



Assignment 2: Clustering and MoG

Student Name: Luca J. Ricagni

Student Number: 200894968

Module Name: Machine Learning

Module Code: ECS708U/ECS708P

Due Date: 09/12/2020

Table of Contents

MoG Modelling using the EM Algorithm	3
Task 1	3
Task 2	4
Task 3	12
Task 4	15
Task 5	18

MoG Modelling using the EM Algorithm

Task 1

This task was to produce a plot of F1 against F2. In order to accomplish this a 2D matrix X_{full} was created within *task_1.py*, containing all the values of F1 in the first column and all the values of F2 in the second column. This was done using the code shown in Figure 1.

```
for i in range(0, len(f1)):
    X_full[i, 0] = f1[i]
    X_full[i, 1] = f2[i]
```

Figure 1 – Code to add all values of F1 and F2 to a 2D matrix

When the code was run with this new matrix the plot of F1 against F2 was produced, shown in Figure 2. As can be seen this plot contains all phonemes of F1 and F2, represented by different coloured points. There is a clear, if rough, distinction between the different phonemes, as each phoneme can be seen to group in a different part of the plot.

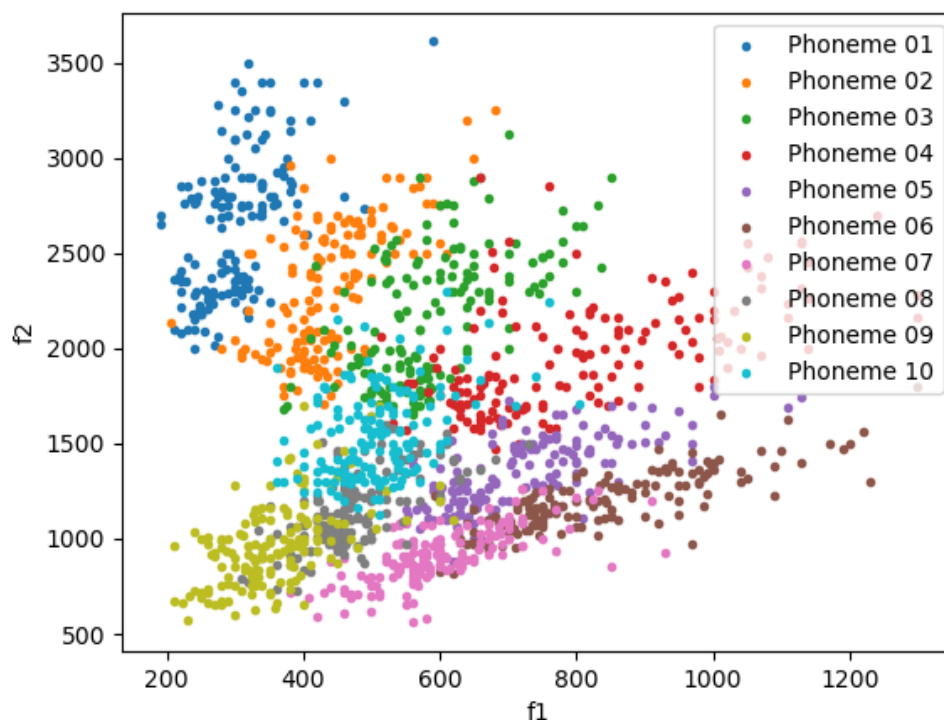


Figure 2 – Plot of F1 against F2 for all phonemes shown in different colours

Having achieved this the next stage was to isolate a single group, or phoneme, of this dataset. The code to do this can be found in Figure 3. This code isolates phoneme 1, as p_id is set to 1.

```
count = 0
for i in range(0, len(X_full)):
    if phoneme_id[i] == p_id:
        X_phoneme_1[count, 0] = X_full[i, 0]
        X_phoneme_1[count, 1] = X_full[i, 1]
        count = count + 1
```

Figure 3 – Code to isolate and store values in X_{full} which are of phoneme 1

In order to do this both the iterator i in the for loop, and an additional *count* variable must be used, this is because the arrays $X_phoneme_1$ and X_full are not the same size, and the values put into the former are selected.

When the code for this was ran the plot shown in Figure 4 was produced.

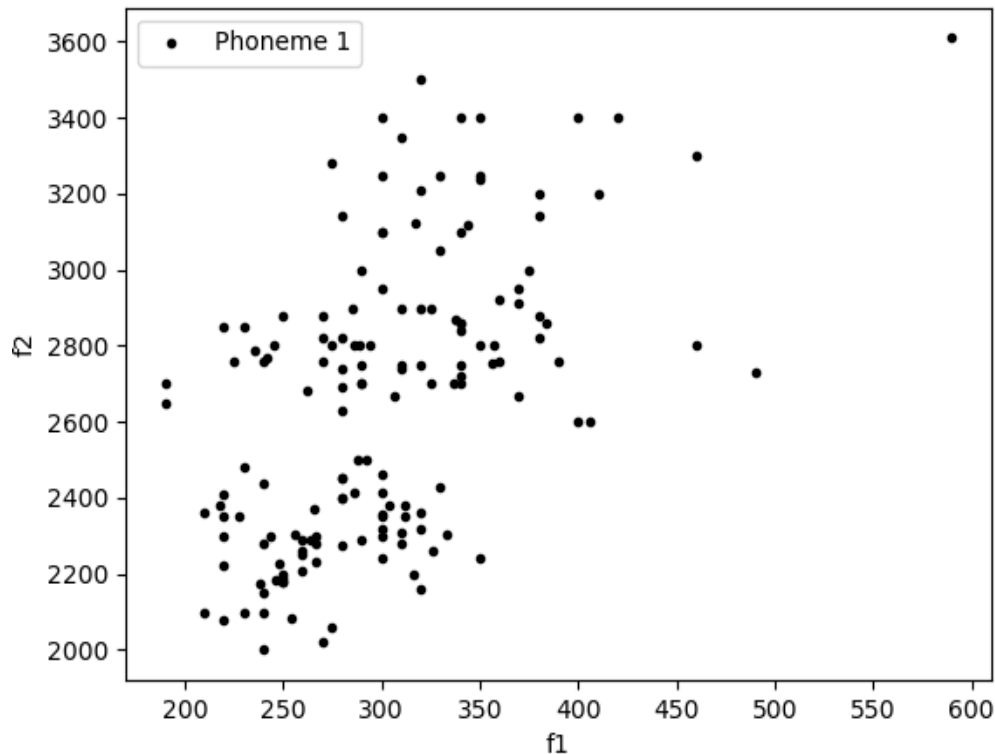


Figure 4 – Plot of F1 against F2 for phoneme 1

This can visually be seen to have been successful as the shape of the data in Figure 4 is the same shape as the data for phoneme 1 (the dark blue dots) in Figure 2.

Task 2

This task was to train the data for phonemes 1 and 2 using Multiple of Gaussians (MoGs). Firstly, the code in Figures 1 & 3 was copied into *task_2.py*, however the code from Figure 3 was adapted slightly to store the data for any given phoneme in an array which was exactly the right length, and which would use the value of p_id as the phoneme number. Figure 5 shows this adapted version of the code.

```
X_phoneme = np.zeros((np.sum(phoneme_id==p_id), 2))
count = 0
for i in range(0, len(X_full)):
    if phoneme_id[i] == p_id:
        X_phoneme[count, 0] = X_full[i, 0]
        X_phoneme[count, 1] = X_full[i, 1]
        count = count + 1
```

Figure 5 – Code to store the values from F1 and F2 for any given phoneme

Having now correctly implemented this code *task_2.py* can be run. For the first run p_id (the phoneme being used for training) was set to 1, and k (the number of Gaussians to use) was set to 3. The result of this run can be found in Figure 6, and the variables μ (the means of each Gaussian), Σ (the covariance matrices) and p (the weights) were saved for later use.

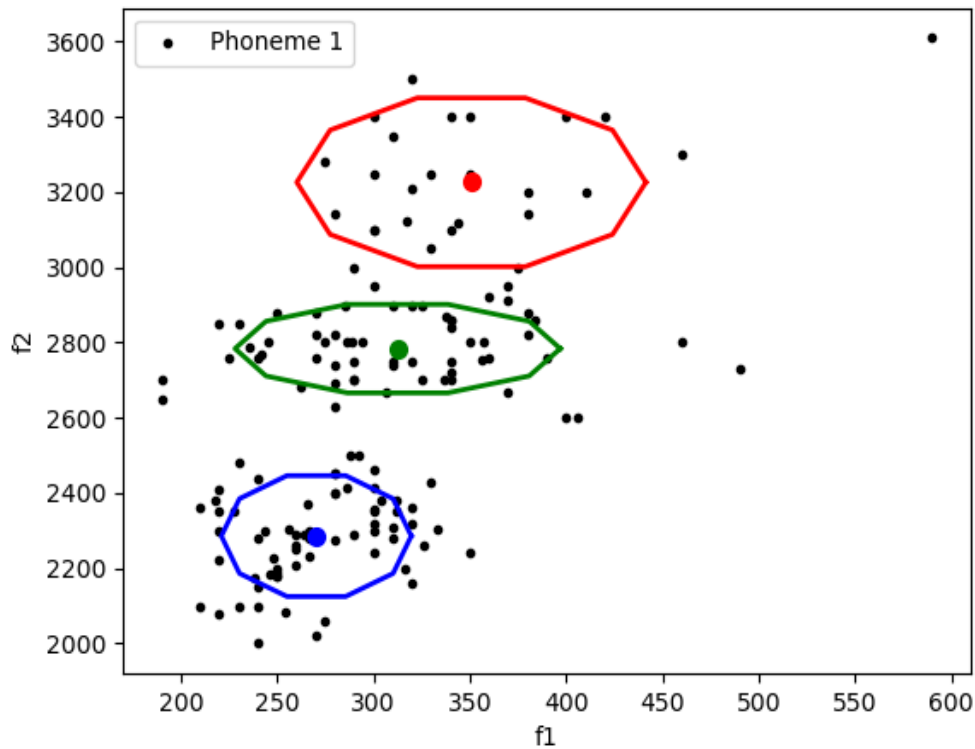


Figure 6 – Plot showing a typical case of the learnt models on phoneme 1 using 3 Gaussians

The different coloured lines in Figure 6 represent each of the trained Gaussians, with the dots in the middle showing the mean of each one. As can be seen the data has been classified into 3 distinct

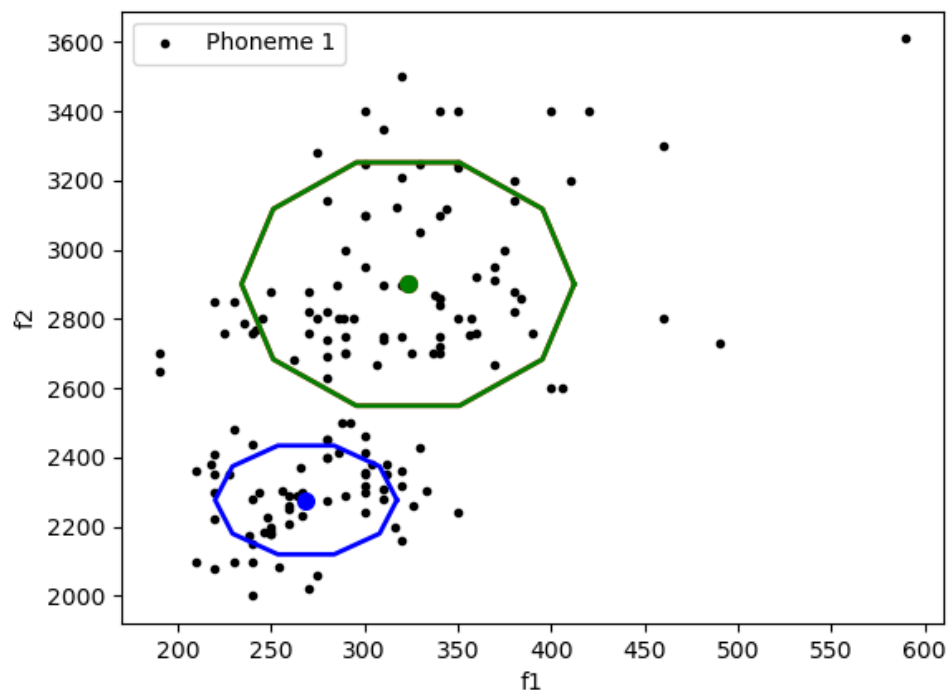


Figure 7 - Plot showing a rare case where 2 Gaussians merged on phoneme 1 using 3 Gaussians

classes, as there is no overlap between the Gaussians. However, this is not a perfect model, as a substantial number of data points are not encompassed within one of the trained Gaussians.

While Figure 6 shows a typical case of the learnt model for 3 Gaussians it is not the only model that was generated. Due to the random initialisation of the mean of each Gaussian some different cases emerge. One example of this is shown in Figure 7, where two of the Gaussians have merged into 1. Another example is shown in Figure 8, where the Gaussians have quite a bit of overlap and are generally lower down in the plot. A third example can be seen in Figure 9, where two of the Gaussians have moved quite low down and become quite small.

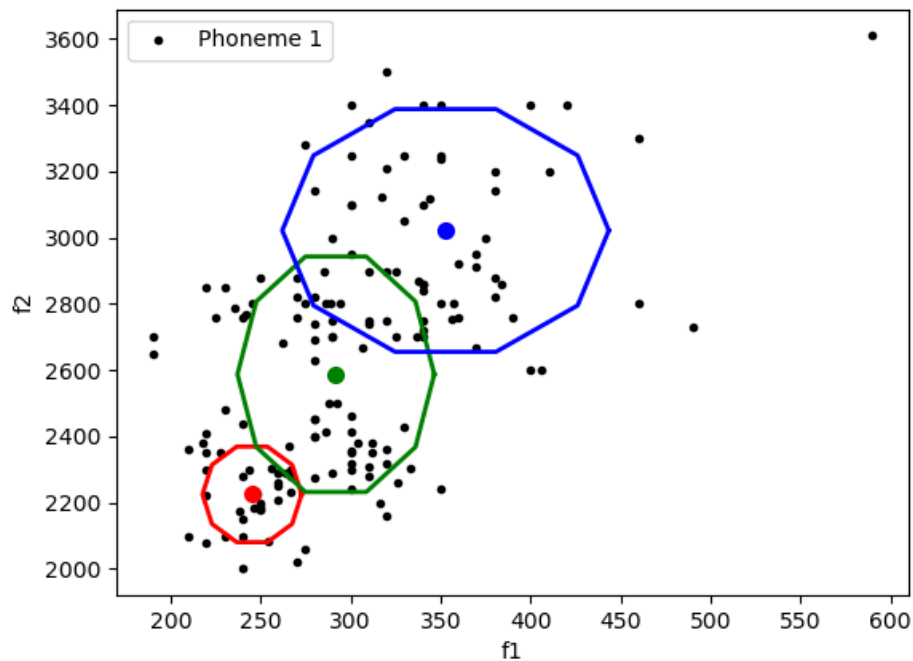


Figure 8 - Plot showing a rare case on phoneme 1 using 3 Gaussians where they overlap

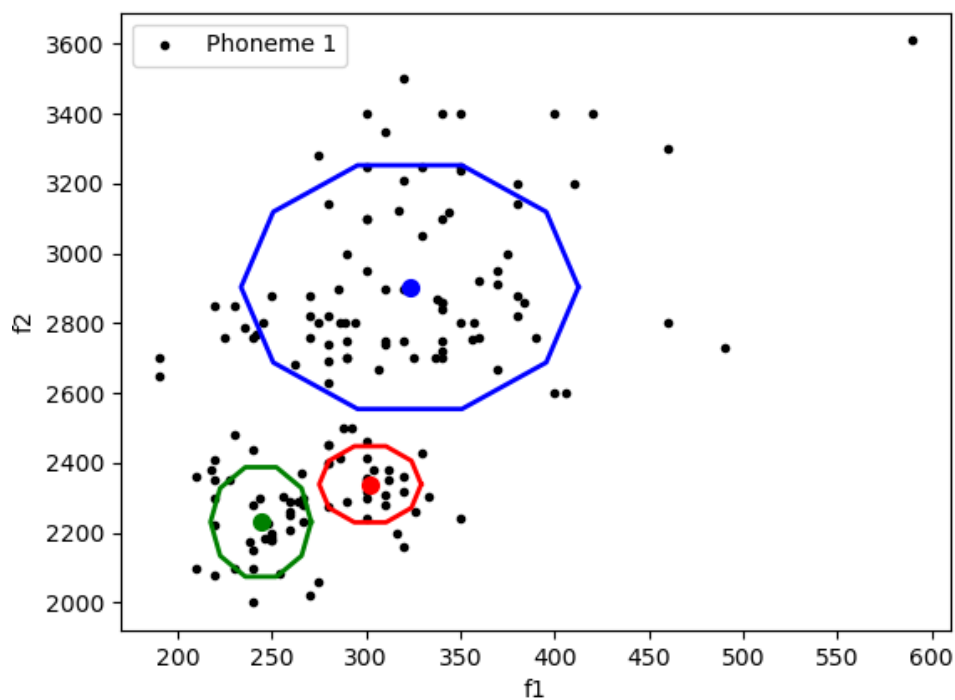


Figure 9 - Plot showing a rare case on phoneme 1 using 3 Gaussians

The code was then run again on the first phoneme using 6 Gaussians ($k = 6$). Figure 10 shows a typical case of the learnt model on phoneme 1 using 6 Gaussians, there is a good representation of the data and minimal overlap between Gaussians.

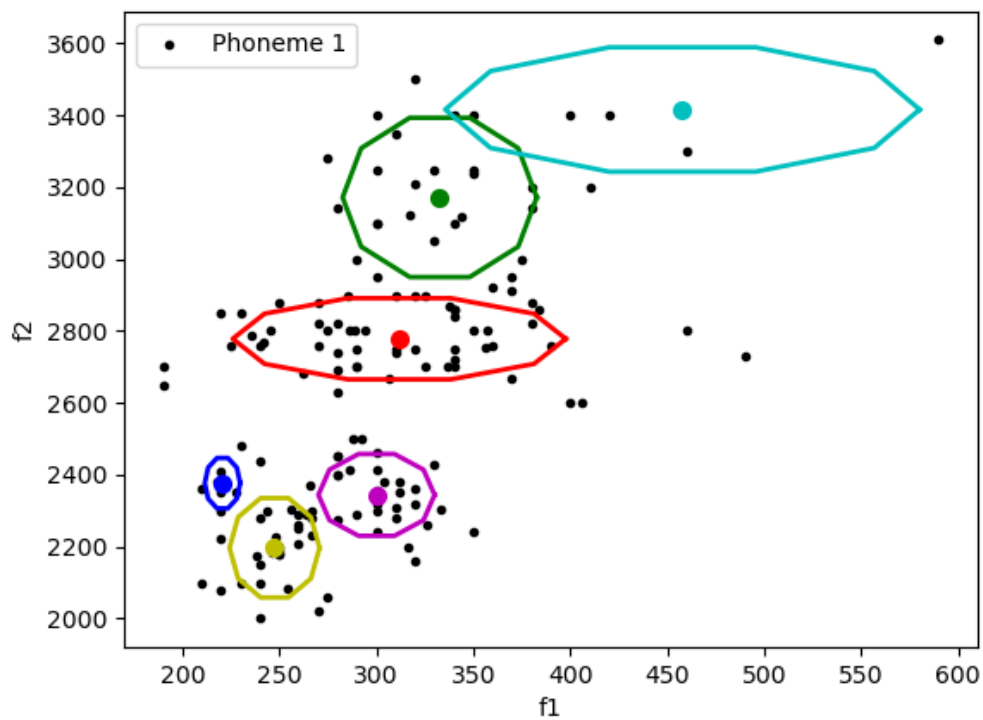


Figure 10 - Plot showing a typical case of the learnt models on phoneme 1 using 6 Gaussians

Once again however there are some rare cases which are generated. Figure 11 shows a rare case where the two centre Gaussians (red and teal) have a great deal of overlap.

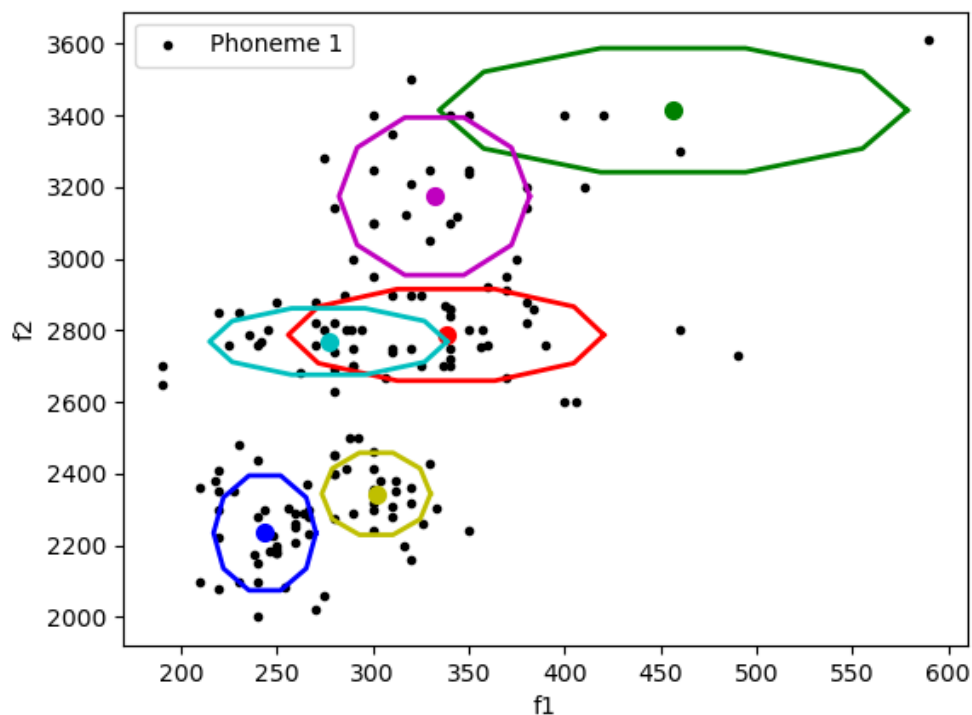


Figure 11 - Plot showing a rare case with lots of overlap on phoneme 1 using 6 Gaussians

Figure 12 shows another rare case with a great deal of overlap across five of the Gaussians, where one has become vertically quite narrow. Figure 13 shows a third rare case, again with a lot of overlap, where one Gaussian (yellow) is almost entirely contained within another (blue).

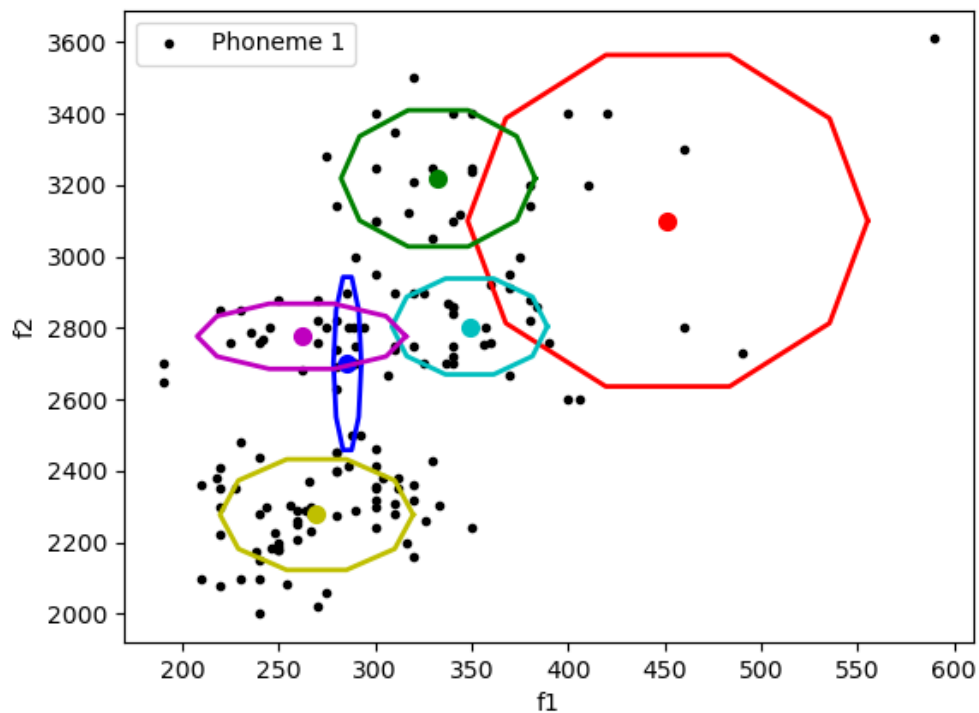


Figure 12 - Plot showing a rare case with multiple Gaussians overlapping on phoneme 1 using 6 Gaussians

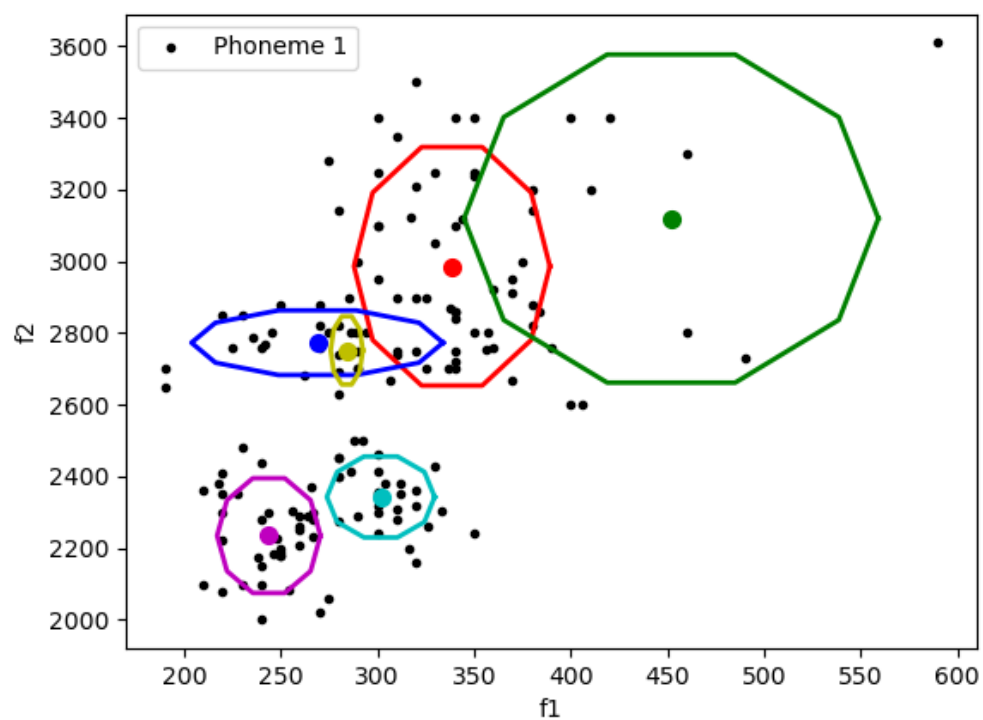


Figure 13 - Plot showing a rare case with one Gaussian contained within another on phoneme 1 using 6 Gaussians

Having completed this the next step was to repeat these experiments of using 3 and 6 Gaussians on phoneme 2. In order to do this the variable p_id was simply changed to 2 and the code was rerun. Figure 14 shows a typical case of the learnt models as again there is relatively good coverage and no overlap.

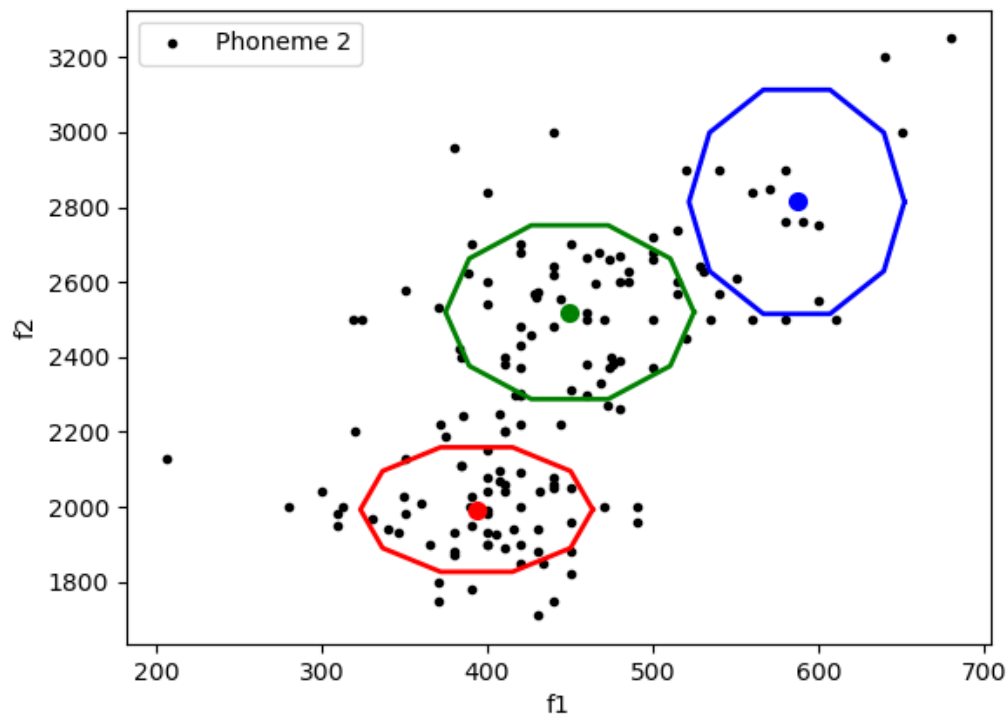


Figure 14 - Plot showing a typical case of the learnt models on phoneme 2 using 3 Gaussians

Once again there were some rare cases which emerged during testing. Figure 15 shows an example where the three Gaussians are all perpendicular to each other with a great deal of overlap.

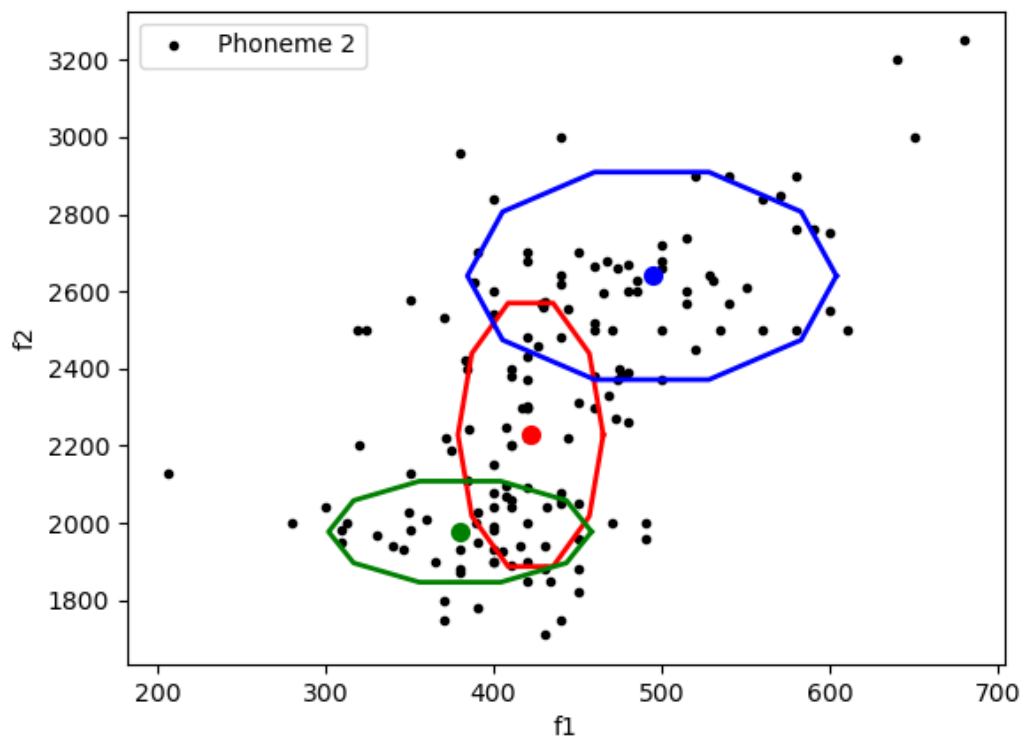


Figure 15 - Plot showing a rare case with the Gaussians perpendicular to one another on phoneme 2 using 3 Gaussians

Figure 16 shows another example where one Gaussian has been completely enveloped within another.

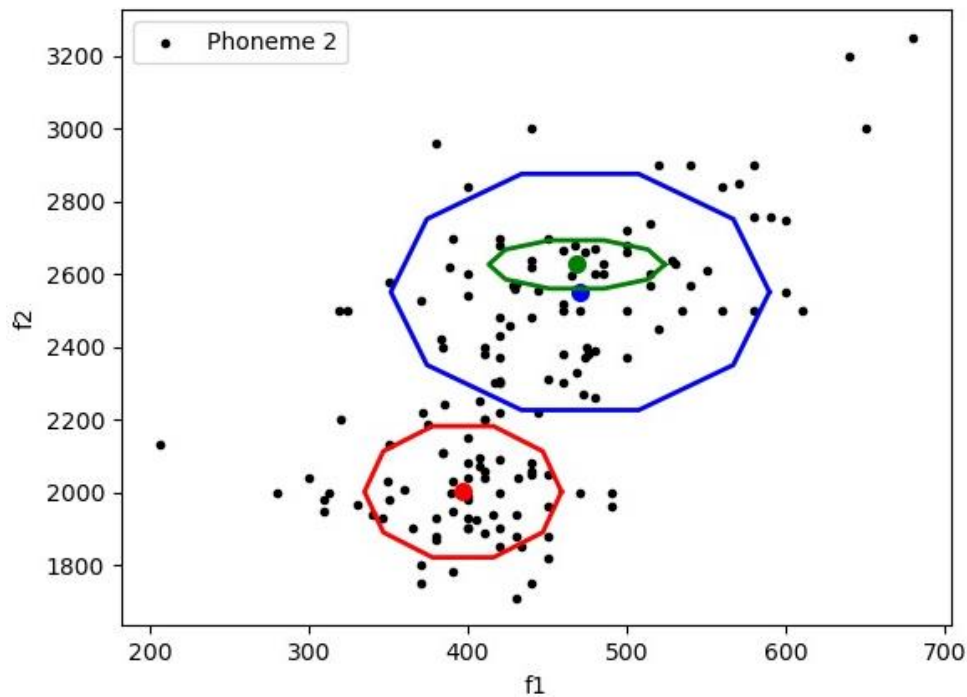


Figure 16 - Plot showing a rare case with one Gaussian within another on phoneme 2 using 3 Gaussians

The code was then run again on the second phoneme using 6 Gaussians ($k = 6$). Figure 17 shows a typical example of the learnt model in this case. It provides good coverage and although there is some overlap it is minimal.

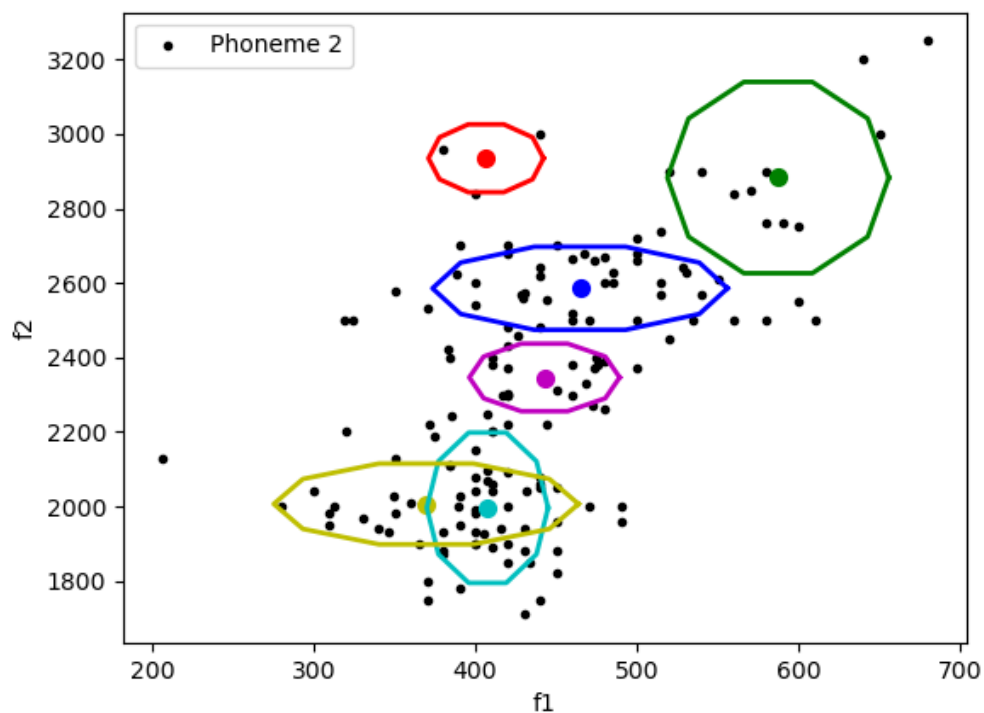


Figure 17 - Plot showing a typical case of the learnt models on phoneme 2 using 6 Gaussians

Once again there are some rare cases which sometimes occur. Figure 18 shows an example where two Gaussians (the purple one located towards the bottom of the plot) have merged and completely overlap.

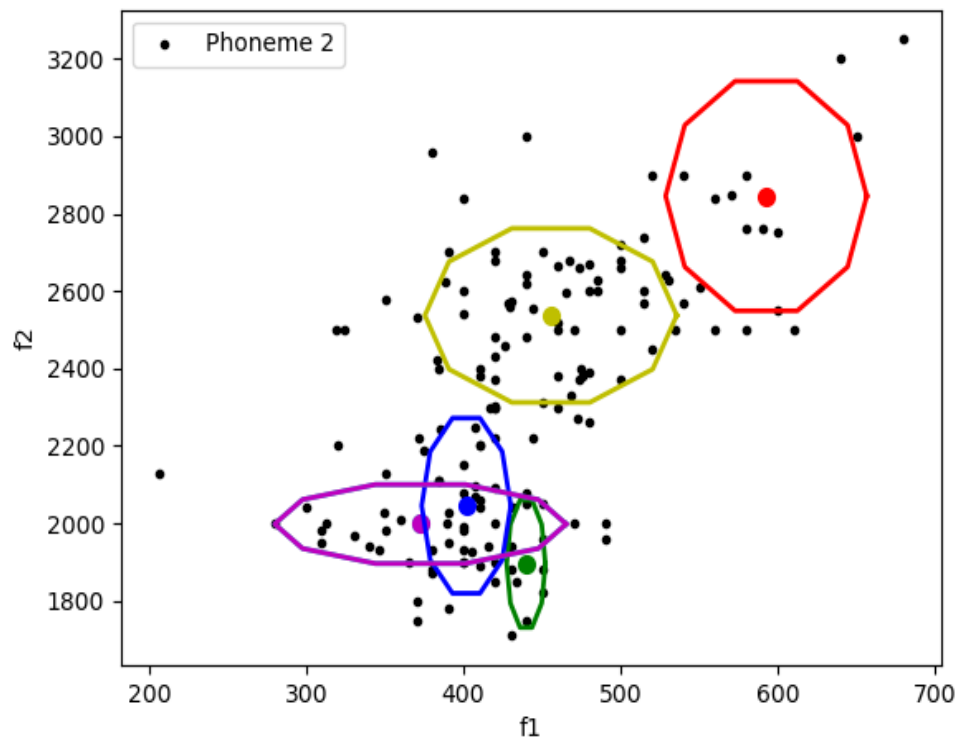


Figure 18 - Plot showing a rare case where two Gaussians have merged on phoneme 2 using 6 Gaussians

Figure 19 shows an example where there is again a great deal of overlap, with one Gaussian wholly contained within another.

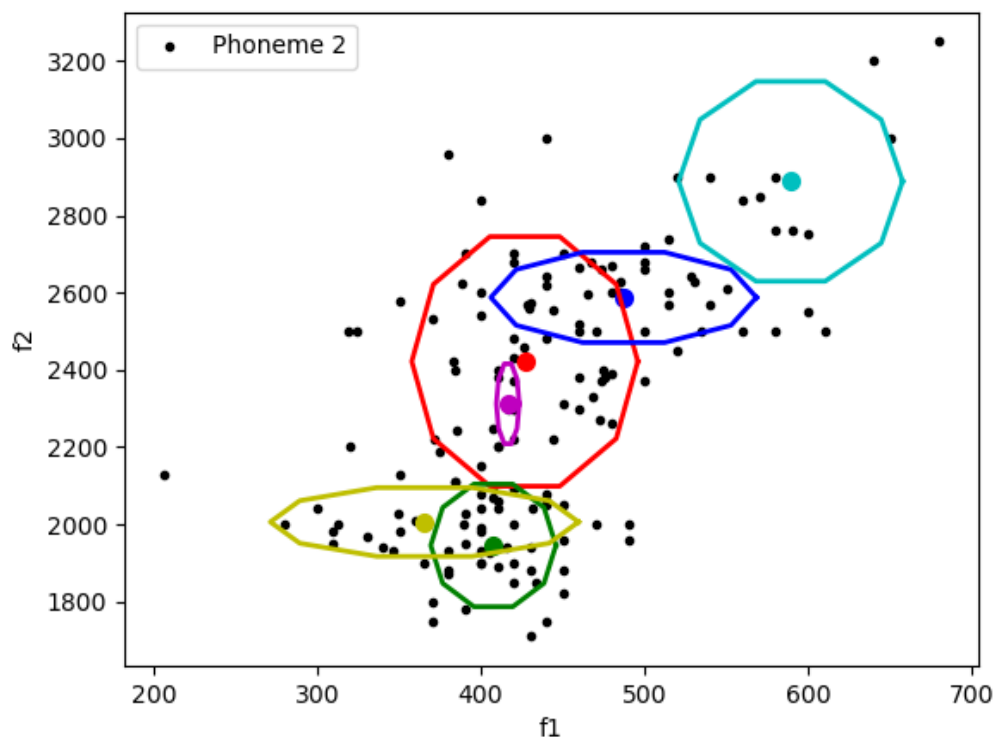


Figure 19 - Plot showing a rare case with one Gaussian within another on phoneme 2 using 6 Gaussians

Figure 20 shows an example again with a great deal of overlap towards the bottom of the plot.

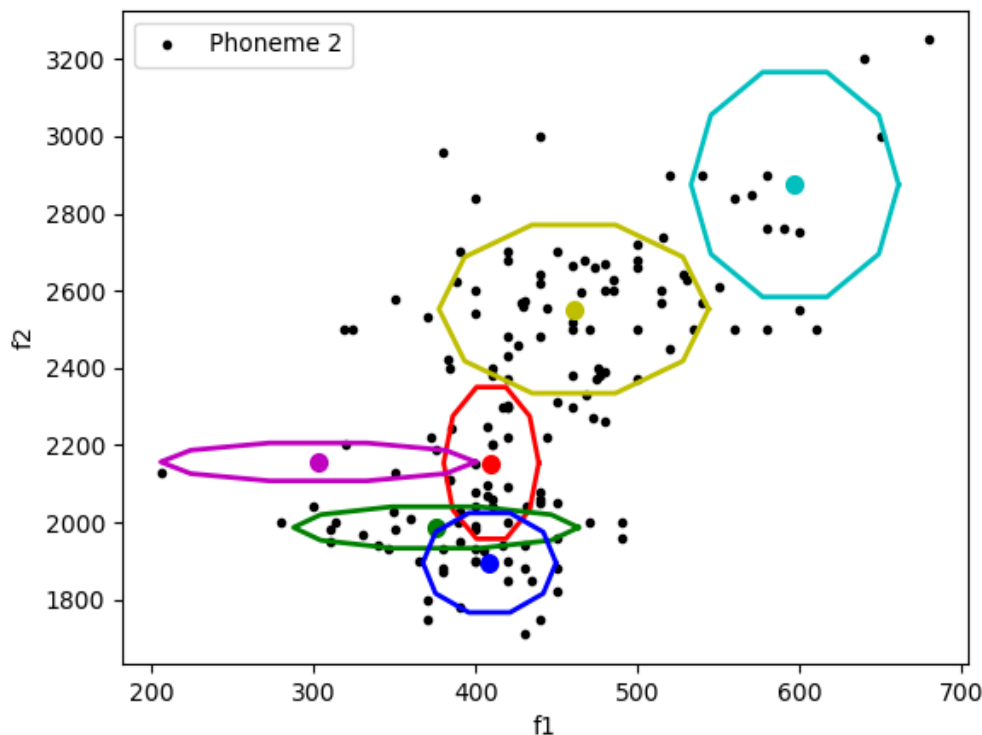


Figure 20 - Plot showing a rare case with a great deal of overlap on phoneme 2 using 6 Gaussians

Task 3

This task was to take the trained MoGs from the previous task and build a classifier to distinguish between phonemes 1 & 2, which was to be done in *task_3.py*. For this the Maximum Likelihood criterion must be used.

Firstly, the array *X_full*, containing all values of F1 and F2, must be created. This was done as before using the code shown in Figure 1. Secondly an array *X_phonemes_1_2*, containing all the values of *X_full* which belong to phonemes 1 & 2, was made using the code shown in Figure 21. This code works by creating an array of zeros which has 2 columns and a number of rows equal to the total number of values belonging to phonemes 1 & 2. Then it initialises two count variables *j* and *l*, the former to 0 and the latter to the end of the first group, this will mean that all values of phoneme 1

```
X_phonemes_1_2 = np.zeros((np.sum(phoneme_id == 1) + np.sum(phoneme_id == 2), 2))
j = 0
l = np.sum(phoneme_id == 1)
for i in range(0, len(X_full)):
    if phoneme_id[i] == 1:
        X_phonemes_1_2[j, 0] = X_full[i, 0]
        X_phonemes_1_2[j, 1] = X_full[i, 1]
        j += 1
    elif phoneme_id[i] == 2:
        X_phonemes_1_2[l, 0] = X_full[i, 0]
        X_phonemes_1_2[l, 1] = X_full[i, 1]
        l += 1
```

Figure 21 – Code to create array containing all values belonging to phonemes 1 & 2 from the sets F1 & F2

will be grouped together at the start and all values of phoneme 2 group at the end. The loop then iterates through the length of *X_full*, checks if a given value belongs to phoneme 1 or 2 and adds it to *X_phonemes_1_2* if it is, and ignores it if it isn't.

Next an array containing the ground truths, i.e. which phoneme each point belongs to, is created to match *X_phonemes_1_2*. This is done by first initialising an array the same length as *X_phonemes_1_2*, then using two loops to add the actual phoneme each point belongs to. The code to do this can be found in Figure 22.

```
ground_truth = np.zeros(len(X_phonemes_1_2))
for i in range(np.sum(phoneme_id == 1)):
    ground_truth[i] = 1

for i in range(np.sum(phoneme_id == 2)):
    ground_truth[i + np.sum(phoneme_id == 1)] = 2
```

Figure 22 – Code to create array of ground truths

Next the data files created at the end of task 2 when running *task_2.py* had to be imported. This was done using the *np.load* function, followed by converting the data type to a list using the *np.ndarray.tolist* function. This code can be found in Figure 23, and for simplicity sake all 4 files were imported at once.

```
# Load data files generated in task_2.py
phoneme1_gaussian3 = np.load('data/GMM_params_phoneme_01_k_03.npy', allow_pickle=True)
phoneme2_gaussian3 = np.load('data/GMM_params_phoneme_02_k_03.npy', allow_pickle=True)
phoneme1_gaussian6 = np.load('data/GMM_params_phoneme_01_k_06.npy', allow_pickle=True)
phoneme2_gaussian6 = np.load('data/GMM_params_phoneme_02_k_06.npy', allow_pickle=True)

# Convert data files to lists
phoneme1_gaussian3 = np.ndarray.tolist(phoneme1_gaussian3)
phoneme2_gaussian3 = np.ndarray.tolist(phoneme2_gaussian3)
phoneme1_gaussian6 = np.ndarray.tolist(phoneme1_gaussian6)
phoneme2_gaussian6 = np.ndarray.tolist(phoneme2_gaussian6)
```

Figure 23 – Code to import all data files and convert them to lists

Having successfully imported the data files, the data must be put through the *get_predictions* function. This function takes the dataset, as well as the *mu*, *s* and *p* values of the Gaussians trained in *task_2.py* and returns an array of probabilities that a data point belongs to a given phoneme. Each row corresponds to one data point, and each column to a Gaussian, therefore when using 3 Gaussians there will be 3 columns and when using 6 Gaussians there will be 6 columns. The code for this is shown in Figure 24.

```
# Get predictions
probabilities_phoneme1_gaussian3 = get_predictions(phoneme1_gaussian3['mu'], phoneme1_gaussian3['s'], phoneme1_gaussian3['p'], X_phonemes_1_2)
probabilities_phoneme2_gaussian3 = get_predictions(phoneme2_gaussian3['mu'], phoneme2_gaussian3['s'], phoneme2_gaussian3['p'], X_phonemes_1_2)
probabilities_phoneme1_gaussian6 = get_predictions(phoneme1_gaussian6['mu'], phoneme1_gaussian6['s'], phoneme1_gaussian6['p'], X_phonemes_1_2)
probabilities_phoneme2_gaussian6 = get_predictions(phoneme2_gaussian6['mu'], phoneme2_gaussian6['s'], phoneme2_gaussian6['p'], X_phonemes_1_2)
```

Figure 24 – Code to calculate the probability of each point being within a given Gaussian

Then each row of these probabilities is summed, hence calculating the overall probability that a given point is contained within the area represented by the combination of Gaussians and therefore belongs to a given phoneme. The code for this can be found within Figure 25.

```
# Get best prediction using sum value
probabilities_sum_phoneme1_gaussian3 = np.sum(probabilities_phoneme1_gaussian3, axis=1)
probabilities_sum_phoneme2_gaussian3 = np.sum(probabilities_phoneme2_gaussian3, axis=1)
probabilities_sum_phoneme1_gaussian6 = np.sum(probabilities_phoneme1_gaussian6, axis=1)
probabilities_sum_phoneme2_gaussian6 = np.sum(probabilities_phoneme2_gaussian6, axis=1)
```

Figure 25 - Code to calculate sum of probabilities for each row

Using these summed values, a prediction can now be made as to which phoneme a given data point belongs to. To do this the summed probabilities for phoneme 1 & 2 are compared and the highest one is taken. Once again this was done for both 3 and 6 Gaussians, the code is shown in Figure 26.

```
# Loop through and use the highest probability to determine whether a point is phoneme 1 or 2
# Get the predicted phoneme of a point for 3 gaussian
predictions_gaussian3 = np.zeros(len(X_phonemes_1_2))
for i in range(0, len(predictions_gaussian3)):
    if probabilities_sum_phoneme1_gaussian3[i] > probabilities_sum_phoneme2_gaussian3[i]:
        predictions_gaussian3[i] = 1
    else:
        predictions_gaussian3[i] = 2

# Get the predicted phoneme of a point for 6 gaussian
predictions_gaussian6 = np.zeros(len(X_phonemes_1_2))
for i in range(0, len(predictions_gaussian6)):
    if probabilities_sum_phoneme1_gaussian6[i] > probabilities_sum_phoneme2_gaussian6[i]:
        predictions_gaussian6[i] = 1
    else:
        predictions_gaussian6[i] = 2
```

Figure 26 - Code to predict which phoneme each point belongs to, based of which probability is higher

These predictions are then compared with the ground truth of their corresponding point, and the number of correct predictions is then counted. The code is shown in Figure 27.

```
# Check prediction against ground truth
correct_count_3gaussian = 0
for i in range(0, len(predictions_gaussian3)):
    if predictions_gaussian3[i] == ground_truth[i]:
        correct_count_3gaussian += 1

correct_count_6gaussian = 0
for i in range(0, len(predictions_gaussian6)):
    if predictions_gaussian6[i] == ground_truth[i]:
        correct_count_6gaussian += 1
```

Figure 27 - Code to compare predictions to ground truth and tally up the number of correct predictions

The count of correct predictions is then used to calculate and print the accuracy of each GMM as a percentage. The code can be found in Figure 28.

```
# Calculate accuracy as a percentage
accuracy_3gaussian = (correct_count_3gaussian/len(predictions_gaussian3)) * 100
accuracy_6gaussian = (correct_count_6gaussian/len(predictions_gaussian6)) * 100

# Print accuracy for both 3 & 6 gaussian
print('Accuracy using GMMs with {} components: {:.2f}%'.format(3, accuracy_3gaussian))
print('Accuracy using GMMs with {} components: {:.2f}%'.format(6, accuracy_6gaussian))
```

Figure 28 - Code to calculate and print accuracy of each GMM as a percentage

Having now filled out *task_3.py* with the correct code it can now be run. After running the code, it outputs the accuracy measurements shown in Figure 29. As can be seen both GMMs have a very high accuracy measurement, both over 95%. Using 6 Gaussians instead of 3 to model the data seems to give a slight improvement in accuracy, as would be expected due to more components covering a larger area in more detail. However, depending on the application, the extra computational overhead may not be worth the marginal increase to accuracy.

Task 4

```
Accuracy using GMMs with 3 components: 95.07%
Accuracy using GMMs with 6 components: 96.38%
```

Figure 29 - Accuracy measurements of each GMM

This task was to create a grid of points which spans the area of the search space, that is the area between the minimum and maximum values of F1 and F2. Then classify each of these points using the classifiers trained in task 2.

In order to accomplish this some code from *task_3.py* was copied into *task_4.py*, namely the code to create the arrays *X_full* and *X_phonemes_1_2*, as well as to import the datasets, shown in Figures 1, 21 and 23 respectively.

Then the *grid* array, which would contain every point within the search space, was created and filled using the code shown in Figure 30. This code works by having two nested for loops which iterate between the minimum and maximum values of F1 and F2 and using an *index* variable which is incremented after every iteration of the inner loop.

```
# Create grid (a 2 column array)
# to contain coordinates of all points
grid = np.zeros((N_f1*N_f2, 2))

# Fill grid with coordinates of all points
index = 0
for i in range(min_f1, max_f1):
    for j in range(min_f2, max_f2):
        grid[index, 0] = i
        grid[index, 1] = j
        index += 1
```

Figure 30 - Code to create and populate grid array

Having created this array, it can now be passed to the *get_predictions* function by simply changing the last argument to be *grid* rather than *X_phonemes_1_2*. This is shown in Figure 31.

```
# Get predictions
probabilities_phoneme1_gaussian3 = get_predictions(phoneme1_gaussian3['mu'], phoneme1_gaussian3['s'], phoneme1_gaussian3['p'], grid)
probabilities_phoneme2_gaussian3 = get_predictions(phoneme2_gaussian3['mu'], phoneme2_gaussian3['s'], phoneme2_gaussian3['p'], grid)
probabilities_phoneme1_gaussian6 = get_predictions(phoneme1_gaussian6['mu'], phoneme1_gaussian6['s'], phoneme1_gaussian6['p'], grid)
probabilities_phoneme2_gaussian6 = get_predictions(phoneme2_gaussian6['mu'], phoneme2_gaussian6['s'], phoneme2_gaussian6['p'], grid)
```

Figure 31 - Code to get predictions for the grid array

As before the probabilities were then summed using the code shown in Figure 26. Then the array *M* was created and populated with the phoneme of the highest probability using 3 Gaussians as before, shown in Figure 32.

```
# Create M
M = np.zeros((N_f1, N_f2))

# Take best prediction for 3 gaussian
for i in range (0, N_f1 * N_f2):
    x = int(i / N_f2)
    y = int(i % N_f2)
    if probabilities_sum_phoneme1_gaussian3[i] > probabilities_sum_phoneme2_gaussian3[i]:
        M[x, y] = 0.0
    else:
        M[x, y] = 1.0
```

Figure 32 - Code to compare which phoneme has the highest probability for 3 Gaussians and then assign *M* to the correct value

This was also done for 6 Gaussians by simply swapping the variables, shown in Figure 33. The reader should note that the code in Figures 32 and 33 cannot be used at the same time, as they both write to *M*.

```
# Take best prediction for 6 gaussian
for i in range (0, N_f1 * N_f2):
    x = int(i / N_f2)
    y = int(i % N_f2)
    if probabilities_sum_phoneme1_gaussian6[i] > probabilities_sum_phoneme2_gaussian6[i]:
        M[x, y] = 0.0
    else:
        M[x, y] = 1.0
```

Figure 33 - Code to compare which phoneme has the highest probability for 6 Gaussians and then assign *M* to the correct value

When the code was run using the 3 Gaussian GMM (i.e. the code in Figure 32) a graph of the predicted values of each point is produced, this graph can be found in Figure 34. The red dots show the data points that belong to phoneme 1, while the green dots belong to phoneme 2. The state space of the graph is then divided into two colours. The purple represents the points which the classifier thinks would belong to phoneme 1 based off the data points it was trained on, the yellow shows the area classified as phoneme 2. As can be seen the classifier has done a good job in classifying the data points, as there is a clear line running through the middle which bends appropriately to follow the data and fits the vast majority of the data well.

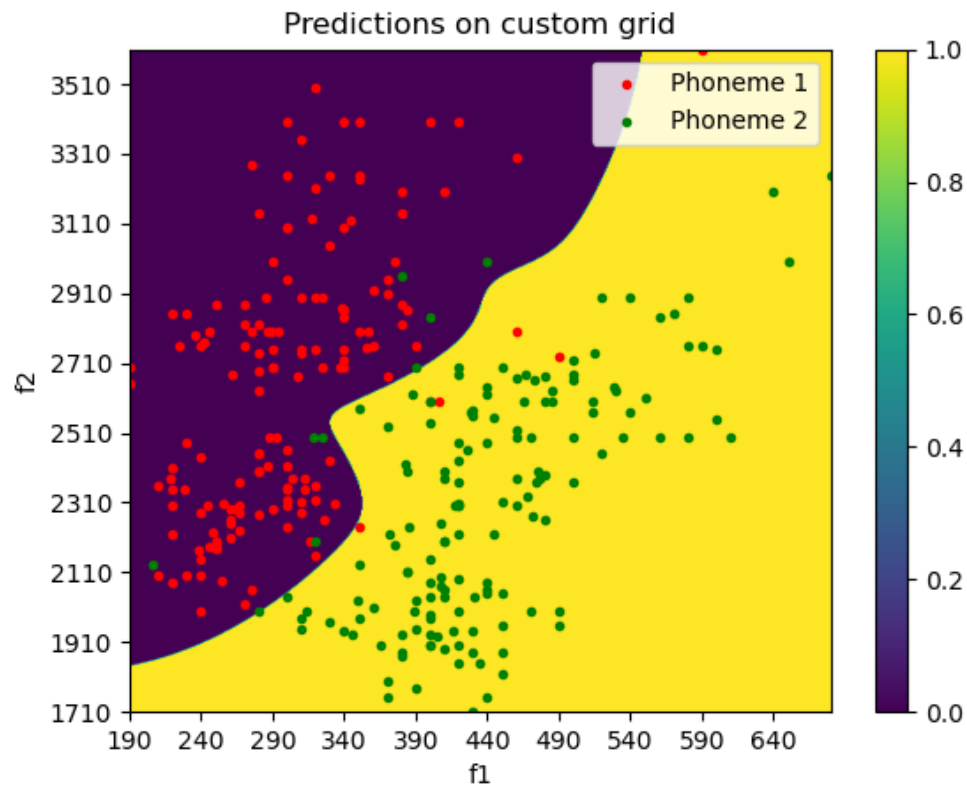


Figure 34 - Graph showing the predicted phonemes of each point in the state space, 3 Gaussian model

This code was also run on the 6 Gaussian model for comparison purposes, the graphical output of this can be found in Figure 35.

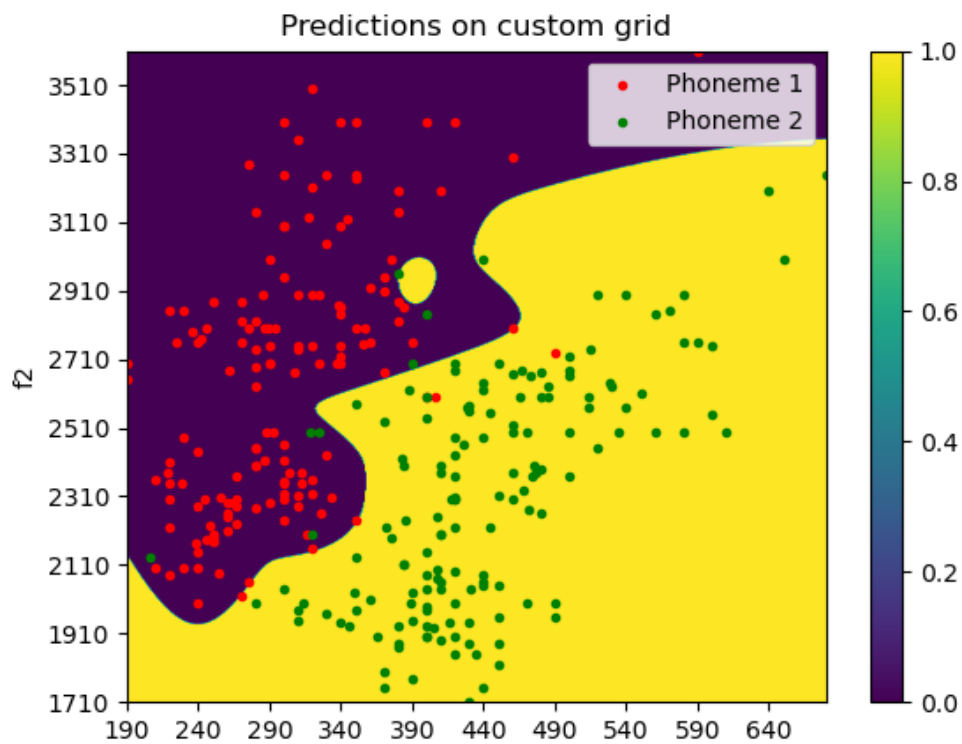


Figure 35 - Graph showing the predicted phonemes of each point in the state space, 6 Gaussian model

While this arguably fits the data better as it has a higher accuracy measurement, as there are fewer incorrect border cases, this is quite clearly a case of overfitting. This is partly because of the much more extreme border between the two phonemes, but also due to the area classified as phoneme 2 which is separate to the main area.

Therefore, while using 6 Gaussians may give a higher accuracy measurement, the overfitting this causes and the minimal gain in accuracy makes the use of the 3 Gaussian model more desirable.

Task 5

This task was to fit an MoG with full covariance matrices to a new dataset, adding a third column of $F1 + F2$ to X_{full} . The code for this can be found in Figure 36.

```
for i in range(0, len(f1)):
    X_full[i, 0] = f1[i]
    X_full[i, 1] = f2[i]
    X_full[i, 2] = f1[i] + f2[i]
```

Figure 36 - Code to make dataset with new column

Then a second array, $X_{phoneme}$, was created using the values for X_{full} which belong to a given phoneme for all three columns, the code for which is shown in Figure 37.

```
X_phoneme = np.zeros((np.sum(phoneme_id==p_id), 3))
count = 0
for i in range(0, len(X_full)):
    if phoneme_id[i] == p_id:
        X_phoneme[count, 0] = X_full[i, 0]
        X_phoneme[count, 1] = X_full[i, 1]
        X_phoneme[count, 2] = X_full[i, 2]
        count = count + 1
```

Figure 37 - Code create array only containing data belonging to a given phoneme

When these things were implemented to *task_5.py* and the code was run there were consistent warning messages given, saying that casting an imaginary number to a real one discards the imaginary part. This is likely due to the code computing the square root of a negative number, therefore creating an imaginary number, before trying to use it as a real number. However, as these were warnings, they did not cause the program to crash. The crash occurred when a divide by zero error occurred in line 29 of *get_predictions*. The exact iteration this transpired on cannot be pinned down, due to the inherent randomness in the starting locations and movement of the Gaussians. However, it seemed to occur mostly in the range of 30-80 iterations.

The divide by zero error was found to occur when trying to invert the covariance matrix, specifically when the determinant of the matrix turned out to be 0, this is known as a singularity. In other words a singularity is when a Gaussian overfits itself to a single point, causing the variance and thus the determinant of the covariance matrix, to be 0. An example of this occurring can be found in Figure 38. The right hand Gaussian has fit itself to only a singular point, thereby creating a singularity.

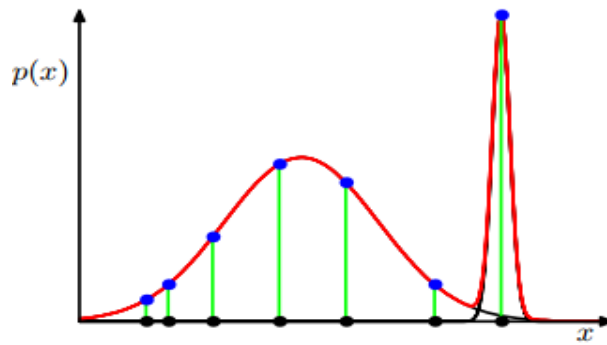


Figure 38 - Graph showing a plot of Gaussians with one as a singularity

<https://stats.stackexchange.com/questions/219302/singularity-issues-in-gaussian-mixture-model>

The first method to solve this was to create an identity matrix of the same dimensions as the covariance matrix (3x3), multiplying it by a very small number, then adding it to the covariance matrix. The code to do this can be found in Figure 39.

```
epsilon = np.identity(D) * 0.001
s[i] += epsilon
```

Figure 39 - Code to create identity matrix of small values then add it to the covariance matrix

This has the effect of incrementing the left-right diagonal of the covariance matrix by 0.001 every time it is updated. This ensures that the diagonal values will not be 0, therefore making the covariance matrix positive definite and consequently that it is invertible, removing the divide by 0 error. In the case where the diagonal of the covariance matrix was not 0 then this solution will just simply increment the values very slightly, which will have a negligible impact on accuracy.

Figure 40 shows an example output with this solution implemented. As can be seen the 3 Gaussians fit the data reasonably well with no overlap.

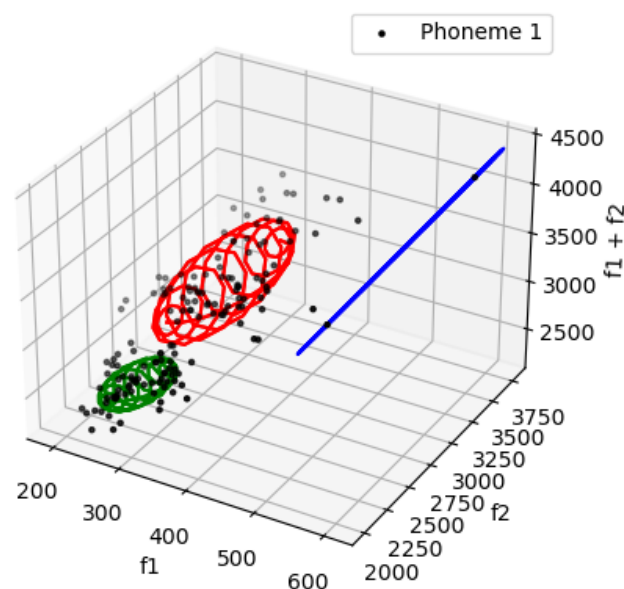


Figure 40 - Example plot showing trained the Gaussians and the data points, using method 1, phoneme 1 and 3 Gaussians

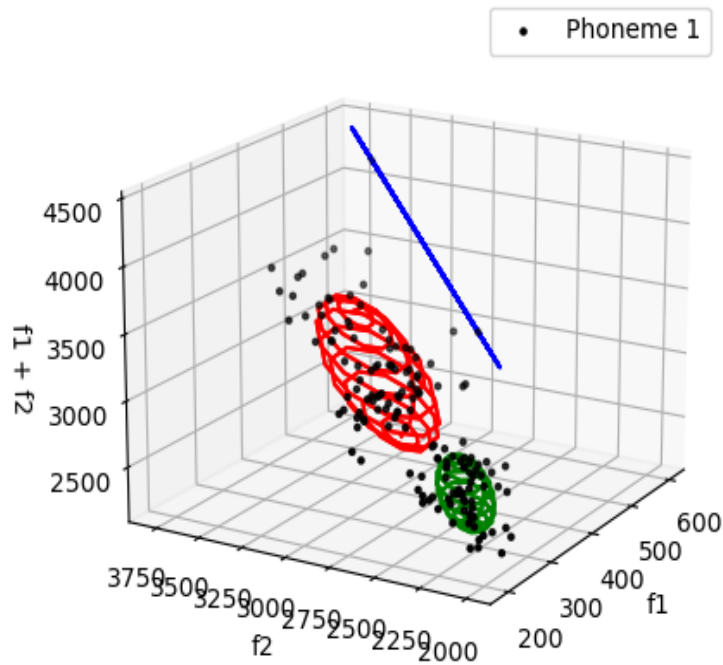


Figure 41 - Example plot showing trained the Gaussians and the data points from a different angle, using method 1, phoneme 1 and 3 Gaussians

Figure 41 shows a different angle of the same plot as Figure 40, again showing the models good fit of the data.

This method was also tested using 6 Gaussians, Figures 42 and 43 show example plots of this from different angles. As can be seen the model fits the data well, significantly better than using 3 Gaussians as more of the data points are represented.

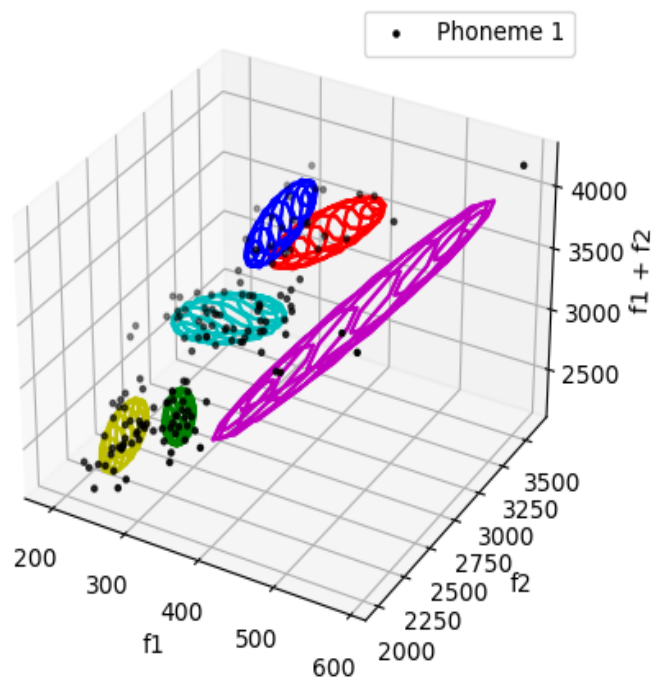


Figure 42 - Example plot showing trained the Gaussians and the data points, using method 1, phoneme 1 and 6 Gaussians

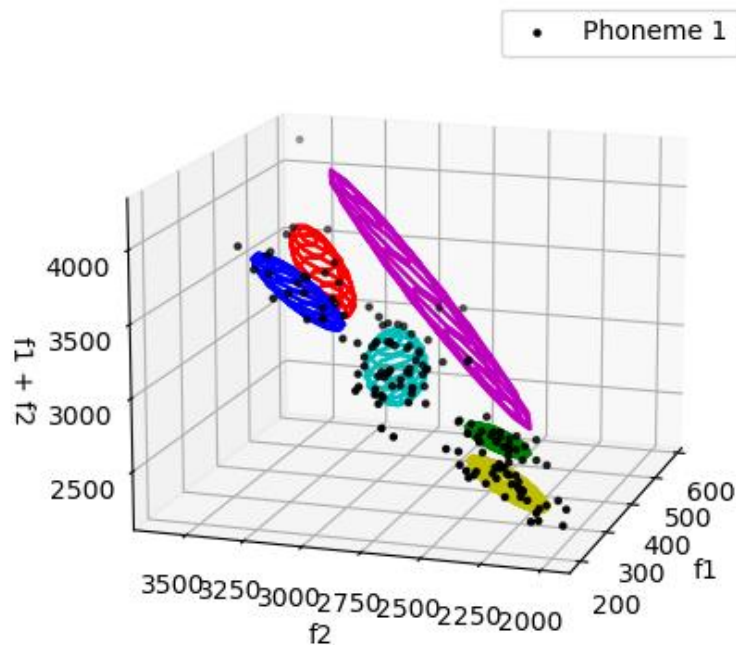


Figure 43 - Example plot showing trained the Gaussians and the data points from a different angle, using method 1, phoneme 1 and 6 Gaussians

The second method to solve this was to observe when a Gaussian was converging onto one point, thus becoming a singularity, by examining the determinant of its covariance matrix. If the determinant is 0 then the covariance matrix is not invertible, causing the program to crash. As such the check must be as it approaches 0 rather than when it reaches 0, in other words there must be a threshold value in use.

If a Gaussian is found to be converging then its mean is reset to a random value in the same way it was at the start of the code, by randomly selecting a point from the dataset. The covariance matrix is also reset to its original value the same way as previously, by working out the covariance matrix of the dataset using `np.cov` and dividing it by `k`.

Figure 44 shows the code needed to implement this solution. The reader should note the use of the `np.abs` function, this is used to account for the fact that a determinant of a matrix can be a negative number.

```
a = s[i]
determinant = np.abs(np.linalg.det(a))
if determinant <= 0.00000001 or np.isnan(determinant):
    random_indices = np.floor(N * np.random.rand((k)))
    random_indices = random_indices.astype(int)
    print("fired")
    cov_matrix = np.cov(X.transpose())
    s[i, :, :] = cov_matrix / k
    print(cov_matrix / k)
```

Figure 44 - Code to check if the determinant is below a threshold and reset the mean and covariance matrix

The code was then run with 3 Gaussians on phoneme 1, Figures 45 and 46 show examples plots of this from two different angles. As can be seen this model fits the data very well, with no overlap and a good representation of the data.

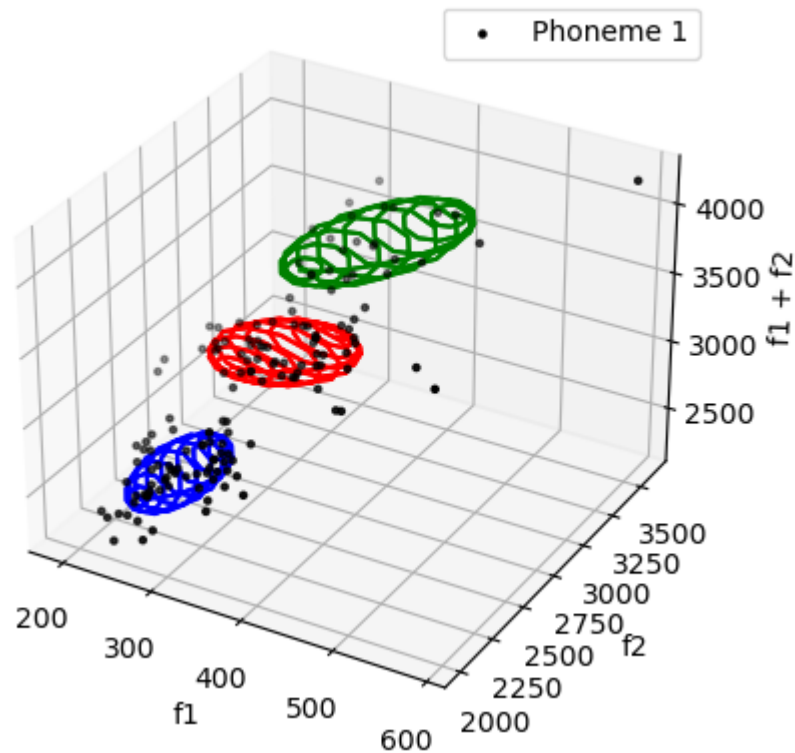


Figure 45 - Example plot showing trained the Gaussians and the data points, using method 2, phoneme 1 and 3 Gaussians

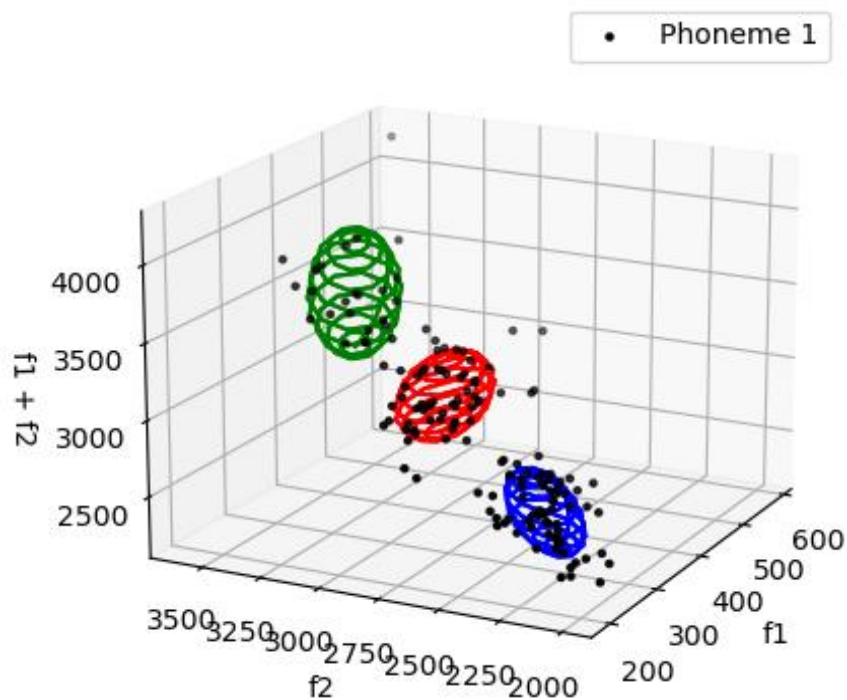


Figure 46 - Example plot showing trained the Gaussians and the data points from a different angle, using method 2, phoneme 1 and 3 Gaussians

The code was then also run on 6 Gaussians, Figures 47 and 48 show some examples plots of this from different angles. As can be seen the model fits the data well, with minimal overlap and a good representation of the data.

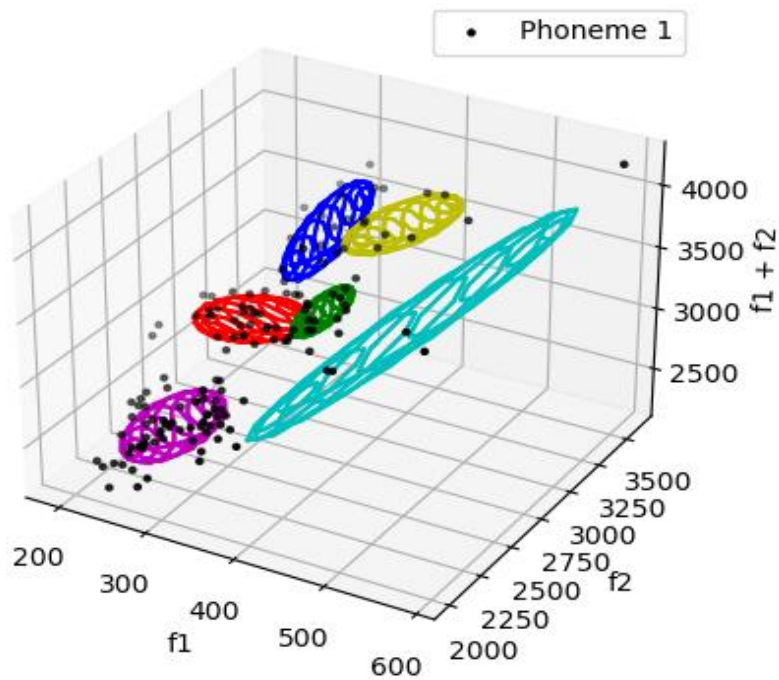


Figure 47 - Example plot showing trained the Gaussians and the data points, using method 2, phoneme 1 and 6 Gaussians

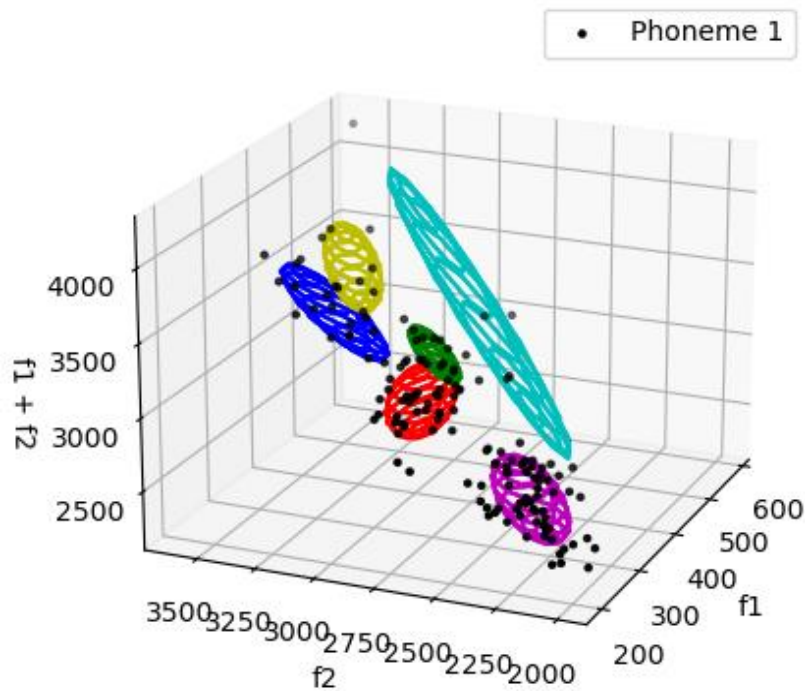


Figure 48 - Example plot showing trained the Gaussians and the data points from a different angle, using method 2, phoneme 1 and 6 Gaussians