AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

ECS708U/ECS708P Machine Learning

Assignment 2: Clustering and MoG

AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

The application of Clustering using the Gaussian Mixture Model to the Gordon Peterson and Harold Barney dataset of vowel format frequencies for phonemes is presented in this report. The dataset, which contains the fundamental frequency F0 and first three format frequency (F1-F3) of sustained English vowels is loaded from the "PB_data.npy" file, which contains four-vectors (F0–F4), as well as a vector "phoneme id" containing a number representing the phoneme's id. Only the datasets associated with formats F1 and F2 will be used in this study.

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

Below given is the code snippet for creating scatter plot for all phenomes and phenome1 against F1 and F2. The variable X_full stores the data F1 and F2 in column 0 and 1 respectively for all the phonemes and X_phoneme_1 stores the same only for phoneme 1.

```
# Write your code here
# Store f1 in the first column of X_full, and f2 in the second column of
X_full
X_full = np.column_stack((f1,f2))
```

```
# 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 = np.column_stack((X_full[phoneme_id==p_id,0],
X_full[phoneme_id==p_id,1]))
```

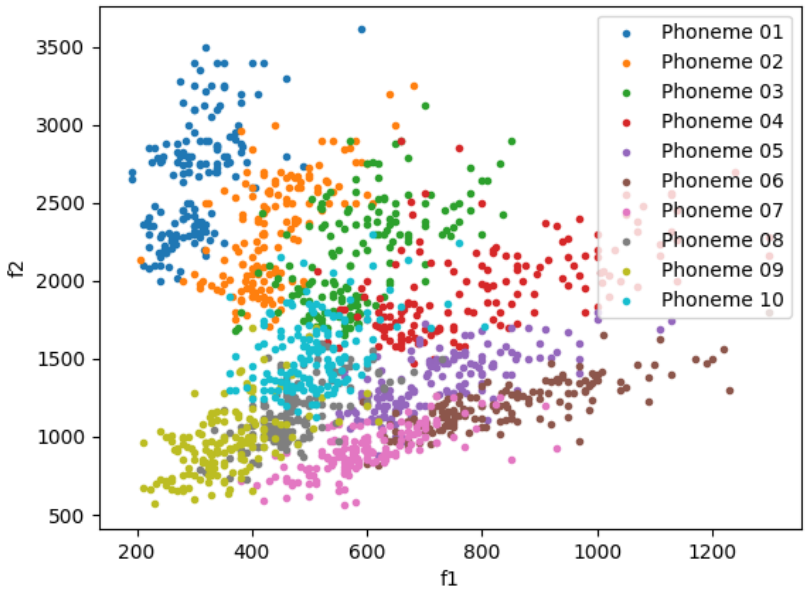

Figure 1: Plot of F1 against F2 for all given Phoneme

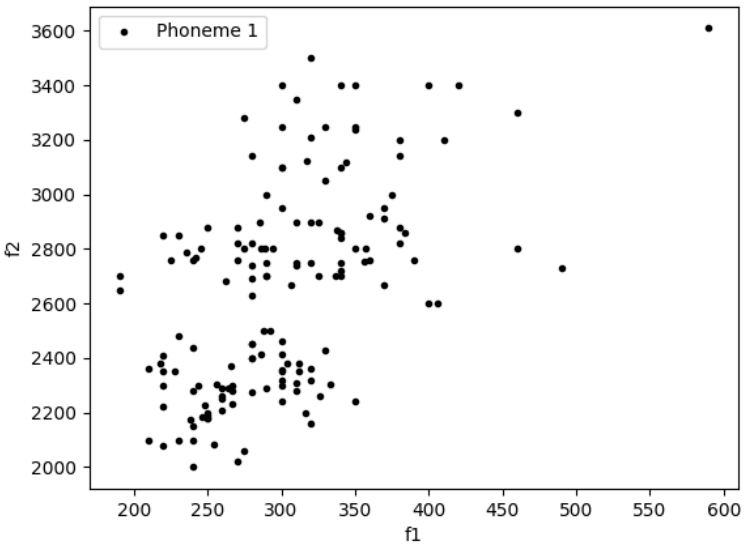

Figure 2: Plot of F1 against F2 for Phoneme 1

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

The Figure 1 and Figure 2 displays the generated scatter plot for all phoneme and phoneme 1 respectively against F1 and F2.

AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

TASK 2

Train the data for phonemes 1 and 2 with MoGs. You are provided with python files task 2.py and plot 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.

Below is the code snippet for generating the X_phoneme with the necessary phoneme id.

```
# Write your code here
# Store f1 in the first column of X_full, and f2 in the second column of
X_full
X_full = np.column_stack((f1,f2))
```

```
# 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 = np.column_stack((X_full[phoneme_id==p_id,0],
X_full[phoneme_id==p_id,1]))
```

Generating the dataset X_phoneme with phoneme 1 and training GMM with 3 components (i.e., p_id = 1 and k = 3) and running the code generates the following result.

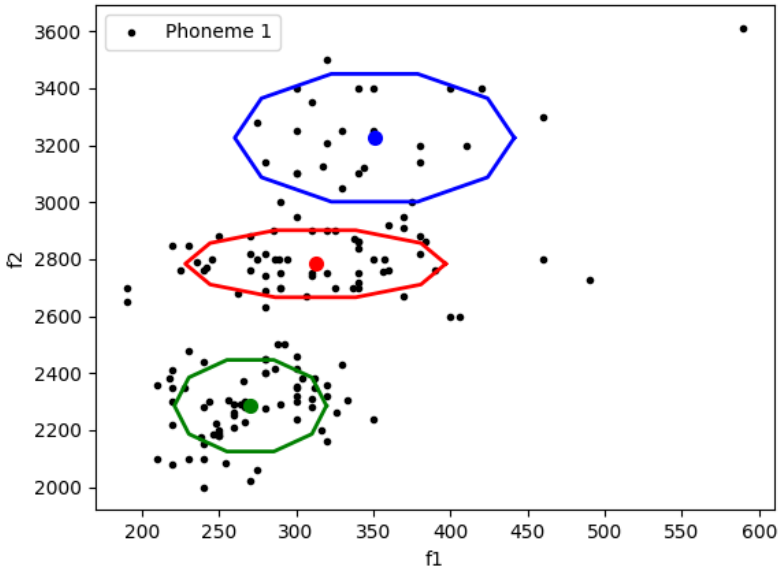

Figure 3: Plot against F1 against F2 for Phoneme 1 with 3 cluster components first time

Implemented GMM | Mean values

[312.58972 2783.891]

[270.3952 2285.4653]

[350.8434 3226.3025]

Implemented GMM | Covariances

[[3562.62140445 0.]

[0. 7656.93220505]]

[[1213.73813979 0.]

[0. 14278.43124378]]

[[4102.68990287 0.]

[0. 27839.03575675]]

Implemented GMM | Weights

[0.38097385 0.43514452 0.18388164]

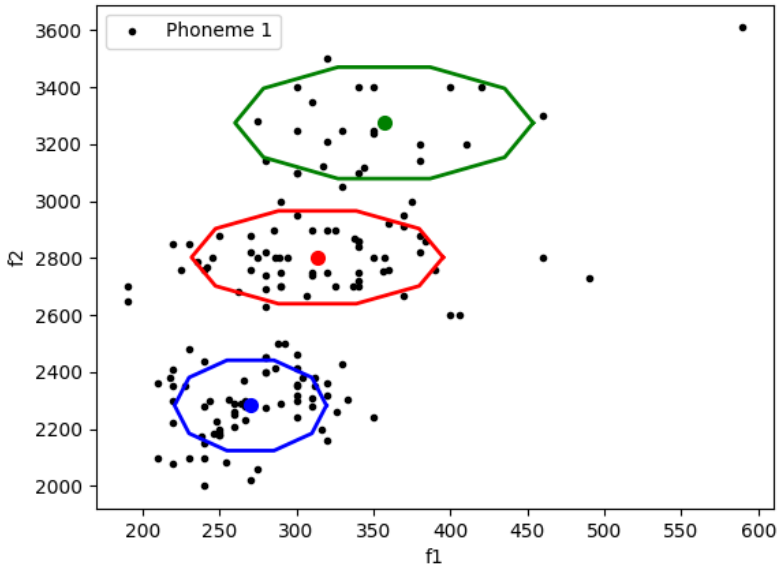

Figure 4: Plot against F1 against F2 for Phoneme 1 with 3 cluster components second time

Implemented GMM | Mean values

[313.61664 2803.1062]

[356.96506 3275.2568]

[270.17844 2282.9531]

Implemented GMM | Covariances

[[3340.97947156 0.]

[0. 14676.19602066]]

[[4687.96005511 0.]

[0. 21186.56312922]]

[[1216.32479908 0.]

[0. 13925.21973939]]

Implemented GMM | Weights

[0.42509426 0.14540708 0.42949866]

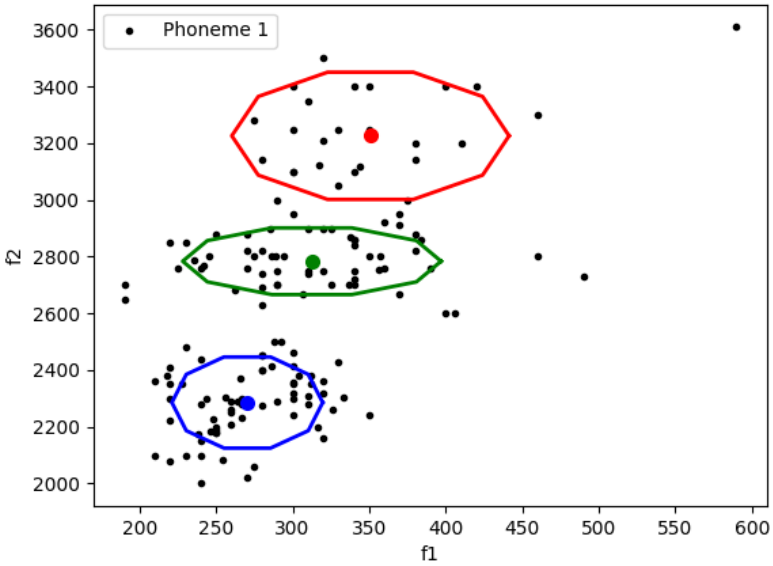

Figure 5: Plot against F1 against F2 for Phoneme 1 with 3 cluster components third time

Implemented GMM | Mean values

[350.8445 3226.3357]

[312.5911 2783.8972]

[270.3952 2285.4653]

Implemented GMM | Covariances

[[4102.8576619 0.]

[0. 27830.45870827]]

[[3562.59967322 0.]

[0. 7657.76145079]]

[[1213.73840568 0.]

[0. 14278.42181486]]

Implemented GMM | Weights

[0.18386242 0.38099322 0.43514436]

Running the data containing F1 and F2 multiple times for Phoneme 1 generates different clusters which can be seen in Figures 3, 4 and 5.

The difference in clusters is because of the random initialization for centres of each clusters every time we run. The closer a point is to the Gaussian's centre, the more likely it belongs to that cluster. This

should make intuitive sense since with a Gaussian distribution we are assuming that most of the data lies closer to the centre of the cluster.

Generating the dataset X_phoneme with phoneme 1 and training GMM with 6 components

(i.e., p_id = 1 and k = 6)

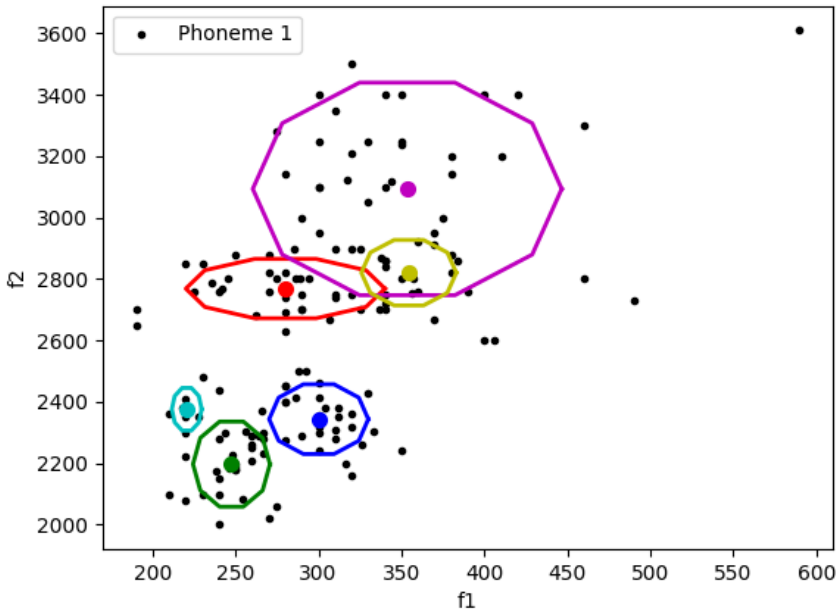

Figure 6: Plot against F1 against F2 for Phoneme 1 with 6 cluster components

Implemented GMM | Mean values

[280.06842 2768.903]

[247.58586 2197.0278]

[300.17416 2343.5957]

[220.64333 2375.9526]

[353.5219 3093.7488]

[354.47702 2820.726]

Implemented GMM | Covariances

[[1800.65536673 0.]

[0. 5165.09333964]]

[[269.06454489 0.]

[0. 10634.38701775]]

[[446.10006277 0.]

[0. 7135.3340821]]

[[40.45542369 0.]

[0. 2667.32349973]]

[[4334.05803239 0.]

[0. 66274.20067307]]

[[410.17240632 0.]

[0. 6280.58673755]]

Implemented GMM | Weights

[0.22253523 0.18443035 0.20792826 0.04035357 0.26132472 0.08342787]

Generating the dataset X_phoneme with phoneme 2 and training GMM with 3 components (i.e., p_id = 2 and k = 3)

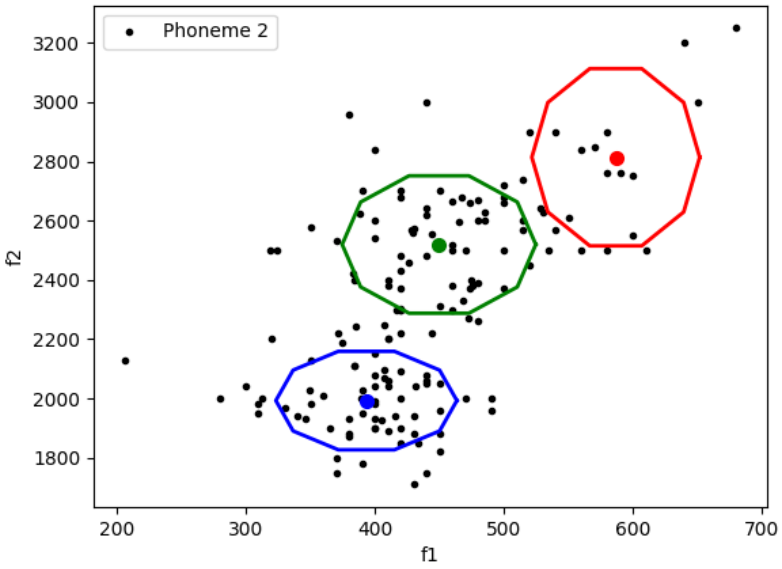

Figure 7: Plot against F1 against F2 for Phoneme 2 with 3 cluster components

Implemented GMM | Mean values

[586.5443 2814.1074]

[449.66672 2519.664]

[393.45197 1993.0735]

Implemented GMM | Covariances

[[2112.08165801 0.]

[0. 49424.45929544]]

[[2808.67210512 0.]

[0. 29722.42755865]]

[[2454.95676364 0.]

[0. 15261.94805286]]

Implemented GMM | Weights

[0.09492041 0.46993007 0.43514952]

Generating the dataset X_phoneme with phoneme 2 and training GMM with 6 components (i.e., p_id = 2 and k = 6)

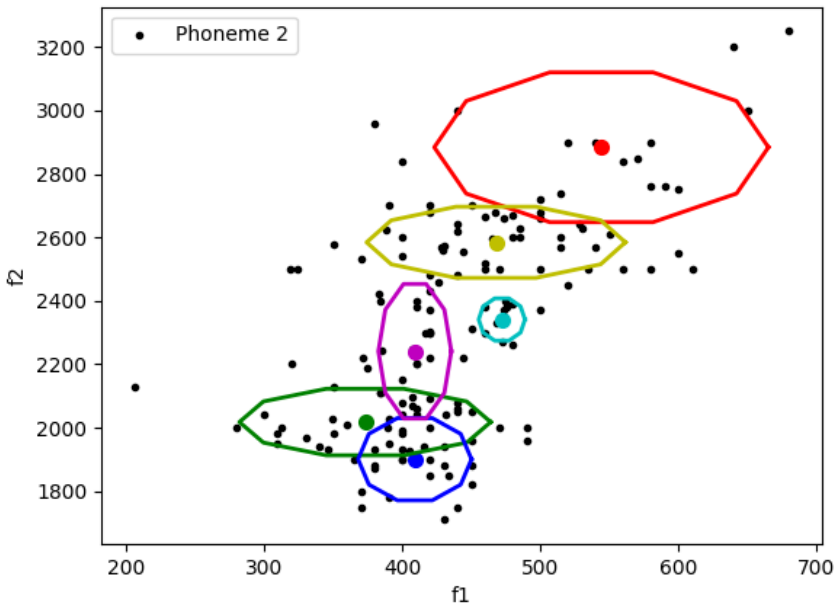

Figure 8: Plot against F1 against F2 for Phoneme 2 with 6 cluster components

Implemented GMM | Mean values

[544.0881 2884.1077]

[373.14145 2018.0928]

[409.13397 1900.8722]

[472.09512 2341.2324]

[409.16095 2241.1914]

[467.85864 2584.585]

Implemented GMM | Covariances

[[7304.52643282 0.]

[0. 30808.35373906]]

[[4130.72228779 0.]

[0. 6074.52891252]]

[[842.15150495 0.]

[0. 9276.34360553]]

[[140.1742741 0.]

[0. 2453.82616936]]

[[348.50499779 0.]

[0. 24864.99395733]]

[[4382.00396321 0.]

[0. 6951.10889809]]

Implemented GMM | Weights

[0.10307978 0.1850247 0.16781078 0.06759968 0.17761728 0.29886777]

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 miss-classification error. Remember that a classification under the ML compares p(x; θ1), where θ1 are the parameters of the MoG learnt for the first phoneme, with p(x; θ2), where θ2 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 your wrote, explanations of what the code does and comment on the differences on the classification performance

---

```
# Write your code here
# Store f1 in the first column of X_full, and f2 in the second column of
X_full
X_full = np.column_stack((f1,f2))
```

```
# 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

ph_1 = phoneme_id[phoneme_id == 1]
ph_2 = phoneme_id[phoneme_id == 2]

X_phoneme_1 = np.zeros((np.sum(phoneme_id==1), 2))
X_phoneme_1 = np.column_stack((X_full[phoneme_id==1,0],
X_full[phoneme_id==1,1]))
X_phoneme_2 = np.zeros((np.sum(phoneme_id==2), 2))
X_phoneme_2 = np.column_stack((X_full[phoneme_id==2,0],
X_full[phoneme_id==2,1]))

ph_1_2 = np.concatenate([ph_1, ph_2])
X_phonemes_1_2 = np.concatenate((X_phoneme_1, X_phoneme_2))
```

```
# 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()

# file path for phoneme 1 and phoneme 2 params learned in task 2
phoneme1_file = 'data/GMM_params_phoneme_01_k_0'+str(k)+'.npy'
phoneme2_file = 'data/GMM_params_phoneme_02_k_0'+str(k)+'.npy'

# prediction for phoneme 1
phoneme_1_data = np.ndarray.tolist(np.load(phoneme1_file,
allow_pickle=True))
predictions1 = get_predictions(phoneme_1_data['mu'], phoneme_1_data['s'],
phoneme_1_data['p'], X)
```

```
predictions1 = np.sum(predictions1, axis=1)

# predictions for phoneme 2
phoneme_2_data = np.ndarray.tolist(np.load(phoneme2_file,
allow_pickle=True))
predictions2 = get_predictions(phoneme_2_data['mu'], phoneme_2_data['s'],
phoneme_2_data['p'], X)
predictions2 = np.sum(predictions2, axis=1)

predictions = np.zeros((np.sum(phoneme_id == 1) + np.sum(phoneme_id == 2),
1))

for i in range(len(predictions)):
    if predictions1[i] > predictions2[i]:
        predictions[i] = 1
    else:
        predictions[i] = 2

# calculating the count of correct data
total_correct_data = 0
for i in range(len(ph_1_2)):
    if predictions[i] == ph_1_2[i]:
        total_correct_data += 1

# calculating the accuracy
total_data = len(ph_1_2)
accuracy = (total_correct_data/total_data)*100
miss_classification = (1-(total_correct_data/total_data))*100
```

The code above calculates the accuracy and miss-classification error on samples from both phenomes 1 and 2 that have been pre-trained on phenome 1 and 2, respectively.

The code performs the following:

1. Import the 2 MoG's of phenome 1 and phenome 2 stored from Task 2 based on the k value.
2. Calculate predictions on phenome 1 and phenome 2 using Maximum likelihood estimation with the imported MoG's.
3. Compare the predictions and store the data in variable predictions
4. Calculate the accuracy by finding the total number of correctly predicted data and dividing it by the total number of data.
5. Calculate the misclassification error using the formula 1-(total_correct_data – total_data).

Below given is the output for running the code with k values as 3 and 6 respectively.

Accuracy and Miss-classification error on phoneme 1 and 2 with k=3

Accuracy using GMMs with 3 components: 95.07%

Misclassification error using GMMs with 3 components: 4.93%

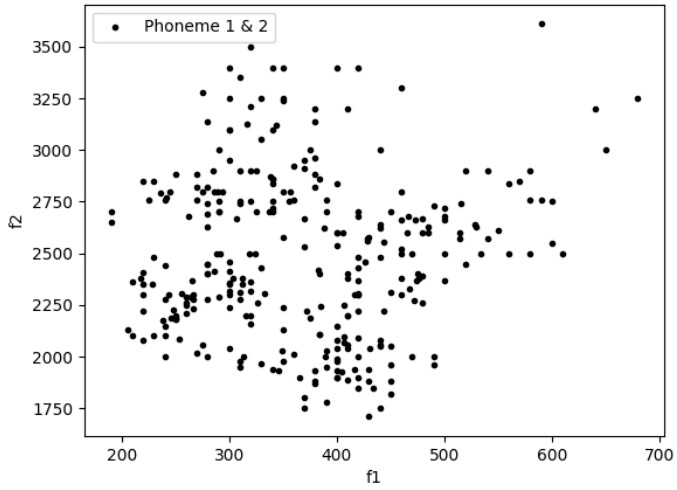

Figure 9: Plot against F1 against F2 for Phoneme 1 and Phoneme 2

Accuracy and Miss-classification error on phoneme 1 and 2 with k=6

Accuracy using GMMs with 6 components: 95.39%

Misclassification error using GMMs with 6 components: 4.61%

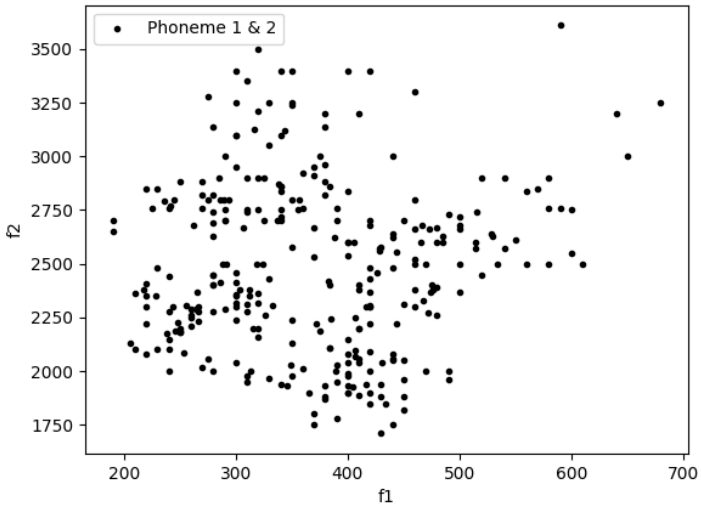

Figure 10: Plot against F1 against F2 for Phoneme 1 and Phoneme 2

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 x1 is classified as belonging to phoneme 1, and is 2 otherwise. x1 is a vector whose elements are between the minimum and the maximum value of F1 for the first two phonemes, and x2 similarly for F2. Display the classification matrix. Include the lines of code in your report, comment them, and display the classification matrix.

```
# Write your code here
# Store f1 in the first column of X_full, and f2 in the second column of
X_full
X_full = np.column_stack((f1,f2))
```

```
# 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_phoneme_1 = np.zeros((np.sum(phoneme_id==1), 2))
X_phoneme_1 = np.column_stack((X_full[phoneme_id==1,0],
X_full[phoneme_id==1,1]))
X_phoneme_2 = np.zeros((np.sum(phoneme_id==2), 2))
X_phoneme_2 = np.column_stack((X_full[phoneme_id==2,0],
X_full[phoneme_id==2,1]))

X_phonemes_1_2 = np.concatenate((X_phoneme_1, X_phoneme_2))
```

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

k = 3

# loading data for phoneme 1
phoneme1_file = 'data/GMM_params_phoneme_01_k_0'+str(k)+'.npy'
phoneme_1_data = np.ndarray.tolist(np.load(phoneme1_file,
allow_pickle=True))

# loading data for phoneme 2
phoneme2_file = 'data/GMM_params_phoneme_02_k_0'+str(k)+'.npy'
phoneme_2_data = np.ndarray.tolist(np.load(phoneme2_file,
allow_pickle=True))

# creating custom grid of shape N_f2 x N_f1
custom_grid = np.zeros((N_f1,N_f2,2))
for i in range(N_f1):
    for j in range(N_f2):
        custom_grid[i][j] = [int(i + min_f1),(j + min_f2)]

def get_predicted_value(k, grid_val):

        phoneme_01 = get_predictions(phoneme_1_data['mu'],
phoneme_1_data['s'], phoneme_1_data['p'], grid_val)
        phoneme_01 = np.sum(phoneme_01,axis=1)
        phoneme_02 = get_predictions(phoneme_2_data['mu'],
phoneme_2_data['s'], phoneme_2_data['p'], grid_val)
        phoneme_02 = np.sum(phoneme_02,axis=1)

        prediction = []
        # appending "0.0" if point belongs to phoneme 1 and "1.0" if points
belong to phoneme 2
        for z1,z2 in zip(phoneme_01, phoneme_02):
                if z1 > z2:
                        prediction.append(0.0)
                else:
                        prediction.append(1.0)

        pred = np.array(prediction)
        return pred

# Store prediction in a 2D numpy array named "M"
M = np.ndarray(shape=(N_f2, N_f1))
for i in range(0,N_f2):
        M[i,:] = get_predicted_value(3, custom_grid[:,i])
```

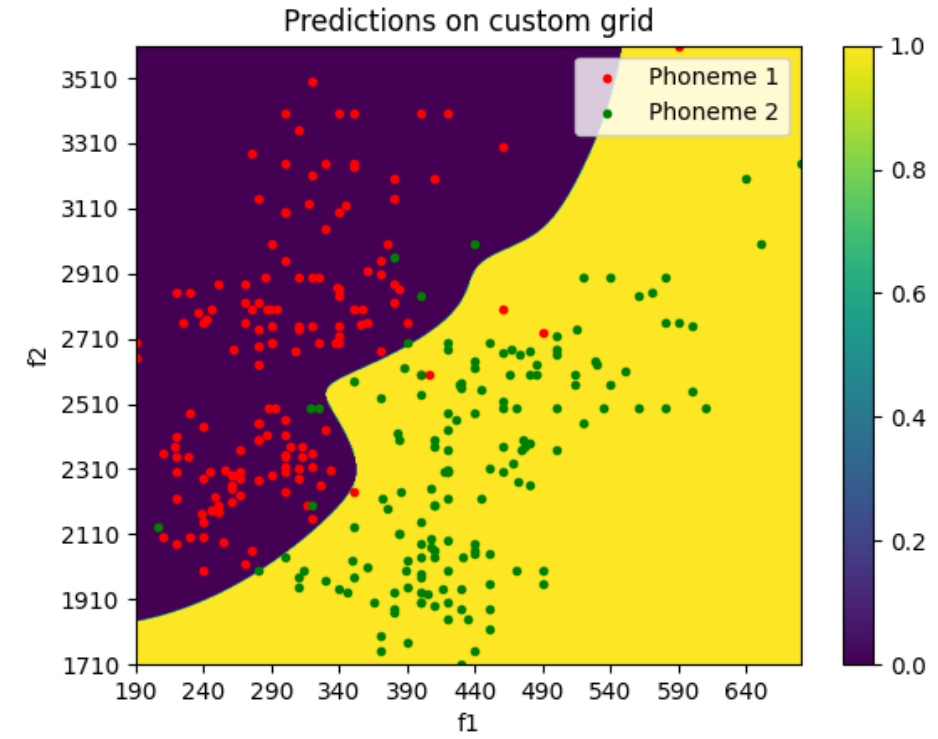

Figure 11: Predictions on custom grid

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] (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

When running the model with new data the following error is thrown

ValueError: Input contains NaN, infinity or a value too large for dtype('float64').

The error is caused by the third column F1+F2, where variance can be zero. When the variance gets to zero, the likelihood of the Gaussian component goes to infinity causing the error.

We can hope to avoid singularities by using suitable heuristics, for instance adding a randomly chosen value to the gaussian component while resetting its covariance to some large value.

```
# 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 = np.column_stack((f1,f2,f1+f2))
```

```
# 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 = np.zeros((len(phoneme_id==p_id), 3))
X_phoneme = np.column_stack((X_full[phoneme_id==p_id,0],
X_full[phoneme_id==p_id,1], X_full[phoneme_id==p_id,2]))
```

```
# Write your code here
# Suggest ways of overcoming the singularity
# Method 1
#s[i, :, :] += 0.001 * np.identity(D)

# Method 2
s[i, :, :] = np.identity(D) * np.diag(s[i])
```

We can avoid the error by using any of the two above methods in the above code snippet.

Below given are the results of running the code with p_id=1 with k=3 and k=6 respectively

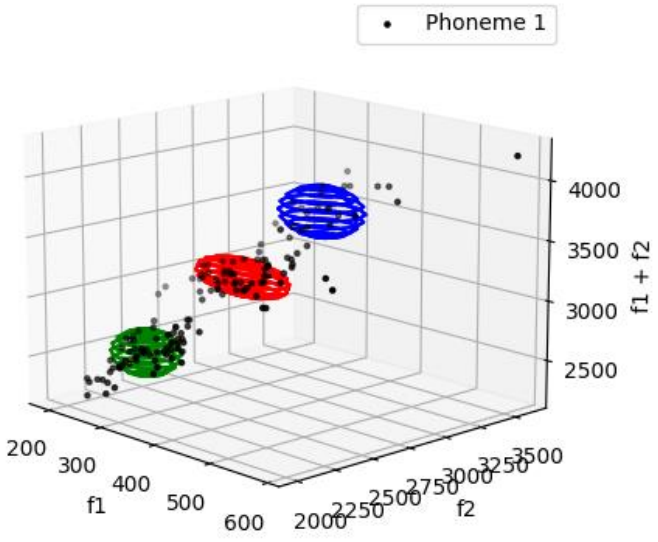

Figure 12: MoG with k = 3 on phoneme 1

Implemented GMM | Mean values
[312.71185 2784.9531 3097.665]
[270.36642 2284.796 2555.1624]
[342.7787 3226.5718 3569.3503]

Implemented GMM | Covariances
[[3545.78011851 0. 0.]
[0. 7550.6154196 0.]
[0. 0. 11152.38896995]]
[[1215.42236466 0. 0.]
[0. 14023.67497821 0.]
[0. 0. 17437.88618968]]
[[2005.59055467 0. 0.]
[0. 18663.25799622 0.]
[0. 0. 22269.12377191]]

Implemented GMM | Weights
[0.38842458 0.4345021 0.17049438]

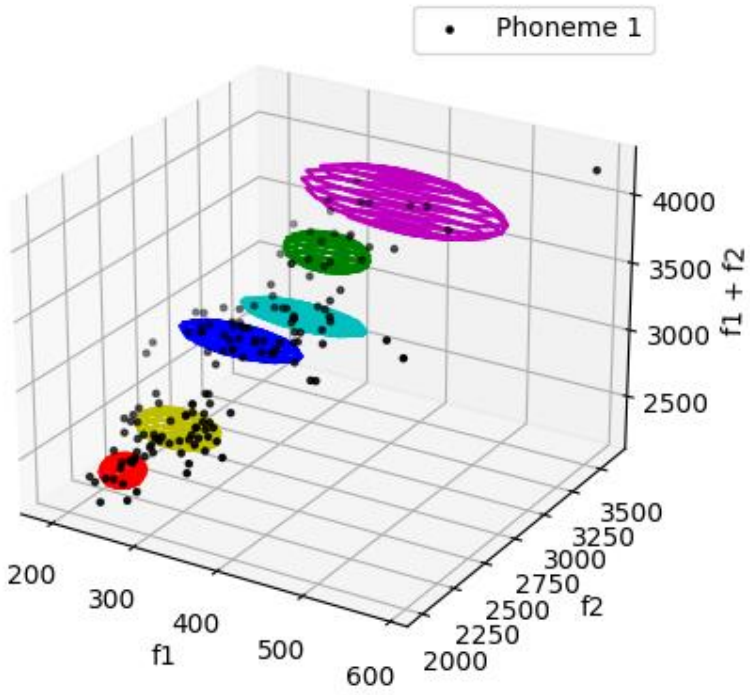

Figure 12: MoG with k = 6 on phoneme 1

Implemented GMM | Mean values
[244.45866 2140.6216 2385.08]
[333.50128 3163.198 3496.6992]
[292.61597 2740.9646 3033.5806]
[349.4483 2870.9082 3220.3564]
[388.69406 3415.734 3804.428]
[280.4988 2340.517 2621.0159]

Implemented GMM | Covariances
[[270.14972124 0. 0.]
[0. 5022.78486844 0.]
[0. 0. 5227.21284842]]
[[1312.33117121 0. 0.]
[0. 6459.18657646 0.]
[0. 0. 7103.38090445]]
[[2754.80111239 0. 0.]
[0. 4190.18003416 0.]
[0. 0. 4236.29584768]]
[[2845.21165407 0. 0.]
[0. 3782.1609862 0.]
[0. 0. 2461.5022325]]
[[7681.61743394 0. 0.]
[0. 7649.4365127 0.]

```
 [    0.           0.        22552.86770625]]
[[1215.70975542    0.           0.        ]
 [   0.        6222.79005316    0.        ]
 [   0.           0.        6527.87813227]]
```

Implemented GMM | Weights
[0.1215731 0.11220152 0.2532429 0.14130365 0.0590368 0.31264204]