k value was altered for k = 6 as well)
#####
# 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()

```

#### Q5: Task 5:

In the code of task 5.py a MoG with a full covariance matrices is fit to the data. Now, create a new dataset that will contain 3 columns, as follows:

$$X = [F1, F2, F1 + F2] \quad (2)$$

Fit a MoG model to the new data. What is the problem that you observe? Explain why. Suggest ways of overcoming the singularity problem and implement them.

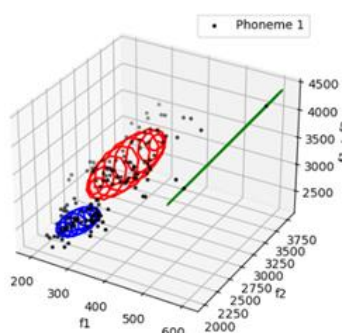
Include the lines of code in your report, and graphs/plots so as to support your observations. [20 points]

The crash occurs because the inverse of the covariance matrix does not exist, so Python must solve the problem in the complex domain. It causes a slew of problems in training, such as the above notation for ignoring the imaginary part of matrices or dividing by zero. The matrix is also singular, according to the statement. If the dataset's attributes were linearly independent, a diagonal covariance matrix would result, which is always invertible.

```
Run task 5
Iteration 048/150
Iteration 049/150
Iteration 050/150
Iteration 051/150
C:\Users\john\OneDrive\Desktop\ML\Assignment 2\assign_2\assign_2\get_predictions.py:29: RuntimeWarning: divide by zero encountered in double_scalars
Z[:,1] = p[1]/(1/np.power(((2*np.pi)**400) + np.abs(s_1_det), 0.5)) * np.exp(-0.5*np.sum(x_s_x, axis=1))
Traceback (most recent call last):
  File "C:\Users\john\OneDrive\Desktop\ML\Assignment 2\assign_2\assign_2\task_5.py", line 117, in <module>
    Z = normalize(Z, axes=1, norm=1)
  File "C:\Users\john\anaconda3\lib\site-packages\sklearn\utils\validation.py", line 63, in inner_f
    return f(*args, **kwargs)
  File "C:\Users\john\anaconda3\lib\site-packages\sklearn\preprocessing\data.py", line 1984, in normalize
    X = check_array(X, accept_sparse=sparse_format, copy=copy,
  File "C:\Users\john\anaconda3\lib\site-packages\sklearn\utils\validation.py", line 63, in inner_f
    return f(*args, **kwargs)
  File "C:\Users\john\anaconda3\lib\site-packages\sklearn\utils\validation.py", line 663, in check_array
    assert_all_finite(array,
  File "C:\Users\john\anaconda3\lib\site-packages\sklearn\utils\validation.py", line 103, in _assert_all_finite
    raise ValueError("Input contains NaN, infinity or a value too large for dtype('float64').")
ValueError: Input contains NaN, infinity or a value too large for dtype('float64').
Process finished with exit code 1
```

If the diagonal covariance enforcement approach from Task 2 is used in Task 5, the singularity problem is solved, and the results for K=3 are shown in below table #Output → Diagonal Matrix. Similar results can be seen when K=6 is used.

The covariance matrix is constrained in this implementation, with all non-diagonal entries set to zero. This implementation avoids the singularity problem because such diagonal matrices are always invertible. F1 and F2 are also linearly independent of each other, based on this diagonal matrix. This is a strong assumption, but it holds up well. When using the entire covariance matrix (as in Task 5) in Task 2, the covariance matrix becomes singular in some runs, causing the GM model to fail. In many cases, training stands the test of time. A new feature vector F1+F2 is added to the dataset in Task 5. Because the third feature is linearly dependent on the first two, it will cause singularity problems, as shown below:



```
155 Implemented GMM | Mean values
156 [ 315.79132 2877.855 3193.6462 ]
157 [ 540.0022 3170.019 3710.0212 ]
158 [ 270.35 2271.1873 2541.537 ]
159 Implemented GMM | Covariances
160 [[ 3023.78059537 5190.09346255 8213.87305792]
161 [ 5190.09346255 70417.08345535 75607.9759179 ]
162 [ 8213.87305792 75607.9759179 83821.84977581]]
Page 4 of 5

File: task_5
163 [[ 2500.00099528 21999.99995845 24499.99995373]
164 [ 21999.99995845 193600.00063639 215599.99995284]
165 [ 24499.99995373 215599.99995284 240100.00054657]]
166 [[ 1187.90005509 1264.6448268 2452.54388189]
167 [ 1264.6448268 13808.5298177 14273.17364448]
168 [ 2452.54388189 14273.17364448 16725.71852639]]
169 Implemented GMM | Weights
170 [0.60221087 0.01315732 0.38463181]
```

Gaussian clusters do not fit the dataset very well in the figure provided in #Output → Diagonal Matrix (as in Green one). F1, F2, and F1+F2 appear to be linearly independent based on the covariance matrix, but this is not the case. F1+F2 is created artificially by combining F1 and F2. Also note that all Gaussians in the figures above are parallel to the x and y axes and have no skewness in the Z direction. This is due to the assumption that all three axes are independent. This is incorrect; the model does not account for this dependency. The results will be very similar to the results in Task 2 if the third dimension is ignored and projection to the F1-F2 axis is used. Regularizing covariance by adding regularization to the covariance matrix is another way to deal with singularity. This method is based on the method used in Task 5.

The below provided implementation is another approach to deal with the singularity for regularization of covariance matrix in the Task 5

- # Write your code here
- # Suggest ways of overcoming the singularity



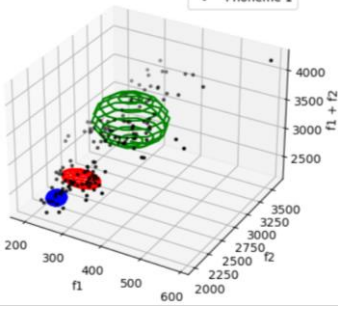
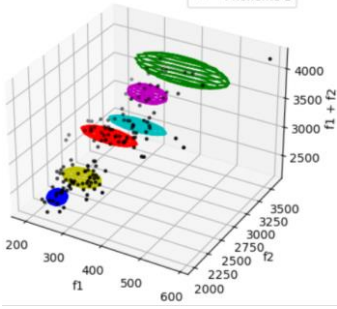
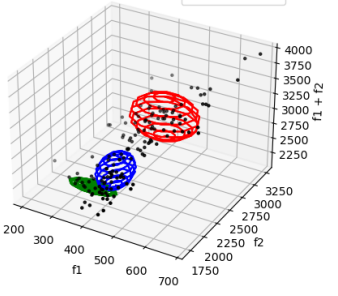
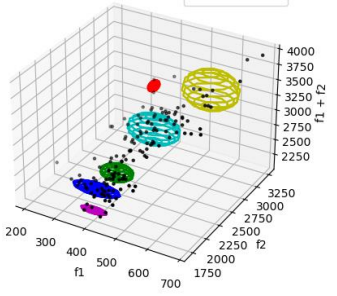
$s[i, :, :] += 0.001 * \text{np.identity}(D)$

With the addition of diagonal matrix the covariance matrix is regularized which has the same dimension as the original covariance matrix. In the diagonal matrix an identity matrix is used. A small amount of 0.001 is used for the diagonal input factors. Different regularization parameters are also possible for each of the attributes. In the EM method even if some of the variances are zero; this little term ensures that all diagonal entries are not zero, so an inversion could be made. After implementing this term, the following results were obtained:

Note the differences in Figures 11, 12 and 13, 14. This time the Gaussian functions are also distributed along the linearly dependent attribute  $F1 + F2$  and the covariance matrices are no longer diagonal. Training on 150 iterations never gives a complex domain answer; divide by zero or other errors related to the singularity. Without the regularization, the code never completed execution. Since a small parameter is always forced on the diagonal entries, the variance of individual attributes was never reached to zero; hence the inverse of the covariance matrix existed.

This is also implemented on Phoneme 2 and similar results were obtained as shown in [#Output](#) where the Phoneme 2 section is displayed.

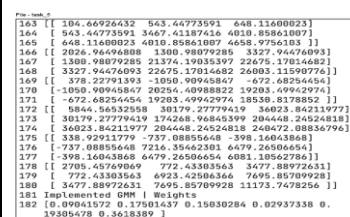
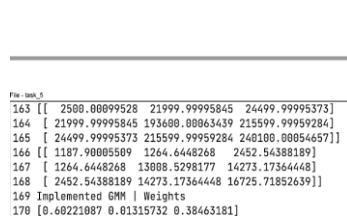
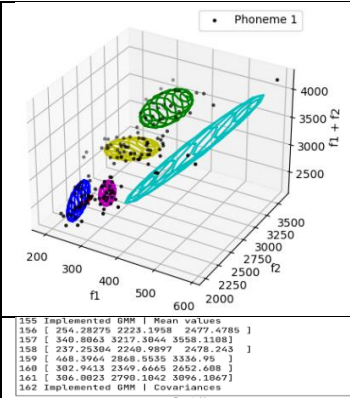
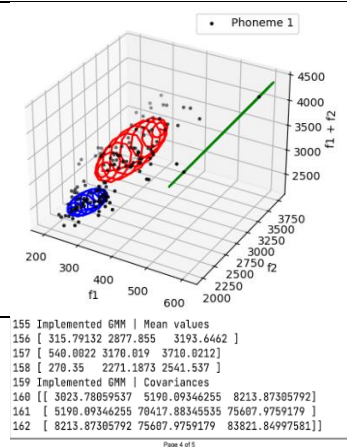
#### #Output for Task 5:

	K = 3	K = 6
<b>Diagonal Matrix</b>		
<b>Phoneme 1</b>	 <pre> 155 Implemented GMM   Mean values 156 [ 288.27118 2337.9487 2618.22 ] 157 [ 324.63422 2923.4524 3248.0867 ] 158 [ 244.42995 2140.2112 2384.6401 ] 159 Implemented GMM   Covariances 160 [[1228.94661893 0. 0. ] 161 [ 0. 5948.5372059 0. ] 162 [ 0. 0. 6289.55377898]] Page 4 of 5 </pre> <pre> File: task_5 163 [[ 4037.13645671 0. 0. ] 164 [ 0. 58454.81413015 0. ] 165 [ 0. 0. 73032.17116943]] 166 [[ 270.11281676 0. 0. ] 167 [ 0. 4989.1807104 0. ] 168 [ 0. 0. 5188.11906304]] 169 Implemented GMM   Weights 170 [0.30777733 0.57106927 0.1211534 ] </pre>	 <pre> 155 Implemented GMM   Mean values 156 [ 292.618 2740.9648 3033.5888 ] 157 [ 388.69406 3415.734 3804.428 ] 158 [ 244.45846 2140.6216 2385.08 ] 159 [ 349.46836 2870.9084 3220.3567 ] 160 [ 333.50128 3163.198 3496.6992 ] 161 [ 288.4988 2540.517 2621.0159 ] 162 Implemented GMM   Covariances Page 4 of 5 </pre> <pre> File: task_5 163 [[2754.80238554 0. 0. ] 164 [ 0. 4190.18457086 0. ] 165 [ 0. 0. 4236.30461078]] 166 [[ 7681.61756384 0. 0. ] 167 [ 0. 7649.43670163 0. ] 168 [ 0. 0. 22552.86717274]] 169 [ 270.14972382 0. 0. ] 170 [ 0. 5022.7849086 0. ] 171 [ 0. 5227.2128164 0. ] 172 [[2865.20983759 0. 0. ] 173 [ 0. 3782.15825638 0. ] 174 [ 0. 0. 2461.49256365]] 175 [[1312.33115936 0. 0. ] 176 [ 0. 6459.18477752 0. ] 177 [ 0. 7183.37918803]] 178 [[1215.70975875 0. 0. ] 179 [ 0. 6222.79813158 0. ] 180 [ 0. 0. 6527.87812658]] 181 Implemented GMM   Weights 182 [0.2532432 0.0590368 0.1215731 0.14130336 0. 11228151 0.31264206] </pre>
<b>Phoneme 2</b>		

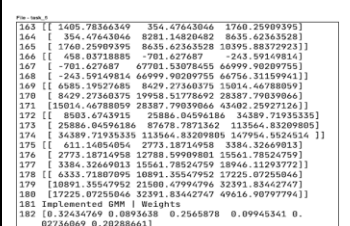
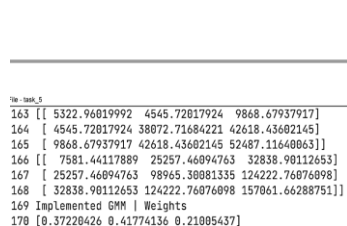
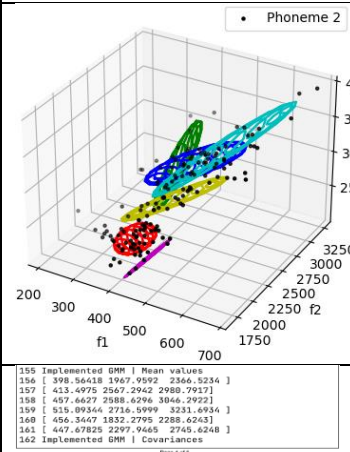
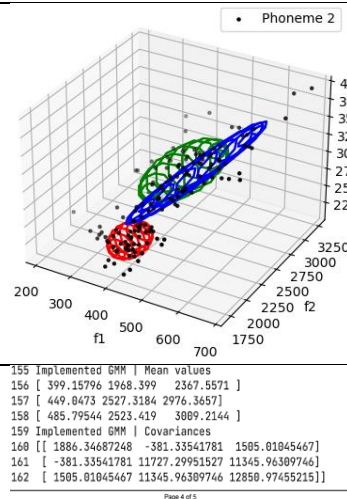
155 Implemented GMM   Mean values 156 [ 551.396 2897.7273 3449.1233 ] 157 [ 457.17286 2506.9622 2964.1343 ] 158 [ 393.32688 1995.9725 2389.2986 ] 159 Implemented GMM   Covariances 160 [ [ 6674.43112856 0. 0. ] 161 [ 0. 25636.87826566 0. ] 162 [ 0. 0. 48844.33964585 ] ] Page 4 of 5	155 Implemented GMM   Mean values 156 [ 399.9927 2900.385 3290.4854 ] 157 [ 412.2961 2100.186 2512.482 ] 158 [ 374.76138 1940.801 2314.7625 ] 159 [ 438.42374 2511.3518 2964.7756 ] 160 [ 488.08293 1757.9937 2157.9966 ] 161 [ 574.0225 2994.4738 3478.4963 ] 162 Implemented GMM   Covariances Page 4 of 5
---	--

## Full Covariance matrix with singularity = 0.001

### Phoneme 1



### Phoneme 2



## #CODE:

```
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
p_id = 2
# number of GMM components
k = 3
#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 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()

# 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()

```