**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:**

**1. MoG Modelling Using the EM Algorithm 3**

**1.1 Task 1 3**

**1.1.1 Code for Task 1 4**

**1.2 Task 2 7**

**1.2.1 Code for Task 2 11**

**1.3 Task 3 16**

**1.3.1 Code for Task 3 18**

**1.4 Task 4 21**

**1.4.1 Code for Task 4 23**

**1.5 Task 5 28**

**1.5.1 Code for Task 5 34**

**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.

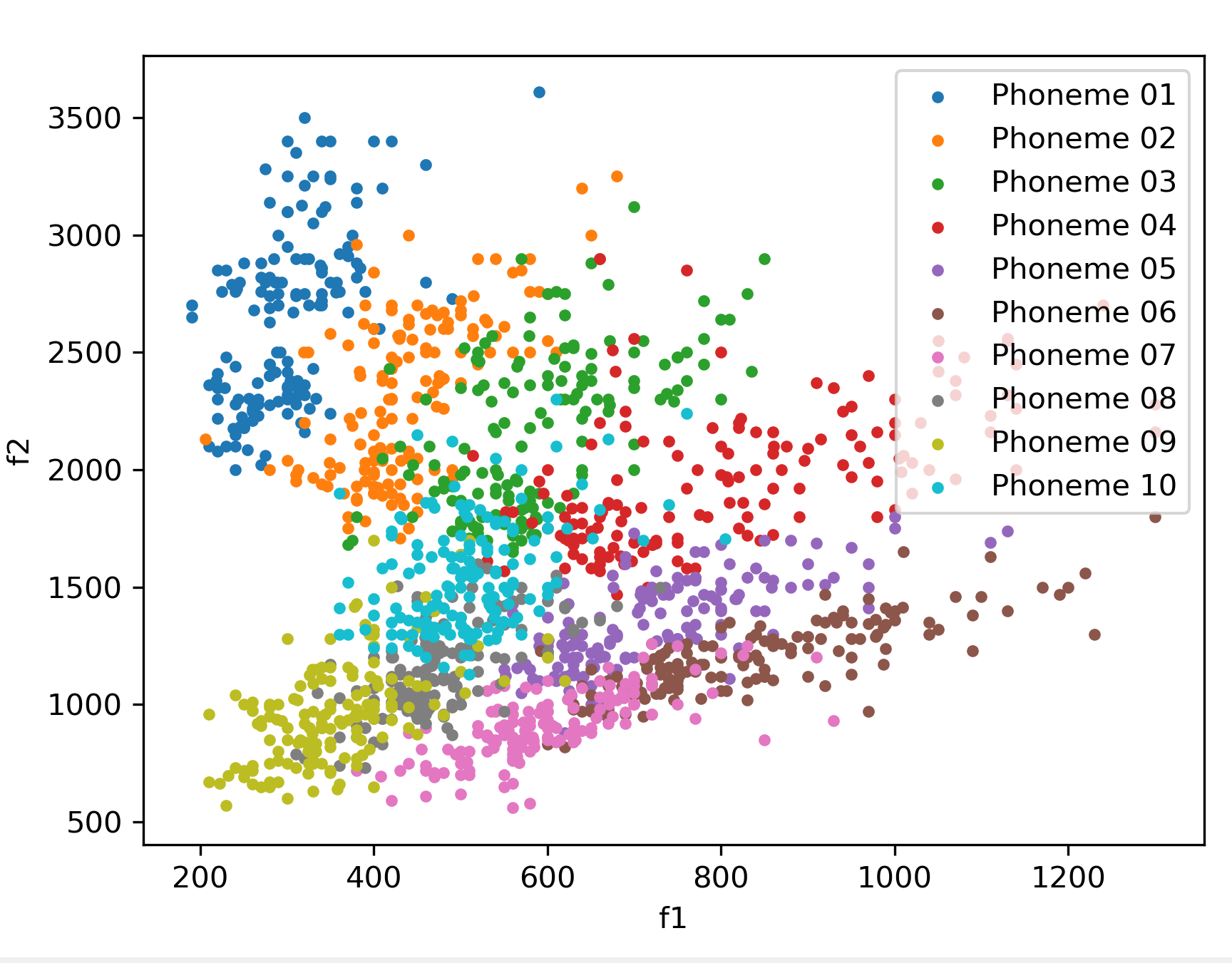
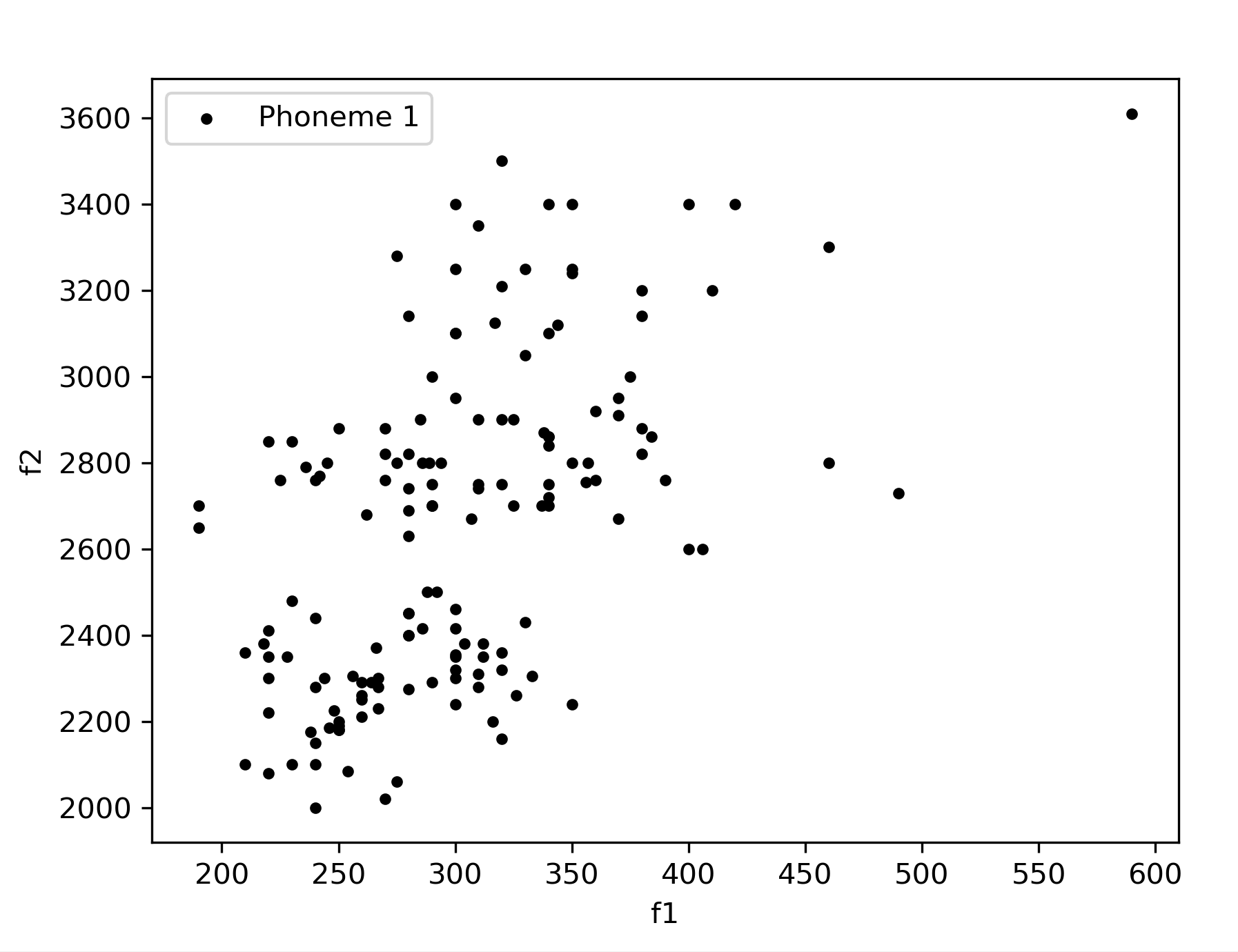


Figure 1. 2D Plot of Phoneme 1 (Left) and All Phonemes (Right) using fundamental frequencies F1 and F2

Additionally, the following statistic is also provided:

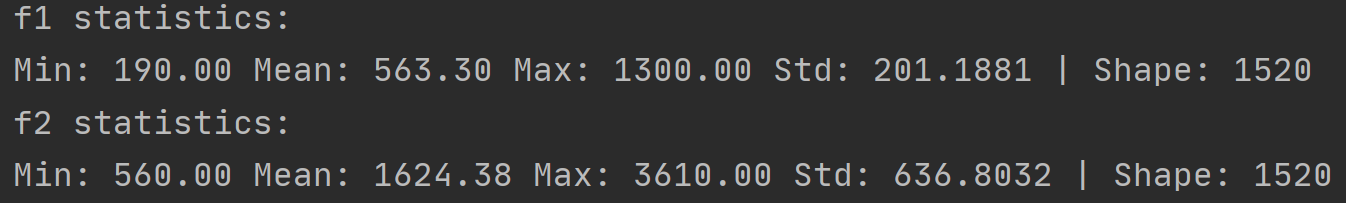


Figure 2. Task 1 fundamental frequency statistics for Phoneme 1

**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:

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:

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.

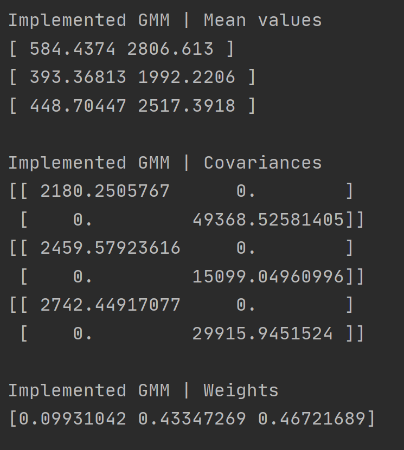
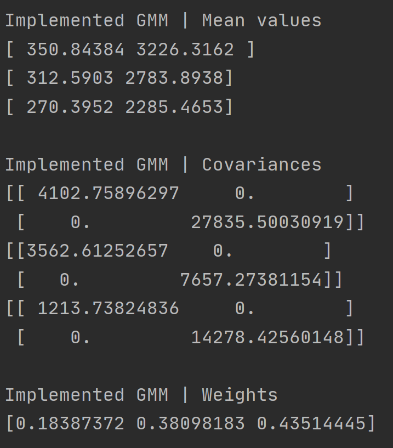


Figure 3. Mean, Variance and GMM Weights for Phoneme 1 (Left) and Phoneme 2 (Right) - K=3

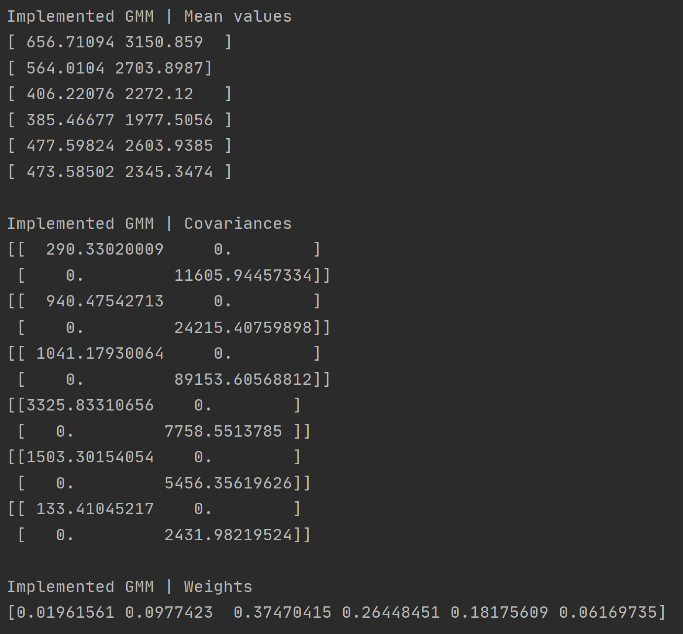
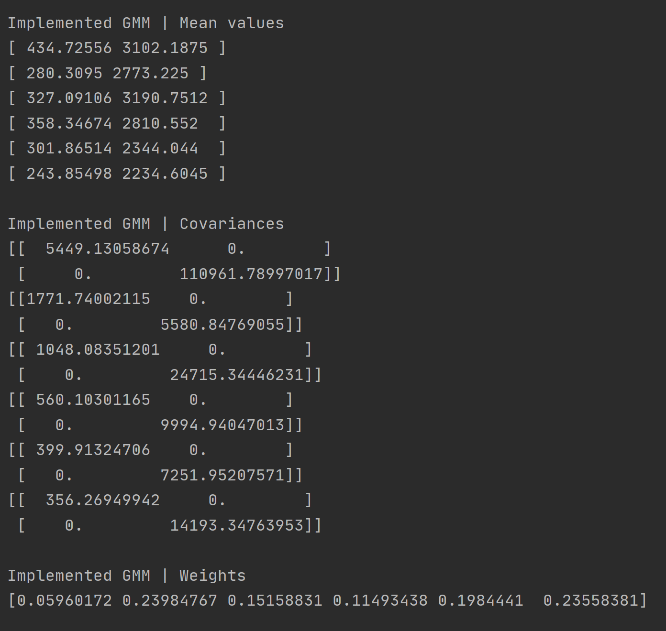


Figure 4. Mean, Variance and GMM Weights for Phoneme 1 (Left) and Phoneme 2 (Right) - K=6

The final Gaussian Models are given below:

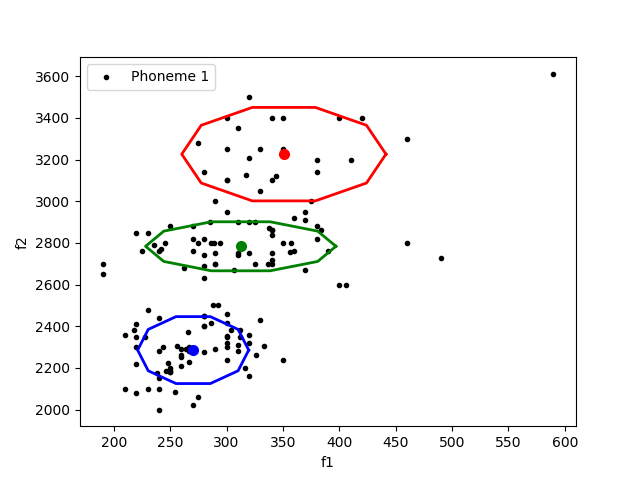


Figure 5. Generated Gaussians Phoneme 1: K =3 (Left) and K=6 (Right)

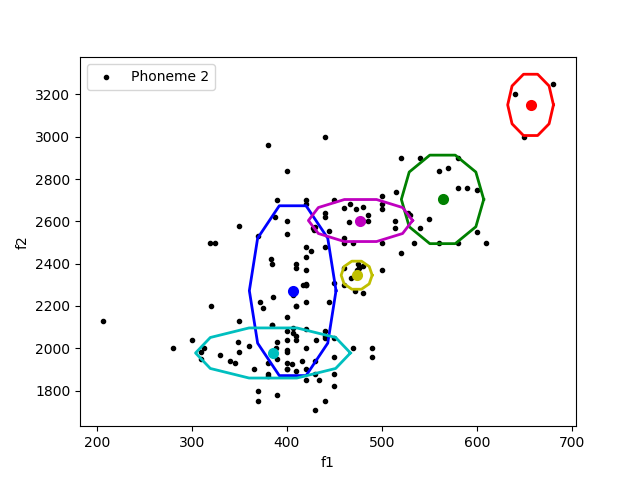
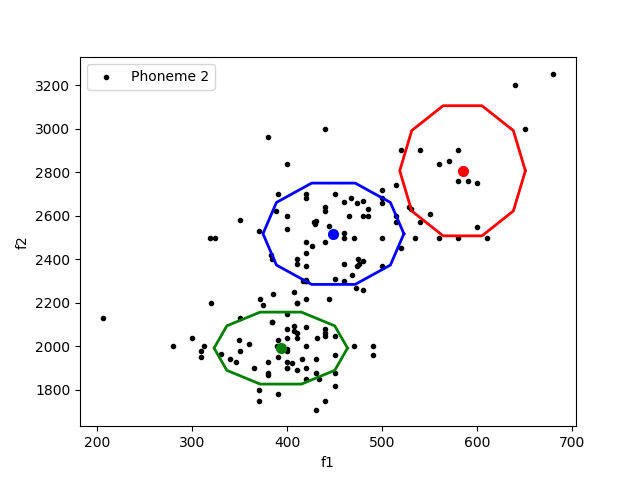


Figure 6. Generated Gaussians Phoneme 2: K =3 (Left) and K=6 (Right)

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:

Chart, scatter chart

Description automatically generated

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.

****

****

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:

Chart, scatter chart

Description automatically generatedChart

Description automatically generated

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:

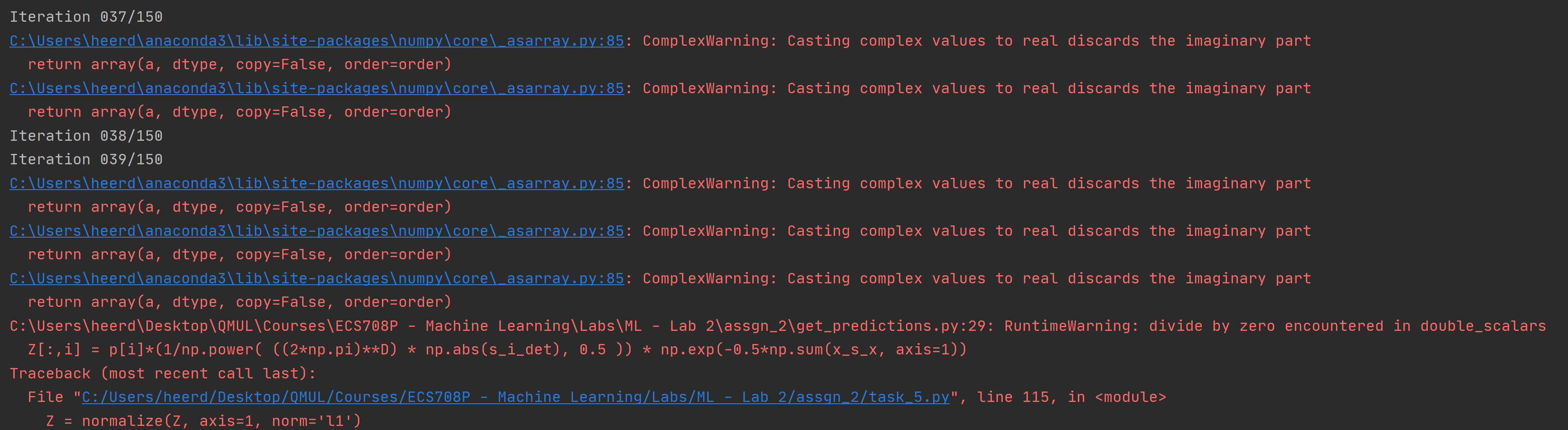


Figure 10. Problems in training due to singularity. Crashed.

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.

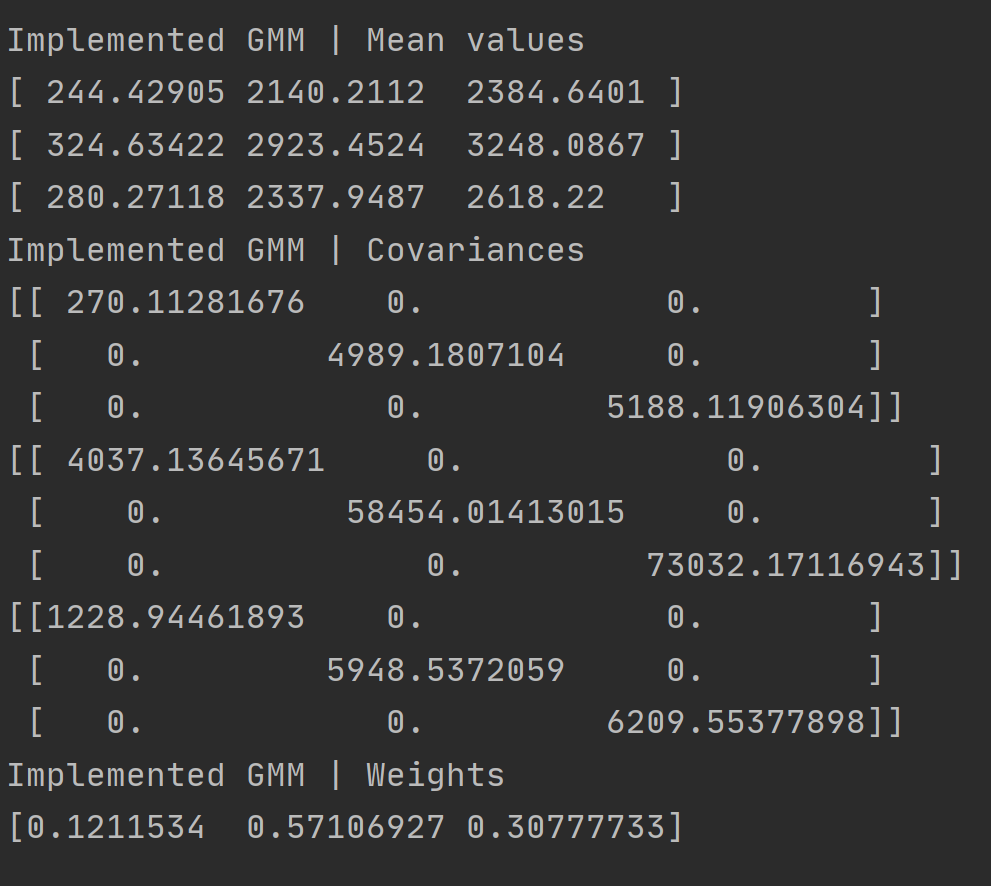
 

Figure 11. Use of Diagonal Entries of Covariance Matrix to Avoid Singularity Issues (K=3, Phoneme 1)

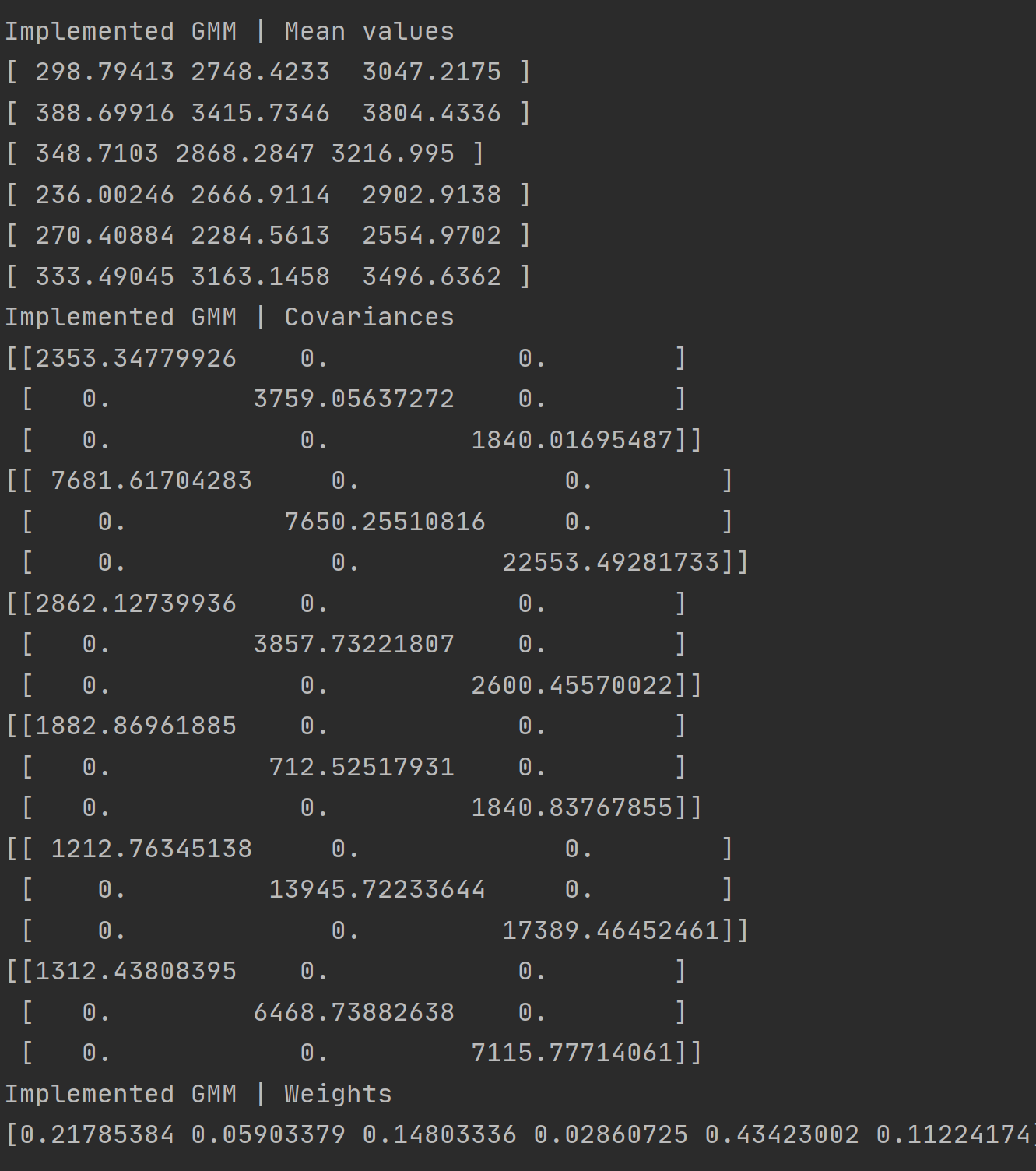
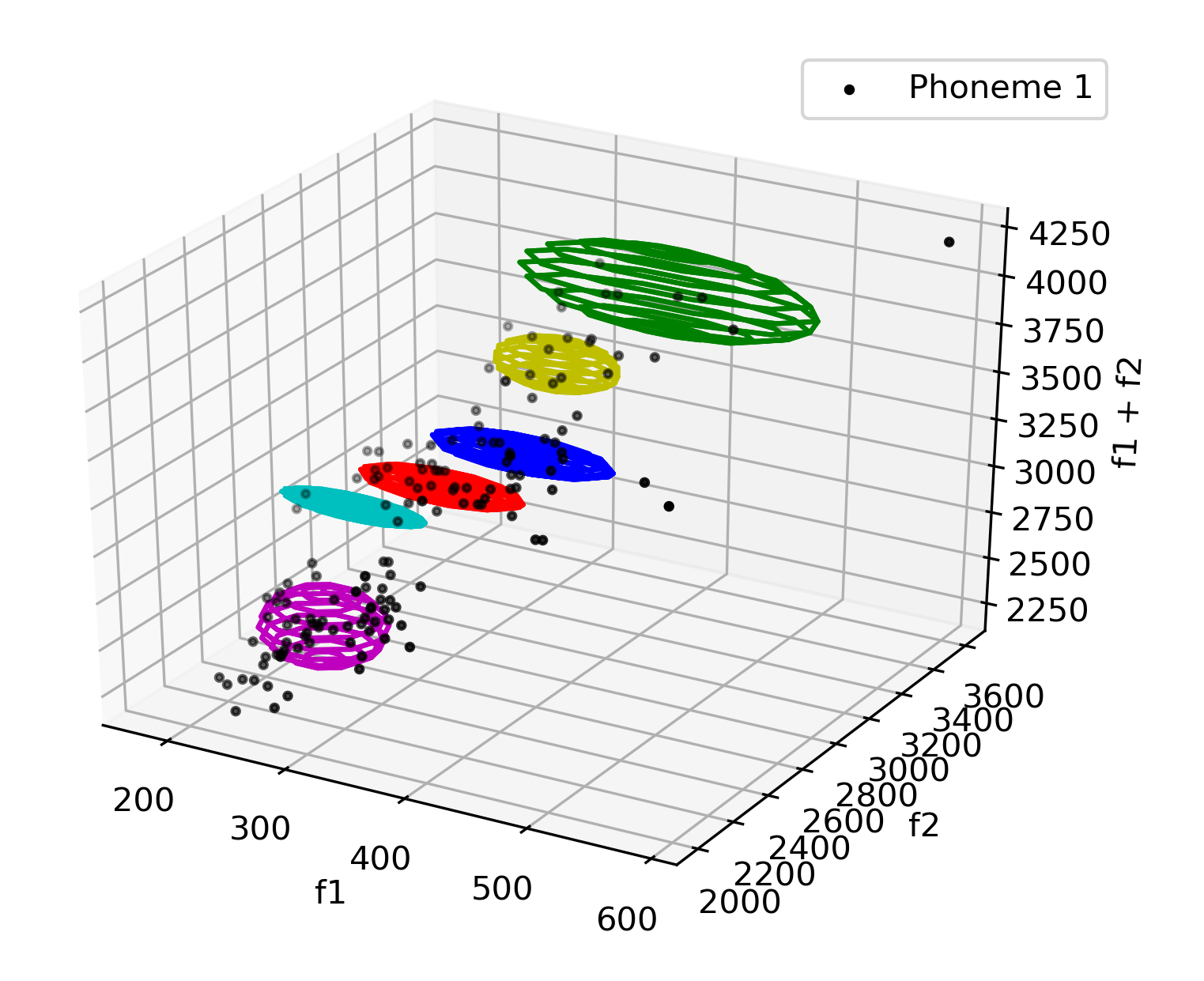


Figure 12. Use of Diagonal Entries of Covariance Matrix to Avoid Singularity Issues (K=6, Phoneme 1)

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:

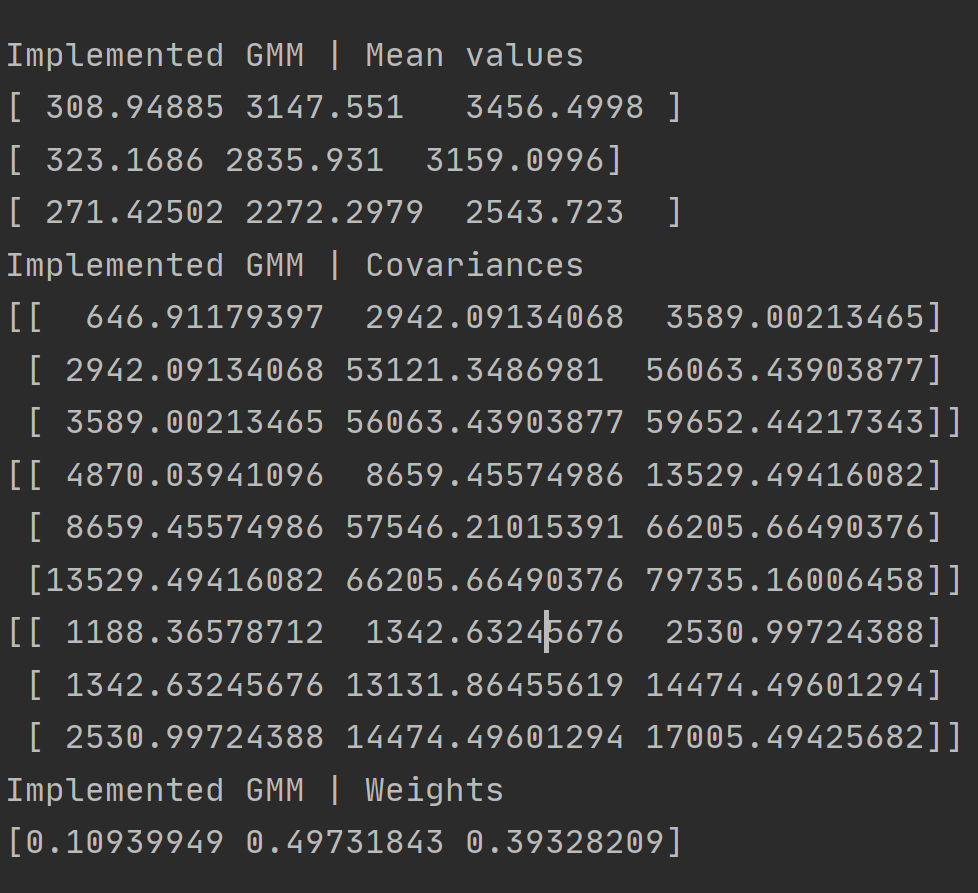
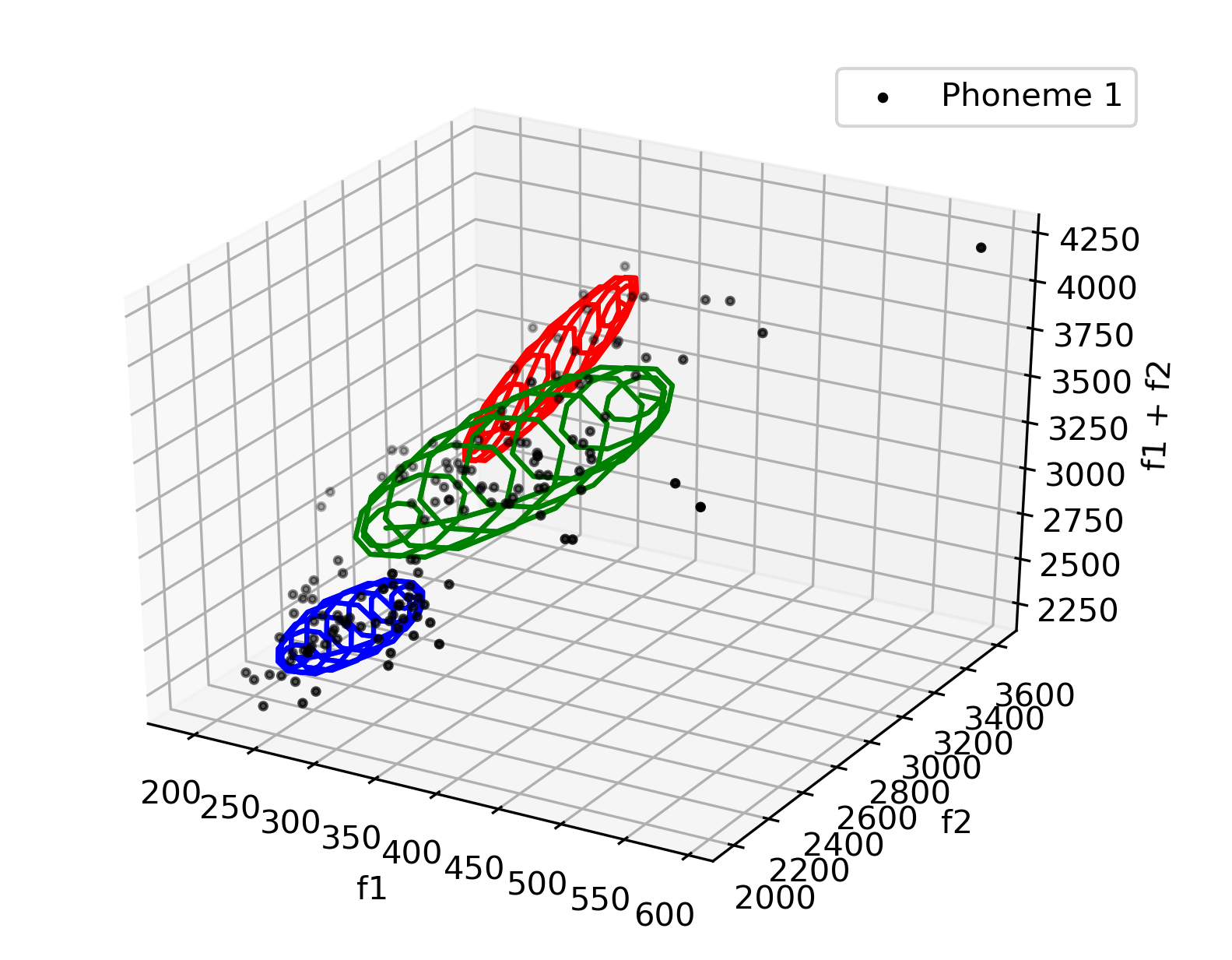


Figure 13. Use of Diagonal Regularization of Covariance Matrix to Avoid Singularity Issues (K=3, reg\_factor = 0.001, Phoneme 1)

For K=6 the results are shown below:

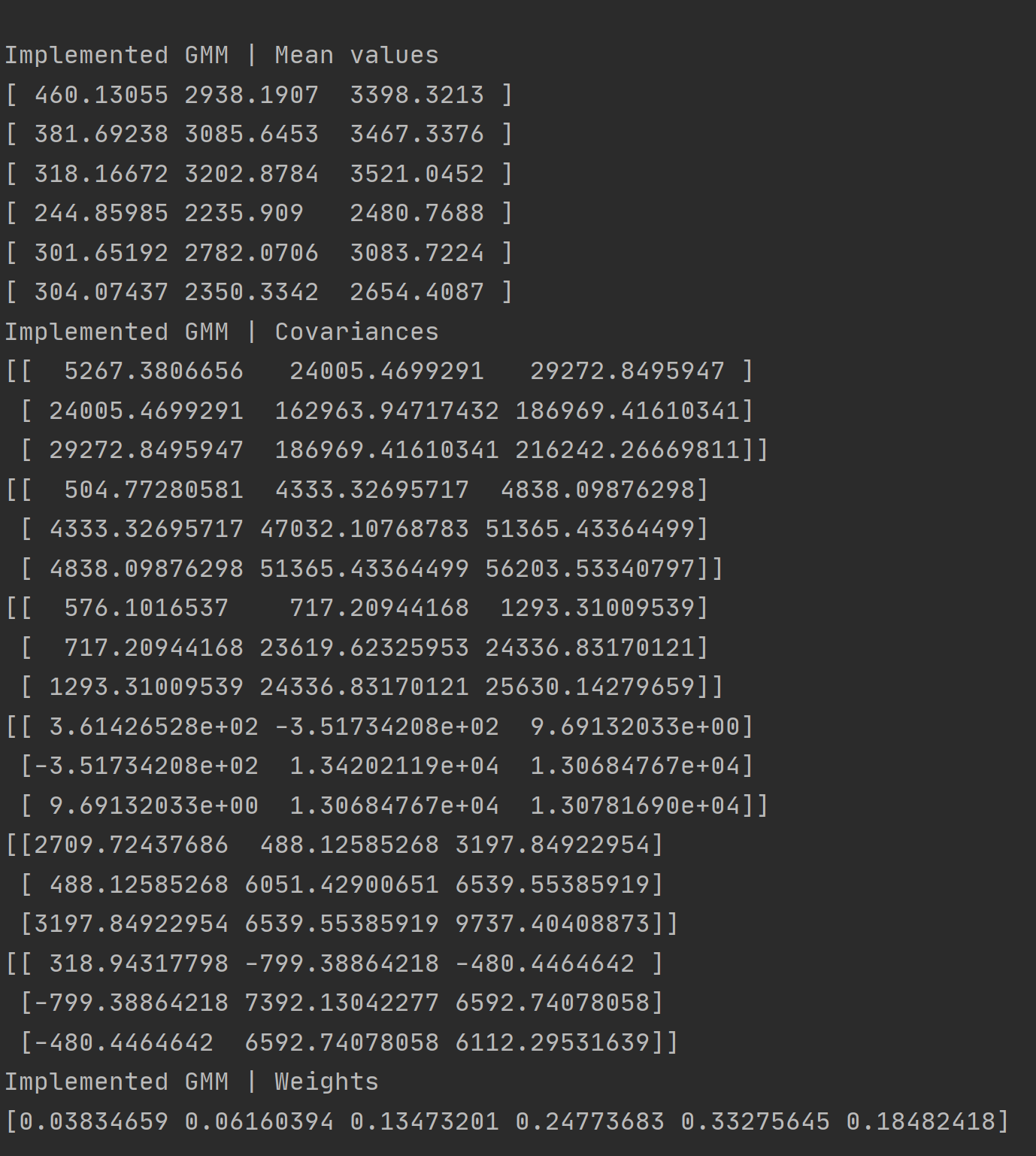
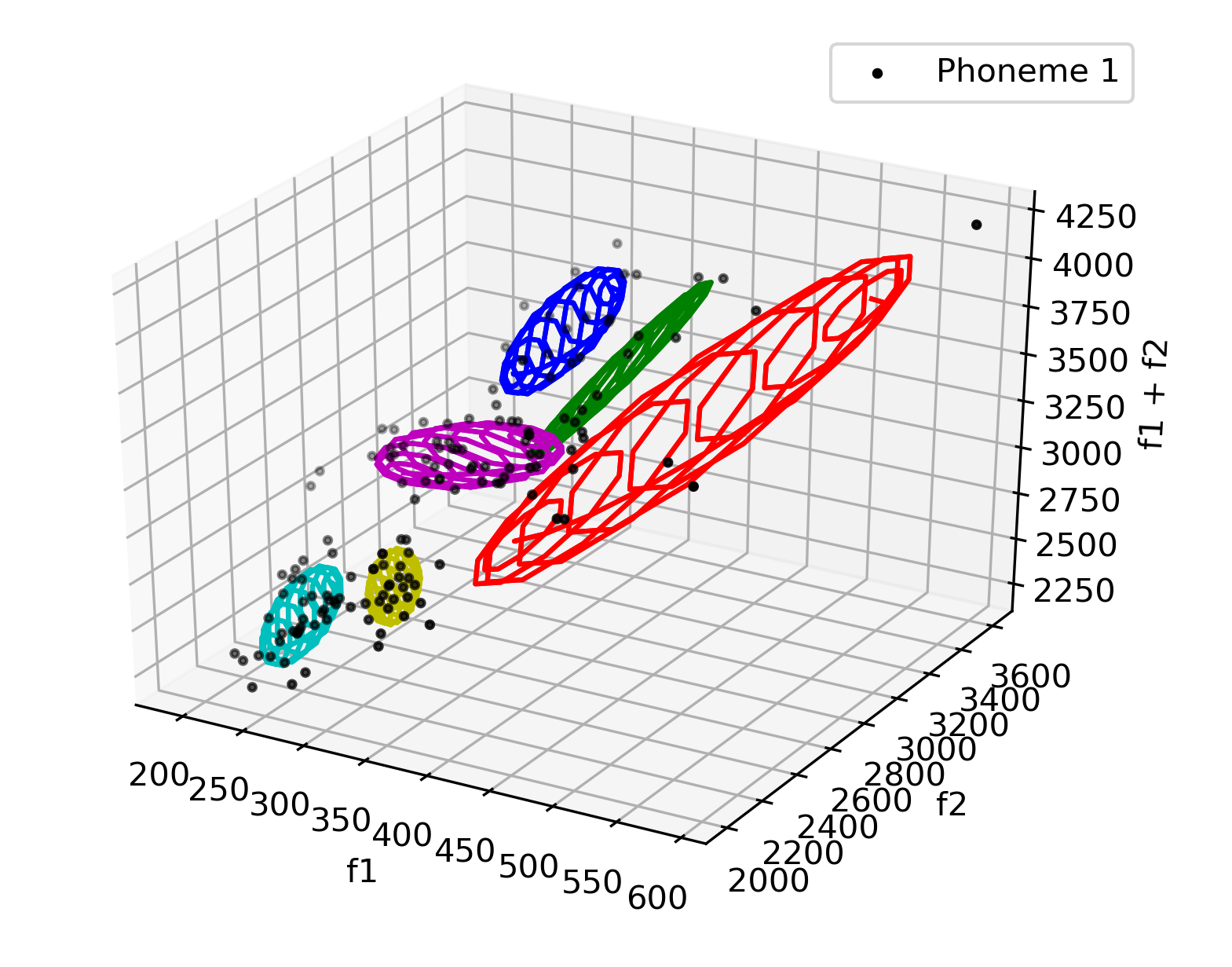


Figure 14. Use of Diagonal Regularization of Covariance Matrix to Avoid Singularity Issues (K=6, reg\_factor = 0.001, Phoneme 1)

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.

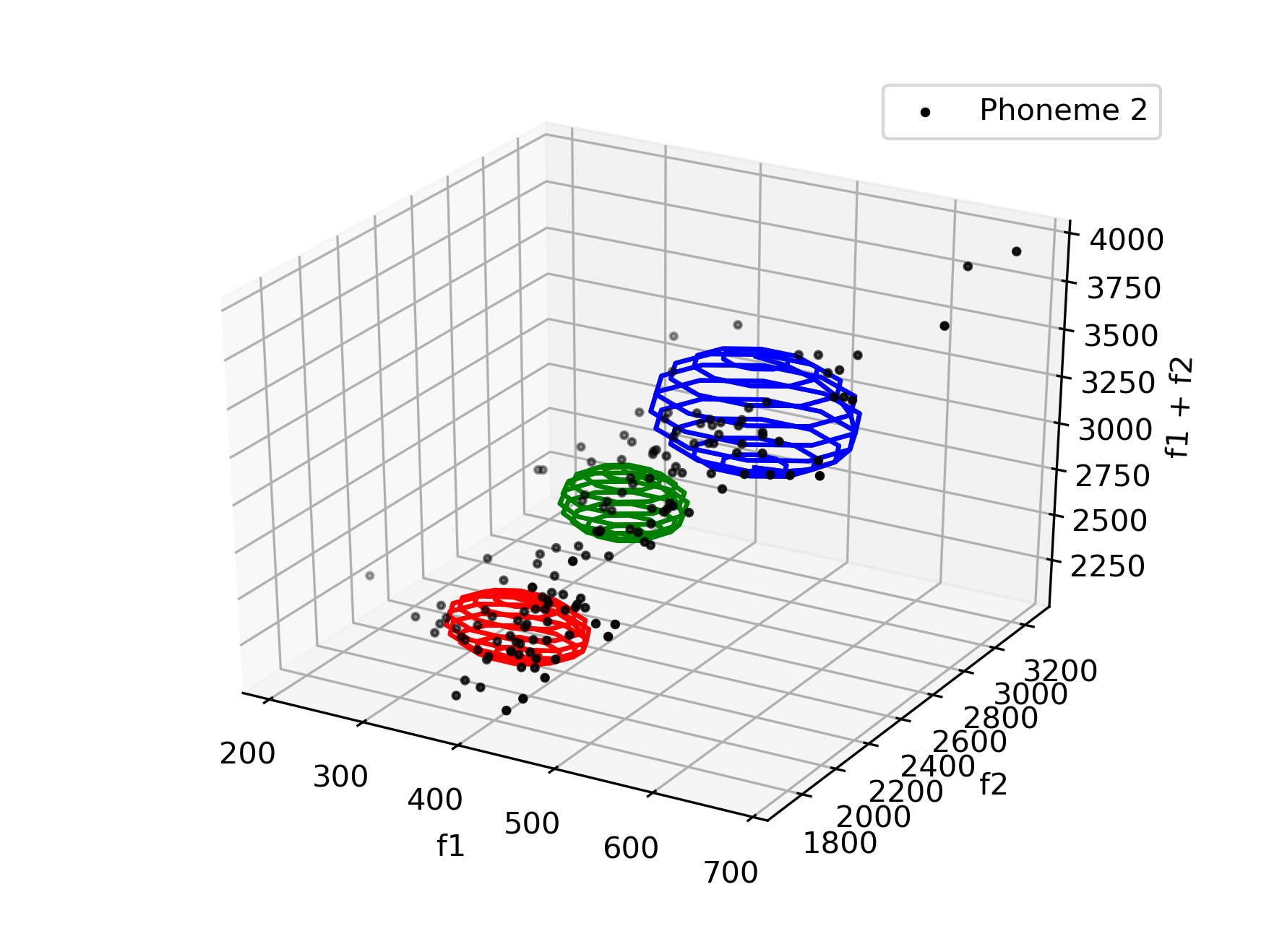
 

Figure 15. Use of Diagonal Entries of Covariance Matrix to Avoid Singularity Issues (K=3, Phoneme 2)

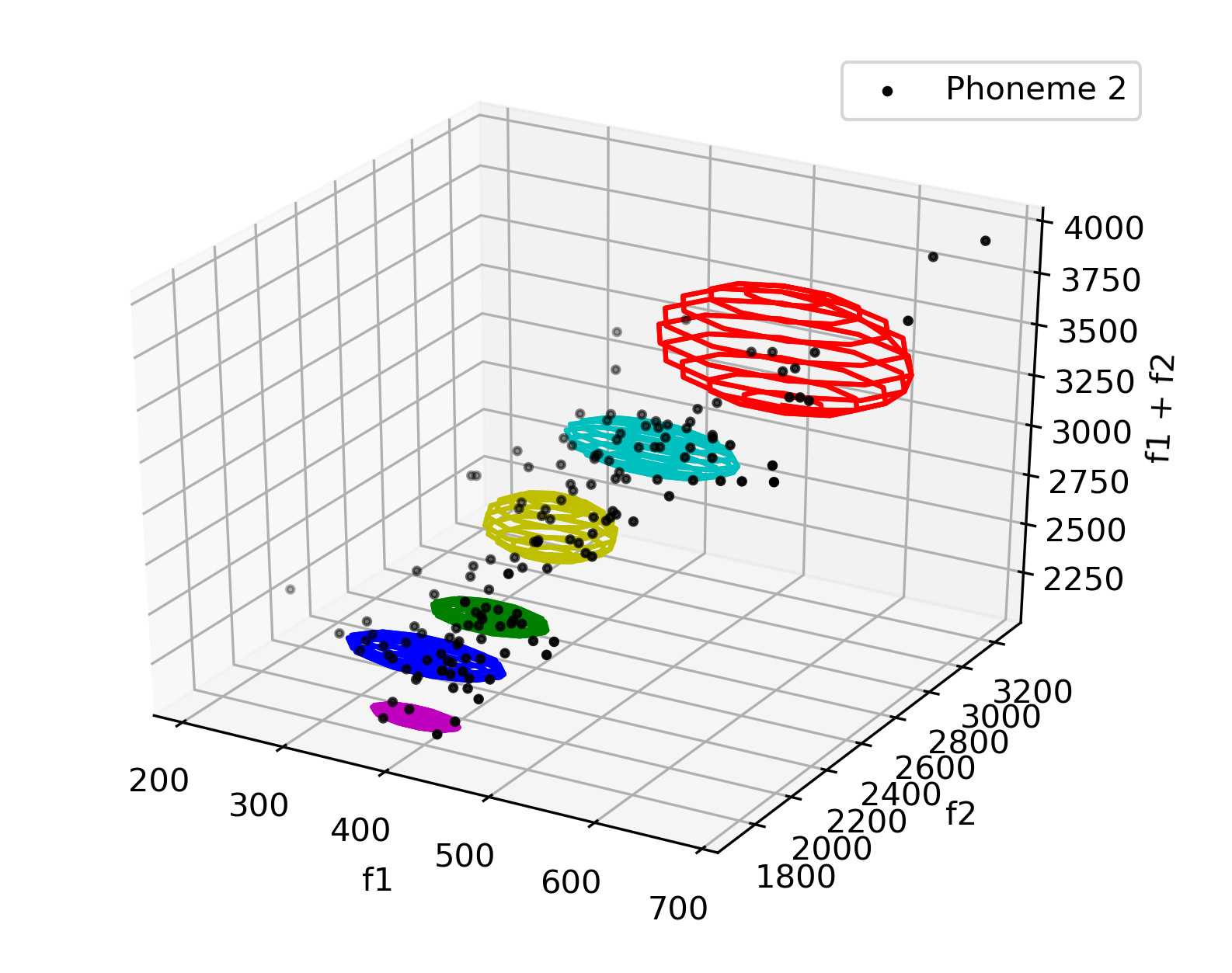
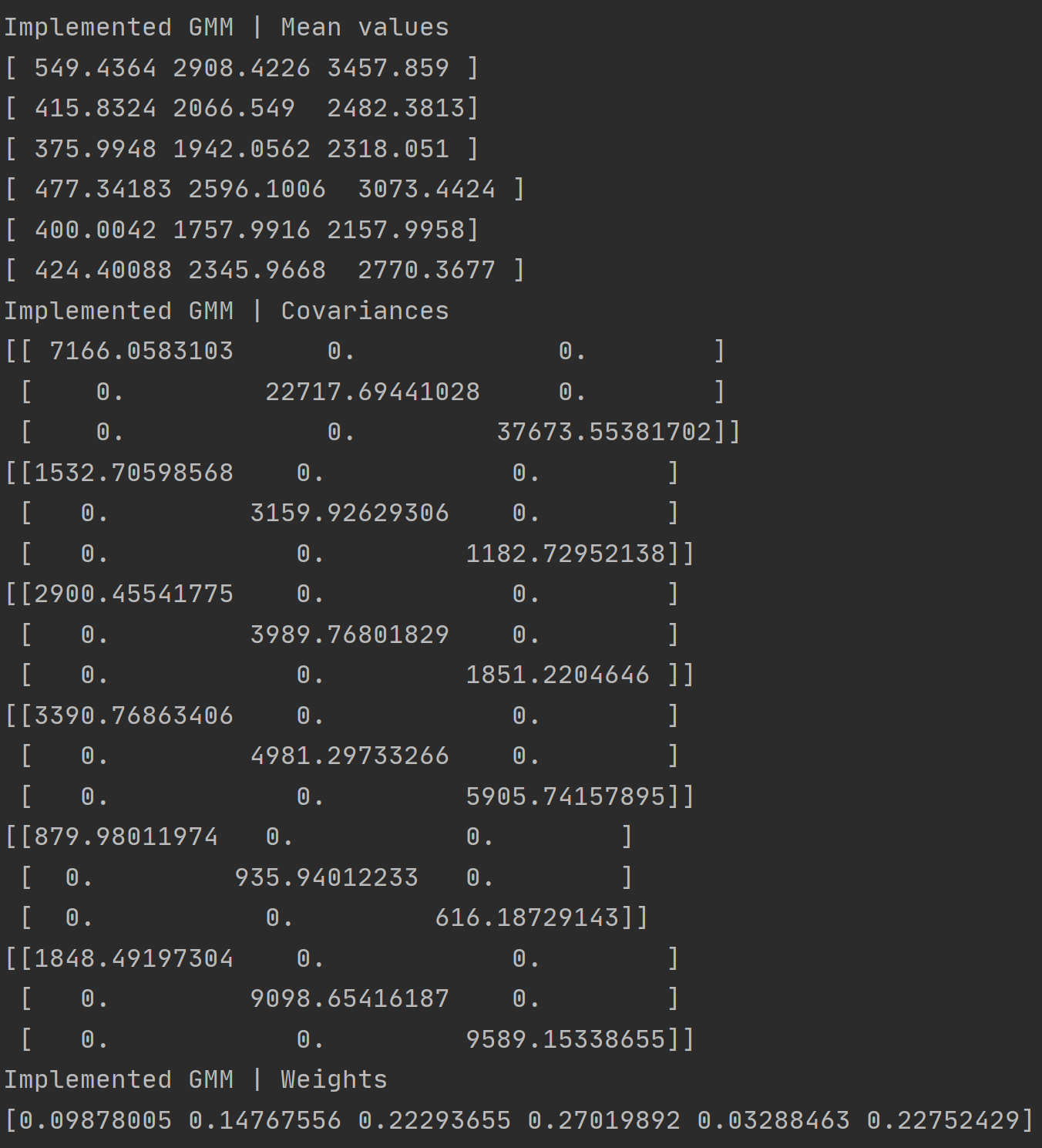
 

Figure . Use of Diagonal Entries of Covariance Matrix to Avoid Singularity Issues (K=6, Phoneme 2)

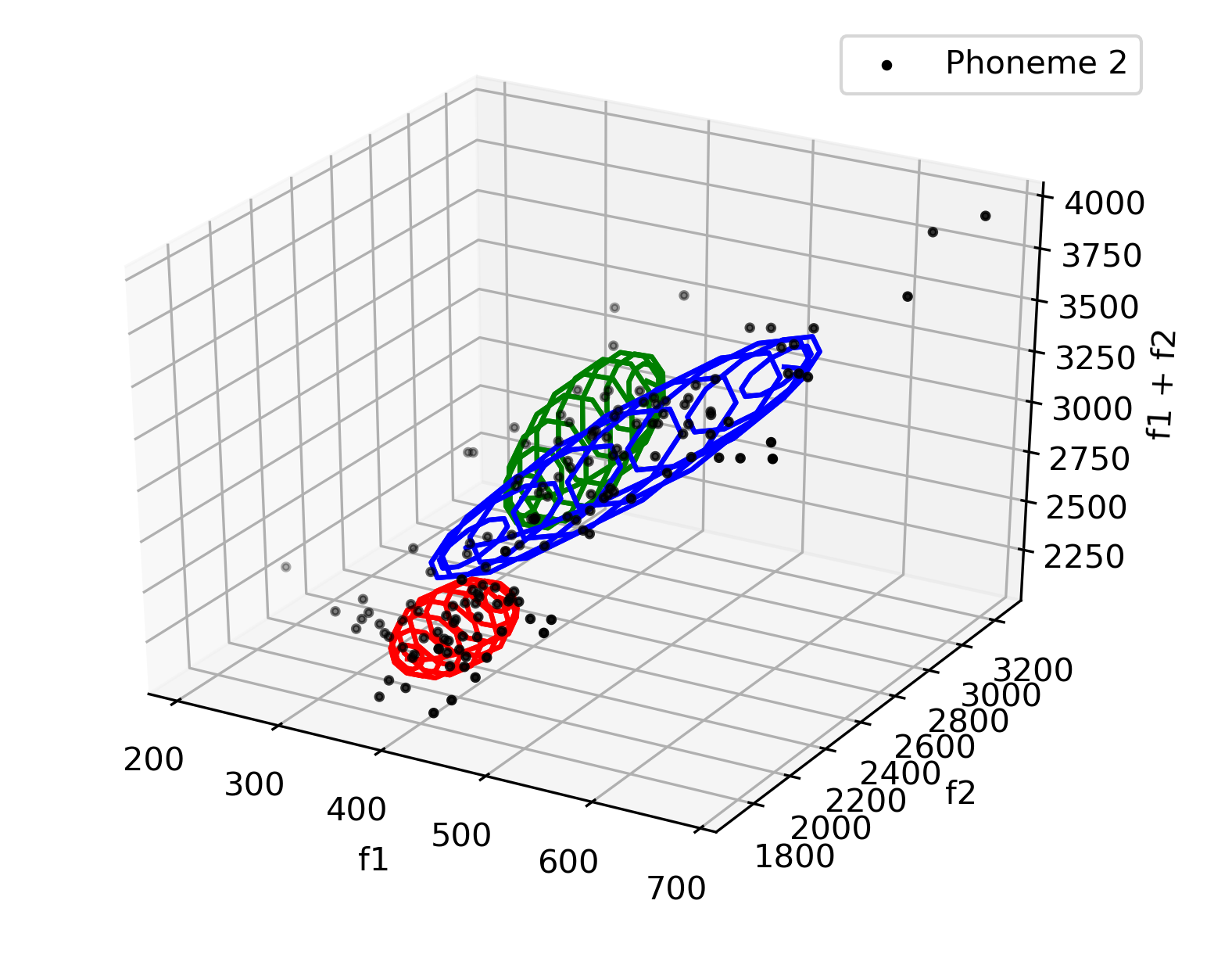
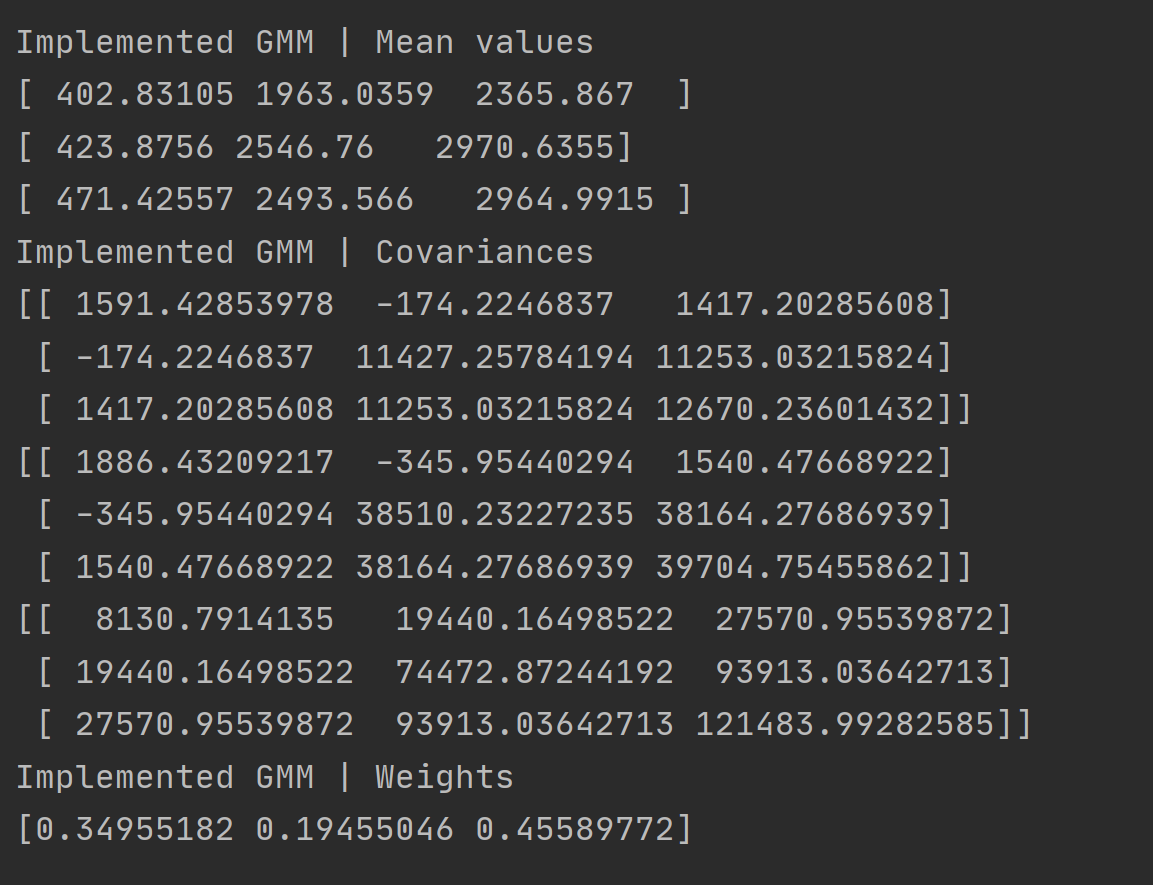
 

Figure 17. Use of Diagonal Regularization of Covariance Matrix to Avoid Singularity Issues (K=3, reg\_factor = 0.001, Phoneme 2)

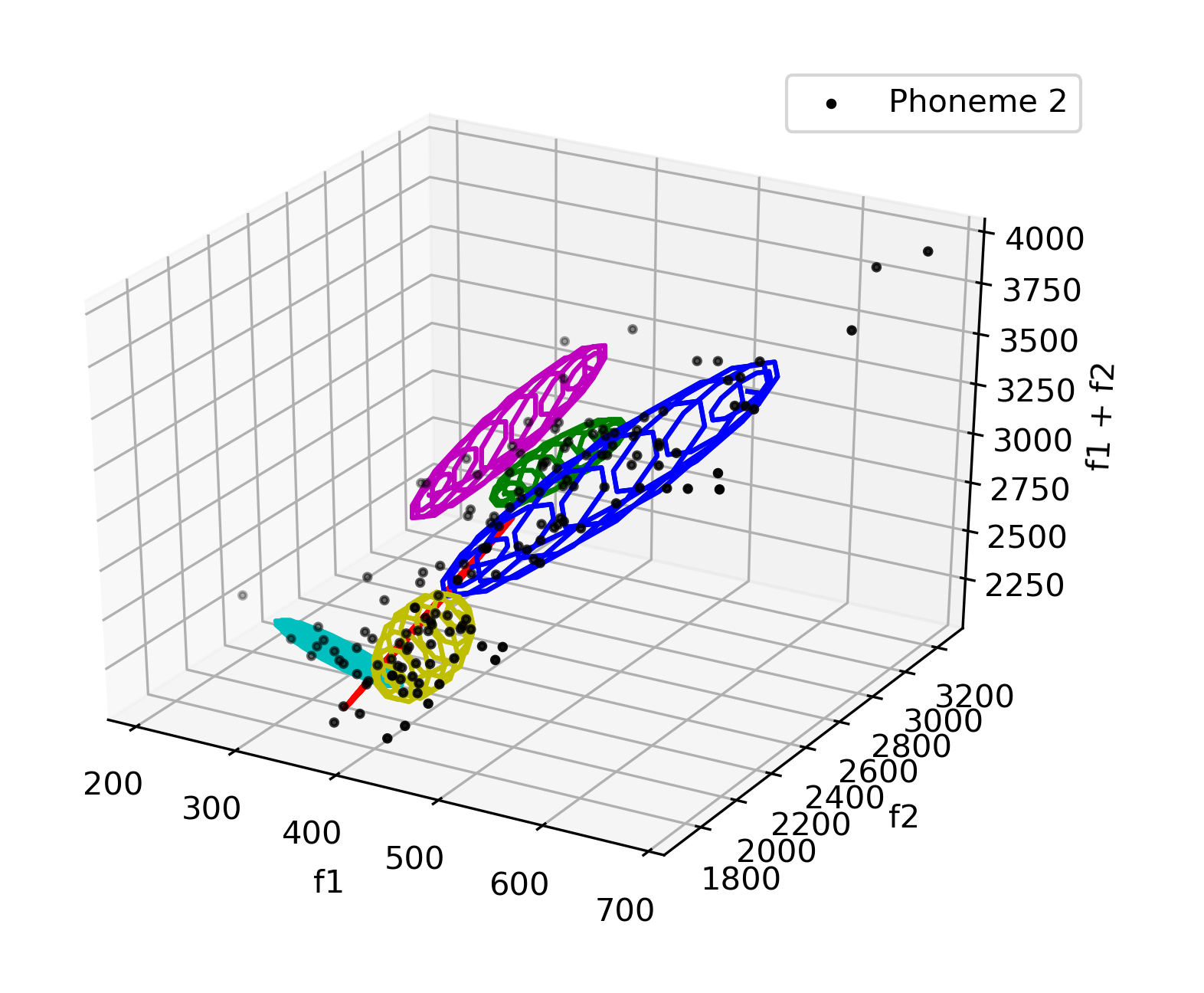
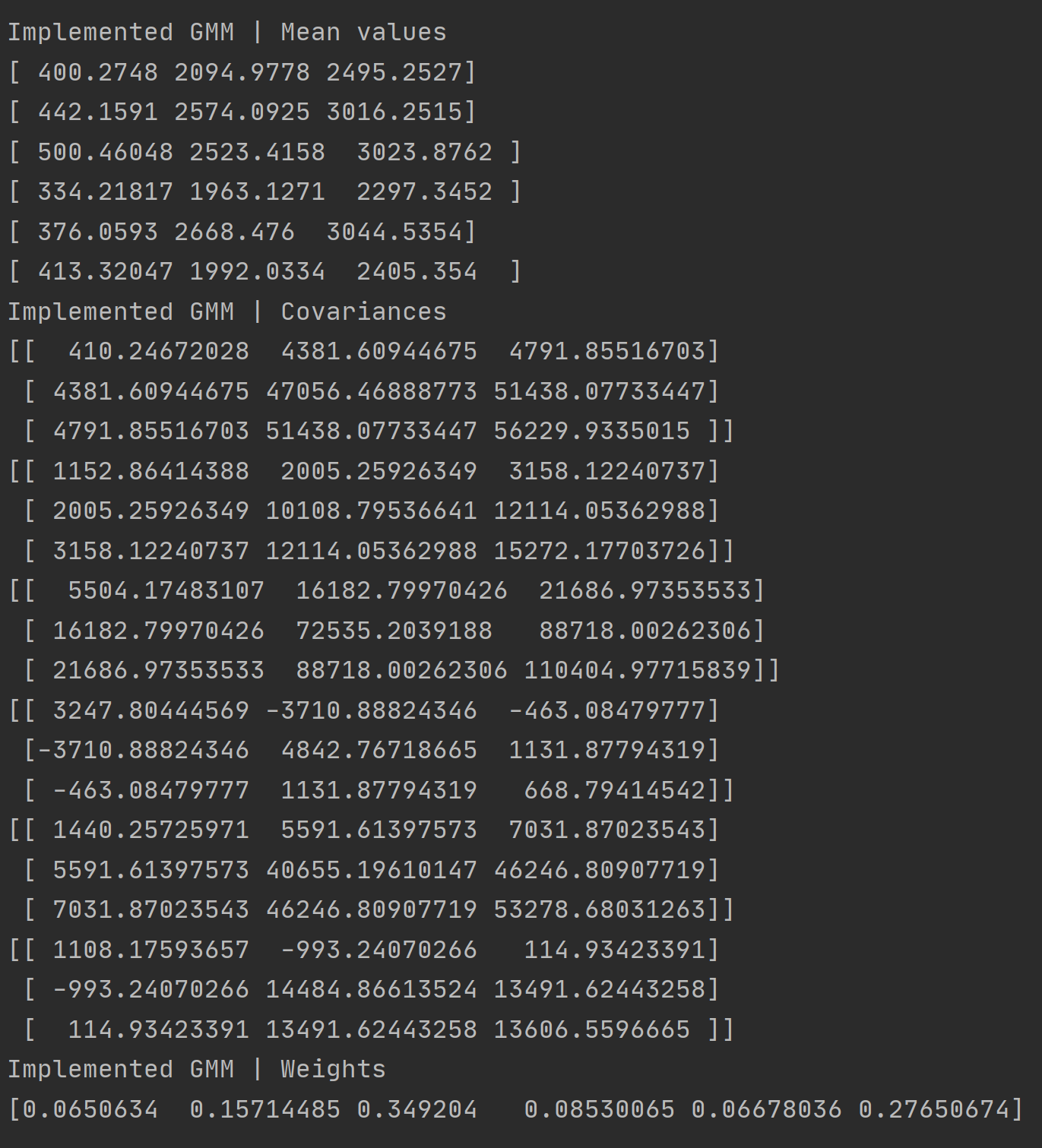
 

Figure . Use of Diagonal Regularization of Covariance Matrix to Avoid Singularity Issues (K=6, reg\_factor = 0.001, Phoneme 2)

**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()