ECS708: *Machine Learning*
Assignment 2: REPORT

- **Student name**: Pranav Narendra Gopalkrishna
- **Student number**: 231052045

Introduction

The aim of this lab is to get familiar with **Mixture of Gaussians** model.

1. This lab is the course-work activity **Assignment 2: Unsupervised Learning(20%)**
2. The Assignment is due on **Friday , 1st December, 11:59pm**
3. A report answering the **questions in red** should be submitted on QMplus along with the completed Notebook.
4. The report should be a separate file in **pdf format** (so **NOT** *doc, docx, notebook* etc.), well identified with your name, student number, assignment number (for instance, Assignment 1), module code.
5. Make sure that **any figures or code** you comment on, are **included in the report**.
6. No other means of submission other than the appropriate QM+ link is acceptable at any time (so NO email attachments, etc.)
7. **PLAGIARISM** is an irreversible non-negotiable failure in the course (if in doubt of what constitutes plagiarism, ask!).

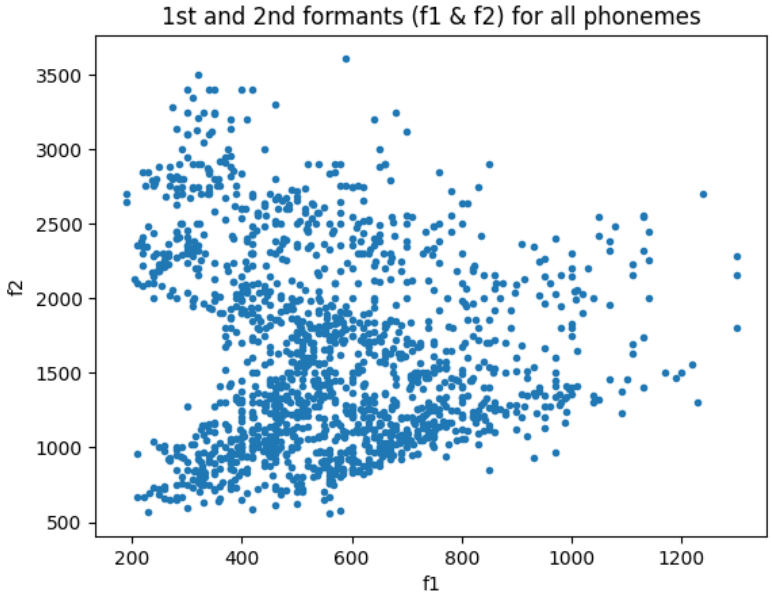
For this lab, we will use the Peterson and Barney's dataset of vowel formant frequencies. (For more info, look at Classification of Peterson & Barney's vowels using Weka - a copy of this article is on QMplus). More specifically, Peterson and Barney measured the fundamental frequency $F0$ and the first three formant frequencies ($F_1 - F_3$) of sustained English Vowels, using samples from various speakers.

The dataset contains 4 vectors ($F_0 - F_3$), containing the fundamental frequencies (F_0, F_1, F_2 and F_3) for each phoneme and another vector `phoneme_id` containing a number representing the ID of the phoneme. The formants here are the features, whereas the phonemes are the classes or labels; each observation of formant values is associated with a particular phoneme. In the exercises that follow, we will use only the dataset associated with formants F_1 and F_2. Also, we shall focus only on two classes, i.e. two phonemes, namely phoneme 1 (`phoneme_1`) and phoneme 2 (`phoneme_2`).

NOTE: Phonemes and frequencies are two different things; `phoneme_id` is the ground truth class of each row while frequencies represent features of each data point. Please review the papers provided with the lab materials for more details.

ABBREVIATION NOTE: "Mixture of Gaussians" will be abbreviated as MoG.

Q1. Produce a plot of $F1$ against $F2$. (You should be able to spot some clusters already in this scatter plot.). Comment on the figure and the visible clusters [2 marks]



OBSERVATIONS:

The above figure shows the scatterplot of observations, wherein one feature (dimension) is the 1st formant while the other feature is the 2nd formant. Together, the 1st and 2nd formants observed form a single point, and each such point belongs to either the *phoneme 1* class or *phoneme 2* class. The classes are not distinguished above, but we can make out some rough non-linear lines of division in the collection of points above, forming roughly two broad clusters, potentially corresponding the the respective classes the points belong to.

1. MoG using the EM algorithm

Recall the following definition of a Mixture of Gaussians. Assuming our observed random vector is $\mathbf{x} \in \mathbb{R}^D$, a MoG models $p(\mathbf{x})$ as a sum of K-many weighted Gaussians. More specifically:

$$
\begin{aligned}
&p(\mathbf{x}) \\
&= \sum_{k=1}^{K} p(c_k) Norm(\mathbf{x}|\mu_k, \Sigma_k) \\
&= \sum_{k=1}^{K} \frac{p(c_k)}{(2\pi)^{D/2} \det(\mathbf{\Sigma}_k)^{1/2}} \exp(-\frac{1}{2}(x-\mu)^T \sum_k^{-1}(x-\mu)) \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)^\top {\mathbf{\Sigma}_k}^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)\right)
\end{aligned}
$$

where:

- $\boldsymbol{\mu}_k \in \mathbb{R}^D$
- $\mathbf{\Sigma}_k \in \mathbb{R}^{D \times D}$
- $p(c_k) = \pi_k \in \mathbb{R}$

denote the k-th gaussian component's **mean vector**, **covariance matrix**, and **mixture coefficients** respectively. The K-many gaussian components' model parameters are referred to collectively as: $\theta = \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}_{k=1}^{K}$

EM Algorithm

For the E step we softly assign each datum to the closest centroid (using the current iteration's fixed model parameters) as in the K means example.

For the M step we update the model parameters θ to maximize the weighted log-likelihood. At a high level, for each centroid k we:

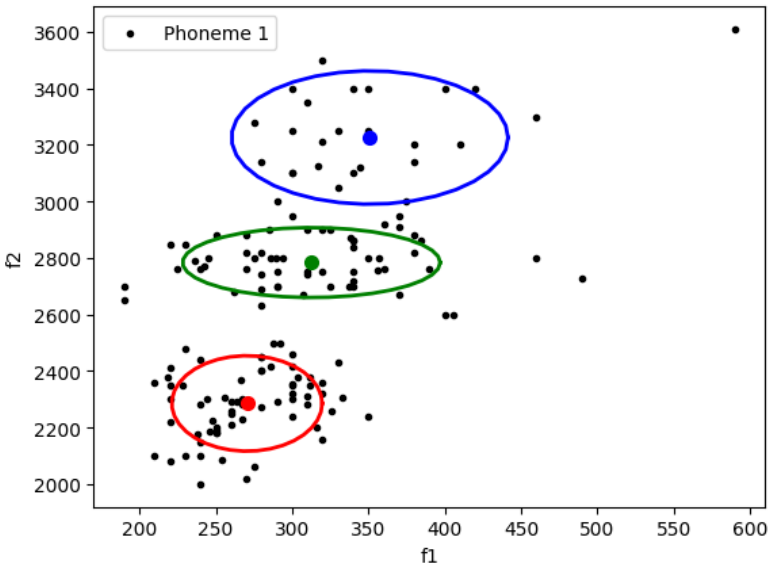
- Update the mean vectors
- Update the covariance matrices (we will fit Gaussians with diagonal covariance matrices)
- Set the mixture coefficients ($p(c_1), p(c_2)$, etc.) as the mean probability of a sample being generated by the k-th gaussian component

Train MoG

Q2. Run the code multiple times for $K = 3$, what do you observe? Use figures and the printed MoG parameters to support your arguments [2 mark]

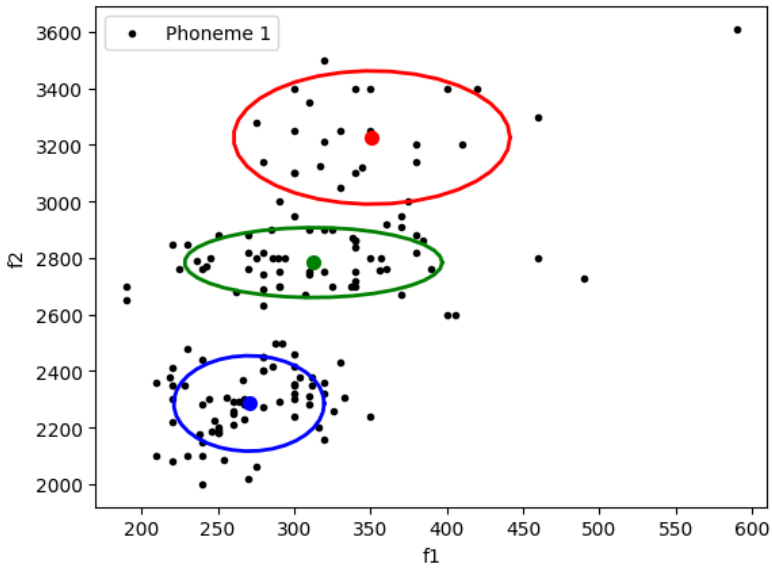
NOTE: *We have a choice of modelling the points for either phoneme 1 or phoneme 2. For this run, I have chosen phoneme 1 as the class of points to model.*

Run 1



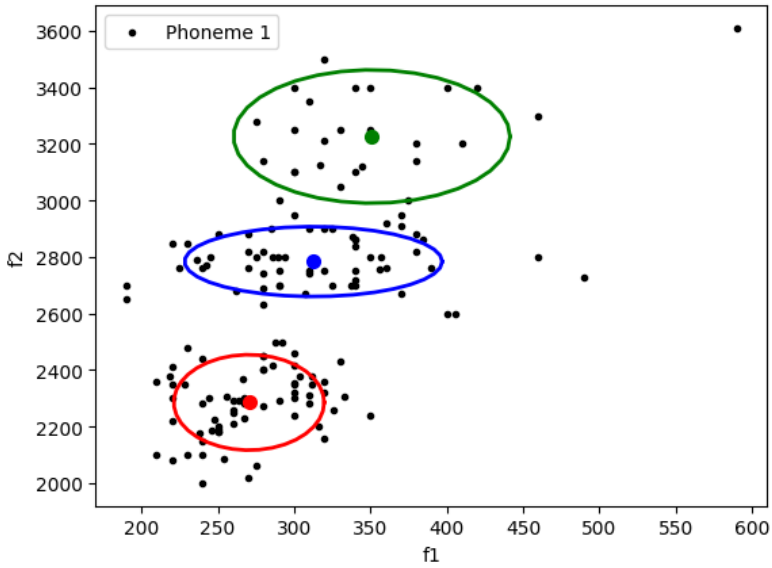
```
Finished.

Optimised a mixture of 3 mutlivariate Gaussian distributions.
------------------------
Array of 3 mean pairs:
[[ 312.59103 2783.897  ]
 [ 350.84445 3226.3342 ]
 [ 270.3952  2285.4653 ]]
------------------------
Array of 3 covariance matrices:
[[[ 3562.60079213     0.        ]
  [    0.          7657.72128897]]

 [[ 4102.84940298     0.        ]
  [    0.         27830.86272245]]

 [[ 1213.73839346     0.        ]
  [    0.         14278.4212236 ]]]
```

Run 2



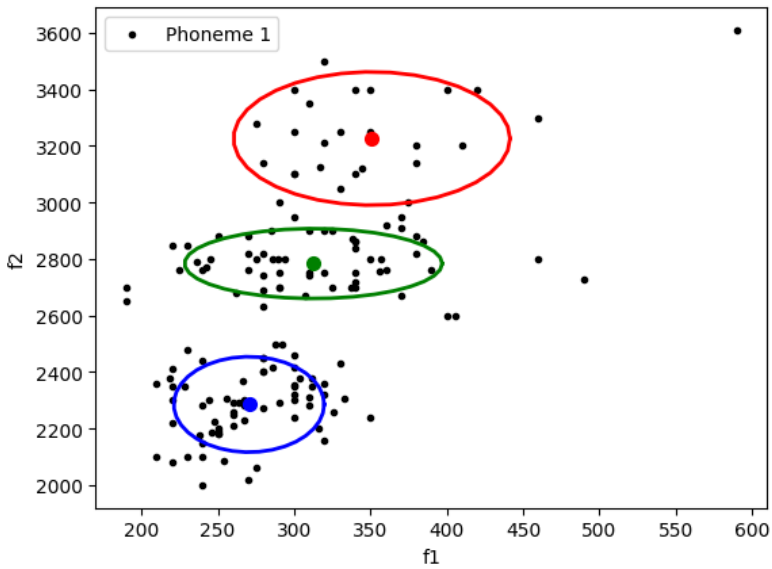

```
Finished.

Optimised a mixture of 3 mutlivariate Gaussian distributions.
------------------------
Array of 3 mean pairs:
[[ 270.3952  2285.4653 ]
 [ 350.84433 3226.331  ]
 [ 312.5909  2783.8965 ]]
------------------------
Array of 3 covariance matrices:
[[[ 1213.73836766     0.        ]
  [    0.         14278.42207326]]

 [[ 4102.83317151     0.        ]
  [    0.         27831.69600449]]

 [[ 3562.6029097      0.        ]
  [    0.          7657.64086123]]]
```

Run 3




```
Finished.

Optimised a mixture of 3 mutlivariate Gaussian distributions.
------------------------
Array of 3 mean pairs:
[[ 350.84454 3226.3376 ]
 [ 312.5912  2783.8977 ]
 [ 270.3952  2285.4653 ]]
------------------------
Array of 3 covariance matrices:
[[[ 4102.86619095     0.        ]
  [    0.         27829.99769203]]

 [[ 3562.59864229     0.        ]
  [    0.          7657.80433159]]

 [[ 1213.73842119     0.        ]
  [    0.         14278.4201857 ]]]
```

Run 4



```
Finished.

Optimised a mixture of 3 mutlivariate Gaussian distributions.
------------------------
Array of 3 mean pairs:
[[ 270.3952  2285.4653 ]
 [ 312.59125 2783.898  ]
 [ 350.8446  3226.3394 ]]
------------------------
Array of 3 covariance matrices:
[[[ 1213.73843494     0.        ]
  [    0.         14278.4202995 ]]

 [[ 3562.59743765     0.        ]
  [    0.          7657.84897245]]

 [[ 4102.875375       0.        ]
  [    0.         27829.54221422]]]
```

OBSERVATIONS

The cluster-boundary polygon roughly represents the area of points within a certain boundary (as scaled by the square root of the covariance matrix) from the centre (mean) of the given mutlivariate Gaussian distribution. Clustering in this case are not as much based on hard-boundary groups (hard clustering) and as they are based on probabilitistic spectrums (soft clustering) where each observation (a vector of multiple feature values) belongs to the k available multivariate Gaussian distribution with some probability density assigned to it; the higher the density a distribution gives it, the more strongly it is part of the cluster represented by the distribution.

In the above graphs, we see a better-than-random clustering of the points with respect to the cluster-boundary polygons. This indicates that the separation of the points with respect to their clusters within a single class (*phoneme 1* in this case) has been obtained, indicating that the mixture of Gaussians adequately models the probability densities of the observations (probability density is with respect to a point belonging to the given class). The cluster-boundary polygons also help identify the outliers, i.e. the points that do belong to the given class, but would generally by regarded as outside the given class if observed apart from training.

From the covariance matrices obtained for the above mixture of Gaussians, we observe that there is zero covariance between the features F_1 and F_2 (i.e. the 1st and 2nd formants). This is based on the assumption of zero covariance, and is achieved by the lines:

```
coef_2 = Z[:,i]/np.sum(Z[:,i])
coef_2 = np.expand_dims(coef_2, axis=1)
res_1 = np.squeeze(np.matmul(coef_1, coef_2))
s[i,:,:] = np.diag(res_1)
```

Hence, the training process assumes that covariance is zero, leaving only the diagonal of the covariance matrix (which contains the variance values) non-zero.

Q5. Use the 2 MoGs ($K = 3$) learnt in tasks 2 & 3 to build a classifier to discriminate between phonemes 1 and 2, and explain the process in the report [4 marks]

Here, we classify using the Maximum Likelihood (ML) criterion and calculate the misclassification error. Note that classification under the ML compares $p(\mathbf{x}; \boldsymbol{\theta}_1)$, where $\boldsymbol{\theta}_1$ are the parameters of the MoG learnt for the first phoneme, with $p(\mathbf{x}; \boldsymbol{\theta}_2)$, where $\boldsymbol{\theta}_2$ are the parameters of the MoG learnt for the second phoneme.

The following code has been included because a comment references it...

```
inds_1 = np.where(phoneme_id==1)
inds_2 = np.where(phoneme_id==2)
inds_1 = inds_1[0]
inds_2 = inds_2[0]
inds_1_and_2 = np.concatenate((inds_1, inds_2), axis=0)
# NOTE: inds_1_and_2 is a 1D array containing all elements of inds_1 & inds_2

X = X_full[inds_1_and_2, :]
Y = phoneme_id[inds_1_and_2]
# Check "IMPLEMENTATION DETAILS" in the text block below this code block
```

IMPLEMENTATION NOTE 1: **The target value array** `phoneme_id`

`phoneme_id` is a 1D array of 1520 values, wherein the same following sequence of 20 numbers is repeated 76 times: $1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10$

`phoneme_id` is the vector of target variable values corresponding to each observation. In this particular case, the feature values (valules of formants 1 and 2) were observed with respect to each phoneme, which is why the target variable values are so ordered.

IMPLEMENTATION NOTE 2: `np.where(phoneme_id==i)`

The function call `np.where(phoneme_id==i)` returns the array of indices of the Boolean-valued array `phoneme_id==i` where the value is `True`; equivalently, it returns the 1D array of indices of `phoneme_id` where the value is `i`. The return value of `np.where` is a tuple of 1D arrays; in this case, this tuple contains only one array (as there is only one array to pick indices from), yet the return value is still a tuple. Hence, to pick out the required array of indices, we index the return value using the index 0, i.e. we pick our the required arrays by `inds_1[0]` and `inds_2[0]`.

IMPLEMENTATION NOTE 3: **Indexing the observation & observed target arrays** `X_full` **&** `phoneme_id` **using** `inds_1_and_2`:

Remember that `X_full` is the matrix of feature variable values, wherein each row is an observation, which is an array of two values: the values of formants 1 and 2 for the given observation. Also remember that the vector of target values for these observations is `phoneme_id`; this means the number of rows in `X_full` is the same as the number of elements in `phoneme_id`.

We select the rows of `X_full` based on the chosen indices of `phoneme_id` obtained before and stored in `inds_1_and_2`. In effect, seeing that the indices in `inds_1_and_2` are all the indices of `phoneme_id` where the phoneme ID is either 1 or 2, `X_full[inds_1_and_2,:]` selects all the observations for which the phoneme ID is 1 or 2. We do the same index selection for `phoneme_id` so that we have target values to which we can compare each observation. This will be relevant when testing the accuracy of our classifier.

EXPLAINING THE CLASSIFICATION PROCESS:

For each class of points (i.e. observations corresponding to `phoneme_1` and `phoneme_2` respectively), we aim to estimate the density distribution for K different Gaussian distributions that together (as a sum) describe the data's classification. A mixture of Gaussians trained for class C is supposed to be such that the sum of the probability densities of the K different Gaussians for a given observation (i.e. a certain tuple of feature values) is the combined probability density of that the observation must be classified into class C.

We have mixture models that model each class of points (i.e. the set of observations corresponding to a certain phoneme), and to perform the classification for a given observation, we must do the following:

- For each model trained on a certain class of points:
  - Obtain the probability densities for each Gaussian in the mixture for each given observation
  - Obtain the weighted sum of the above-obtained probability densities for each observation
- For each observation:
  - Compare the combined densities from each model
  - Classify the observation according to the class (i.e. phoneme) corresponding to the model showing the highest combined density

NOTE: *The weights are assigned to each Gaussian within a mixture of Gaussians such that the total weighted sum amounts to* 1.

Q6. Repeat for $K = 6$ and compare the results in terms of accuracy. [2 mark]

Running the created classifier for $K = 3$ (given in the source code file), we obtain...

```
Classifier accuracy: 95.066%
```

Running the created classifier for $K = 6$ (given in the source code file), we obtain...

```
Classifier accuracy: 96.053%
```

COMPARISON:

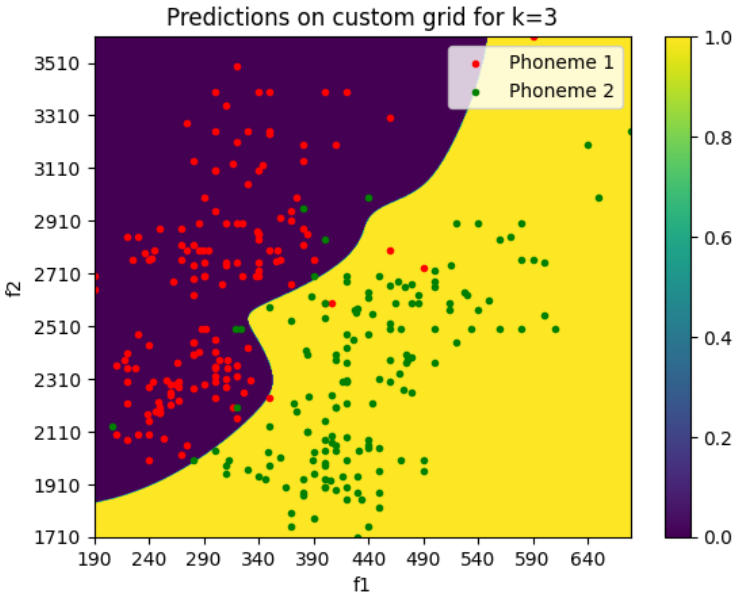
When using a mixture of 6 Gaussians instead of 3, i.e. when $K = 6$, the accuracy of classification is very similar to when $K = 3$ (with some variation). Over multiple trials, neither K value showed consistently more accuracy compared to the other.

Q7. Display a "classification matrix" assigning labels to a grid of all combinations of the $F1$ and $F2$ features for the $K = 3$ classifiers from above. Next, repeat this step for $K = 6$ and compare the

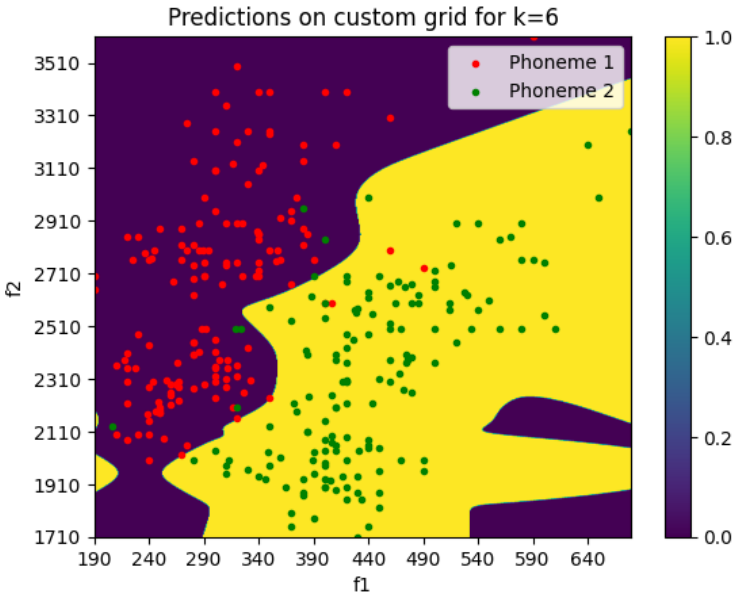
two. [3 marks]

In particular, we must create a custom uniform grid of points that spans all possible intermediate combinations of $F1$ and $F2$ features, using the features' minimum and maximum values as the limits. (Hint: check documentation for np.meshgrid and np.linspace). Classify each point in the grid using one of your classifiers. That is, create a classification matrix, $\mathbf{M}$, whose elements are either 0 or 1. $M(i, j)$ is 0 if the point at location (i, j) on the custom grid is classified as belonging to phoneme 1, and is 1 otherwise.

Displaying classification matrix for $K = 3$...



Displaying classification matrix for $K = 6$...



COMPARISON OF THE ABOVE:

NOTE: *In the above colour scale, 0 means phoneme 1 and 1 means phoneme 2.*

While both the above mixtures of Gaussians classify the data points relatively accurately, one notable difference is that when using a mixture of 6 Gaussians instead of 3, i.e. when $K = 6$, the classification boundary becomes more complex and more closely fitted to the given data. This means with $K = 6$, fewer points are regarded as outliers; this indicates a greater danger to overfitting the mixture of Gaussians model to the given training data. Furthermore, the model also classifies certain non-existent outliers (ex. near the bottom right of the graph) as belonging to a certain class (ex. `phoneme_1`), indicating a greater danger of misclassification due to the overconfident classification of outliers.

Q8. Try to fit a MoG model to the new data. What is the problem that you observe? Explain why it occurs [2 marks]

The new data refers to a new dataset that will contain 3 columns, as follows:

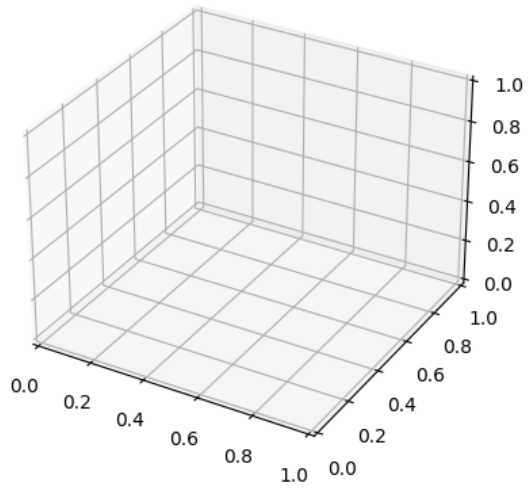
$$X = [F_1, F_2, F_1 + F_2]$$

NOTE: *We have a choice of modelling the points for either phoneme 1 or phoneme 2. For this run, I have chosen phoneme 1 as the class of points to model.*

When running the following code, we get the following error...

```
ax = plt.axes(projection='3d')
title_string = 'Phoneme {}'.format(p_id)

N, D = X.shape
p = np.ones((k))/k
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
n_iter = 150

# Initialize covariances:
for i in range(k):
    cov_matrix = np.cov(X.transpose())
    s[i,:,:] = cov_matrix/k

Z = np.zeros((N,k)) # shape Nxk

#================================================

# Run Expectation Maximization algorithm for n_iter iterations
zeros = {"t":[], "i":[]}
for t in range(n_iter):
    display.clear_output(wait=True)
    print('Iteration {:03}/{:03}'.format(t+1, n_iter))

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

      #------------------------------------

      # We will fit Gaussians with 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])
      p[i] = np.mean(Z[:,i])

plot_gaussians(ax, 2*s, mu, X, 'Phoneme {}'.format(p_id))
plt.show()
```

```
Iteration 016/150



---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-23-0561ba74a805> in <cell line: 43>()
     46
     47     # Do the E-step
---> 48     Z = get_predictions(mu, s, p, X)
     49     Z = normalize(Z, axis=1, norm='l1')
     50     # Do the M-step:


<ipython-input-6-592dfa548f29> in get_predictions(mu, s, p, X)
     21                 mu_i_repeated = np.repeat(mu_i, N, axis=1)
     22                 X_minus_mu = X - mu_i_repeated.transpose()
---> 23                 inverse_s = scipy.linalg.pinv(s[i])
     24                 inverse_s = np.squeeze(inverse_s)
     25                 s_i_det = scipy.linalg.det(s[i])


/usr/local/lib/python3.10/dist-packages/scipy/linalg/_basic.py in pinv(a, atol, rtol, return_rank, check_finite, con
   1431
   1432     """
-> 1433     a = _asarray_validated(a, check_finite=check_finite)
   1434     u, s, vh = _decomp_svd.svd(a, full_matrices=False, check_finite=False)
   1435     t = u.dtype.char.lower()


/usr/local/lib/python3.10/dist-packages/scipy/_lib/_util.py in _asarray_validated(a, check_finite, sparse_ok, object
    238             raise ValueError('masked arrays are not supported')
    239     toarray = np.asarray_chkfinite if check_finite else np.asarray
--> 240     a = toarray(a)
    241     if not objects_ok:
    242         if a.dtype is np.dtype('O'):


/usr/local/lib/python3.10/dist-packages/numpy/lib/function_base.py in asarray_chkfinite(a, dtype, order)
    625     a = asarray(a, dtype=dtype, order=order)
    626     if a.dtype.char in typecodes['AllFloat'] and not np.isfinite(a).all():
--> 627         raise ValueError(
    628             "array must not contain infs or NaNs")
    629     return a


ValueError: array must not contain infs or NaNs
```



OBSERVATIONS ON THE ABOVE ERROR:

Tracing the error...

We observe that the error arises after a few iterations when calling the `get_predictions` function. Within the `get_predictions` function, the error arises in the line:

```
mu[i,:] = np.matmul(X.transpose(), Z[:,i])/np.sum(Z[:,i])
```

Within the above line, the error is a `ValueError` ; a NumPy array cannot contain infinity or NaN (not-a-number) values, i.e. `float("inf")` or `float("nan")` , because these values are undefined and not arithmetically interpretable. This happens when `np.sum(Z[:,i])` $= 0$, `Z[:,i]` denotes the i-th column of `Z` , so to understand how this error arose, we must understand how the matrix `Z` came to contain zero columns. For reference, here is some of the code of the function `get_predictions` :

```
def get_predictions(mu, s, p, X):
    """
    param mu    : means of GMM components
    param s             : covariances of GMM components
    param p     : weights of GMM components
    param X             : 2D array of our dataset
    """

    # Get number of GMM components:
    # GMM ==> Gaussian Mixture Model
    k = s.shape[0]
    # Get number of data samples:
    N = X.shape[0]
    # Get dimensionality of our dataset:
    D = X.shape[1]

    Z = np.zeros((N, k))
    for i in range(k):
        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 - mu_i_repeated.transpose()
        inverse_s = scipy.linalg.pinv(s[i])
        inverse_s = np.squeeze(inverse_s)
        s_i_det = scipy.linalg.det(s[i])
        x_s_x = np.matmul(X_minus_mu, inverse_s)*X_minus_mu
        try:
          a = ((2*np.pi)**D) * np.abs(s_i_det)
          if a == 0: continue
          Z[:, i] = p[i]*(1/np.power(a, 0.5)) * np.exp(-0.5*np.sum(x_s_x, axis=1))
          except: pass
    return Z
```

We see that after its initialisation, `Z` is only being assigned values in the line:

```
Z[:, i] = p[i]*(1/np.power(a, 0.5)) * np.exp(-0.5*np.sum(x_s_x, axis=1))
```

The issue of division by zero can arise if `a` is zero. `a` is assigned as:

```
a = ((2*np.pi)**D) * np.abs(s_i_det)
```

The code has been altered so that if `a` happens to be zero, we skip the following lines and move to the next iteration (this was done to enable the training of the models to continue despite a few instances where `a` $= 0$). Skipping the following lines skips the assignment of a particular column of `Z`. But since every value of `Z` was initialised as zero, skipping the assignment of a particular column of `Z` leaves that column as a zero column. Hence, to understand why `Z` contains zero columns, we must understand when `a` equals zero.

`D` is a non-zero constant (denoting the number of columns in `X`), as is `2*np.pi`. Hence, `a` is zero if and only if `np.abs(s_i_det)` is zero. Now, `s_i_det` is assigned in the previous lines as:

```
s_i_det = scipy.linalg.det(s[i])
```

In other terms, it is the determinant of `s[i]`, i.e. the covariance matrix of the i-th Gaussian distribution in the given mixture of Gaussians.

Understanding the error...

The determinant of a matrix is zero if and only if either of the following holds:

- At least one row is a linear combination of one or more of the other rows
- At least one column is a linear combination of one or more of the other columns

`s_i_det` is zero if and only if at least one row or column of `s[i]` (the covariance matrix for the i-th Gaussian) is a linear combination of one or more other rows or columns. This happens in `s[i]` if and only if one or more of the columns (i.e. features) in the dataset are perfectly linearly correlated to each other. In other words, when two or more features are perfectly linearly correlated, we face the issue of `s[i]` becoming a singular matrix (i.e. we face the singularity problem). In our case, we see this linear correlation to clearly be the case, as we include the column $F_1 + F_2$, which is clearly the linear combination of the features F_1 and F_2.

Q9. Suggest ways of overcoming the singularity problem and implement one of them. Show any training outputs in the report and discuss. [3 marks]

NOTE: *We have a choice of modelling the points for either phoneme 1 or phoneme 2. For this run, I have chosen phoneme 1 as the class of points to model.*

In regression analysis and statistical models in general, singularity is the extreme form of multicollinearity - when a perfect linear relationship exists between variables or, in other terms, when the correlation coefficient is equal to 1.0 or -1.0. Such absolute multicollinearity could arise when independent variable are linearly related in their definition. A simple example: two variables "height in centimeters" and "height in inches" are included in the regression model. In our case, we see this linear correlation with the inclusion of $F_1 + F_2$.

> REFERENCE: https://www.statistics.com/glossary/singularity/

POSSIBLE SOLUTIONS FOR SINGULARITY PROBLEM:

- Instead of including $F_1 + F_2$, include a non-linear combination of F_1 and F_2 (ex. $F_1\dot{F}_2$)
- Apply a non-linear transformation to the linearly dependent features before fitting the data
- Add zero mean noise to the original data to ensure that the values are not perfectly correlated
- Add a small constant to each diagonal element of the covariance matrix after intialisation and update
- Add zero mean noise to the diagonal of each covariance matrix after initialisation and update
- Assume zero covariance, i.e. retain only the diagonal of each covariance matrix, keeping all else zero

Transforming the linearly dependent feature $F_1 + F_2$, if done while retaining the nature of change in the feature's values (i.e. the higher the value of $F_1 + F_2$, the higher the value of its transformation and vice

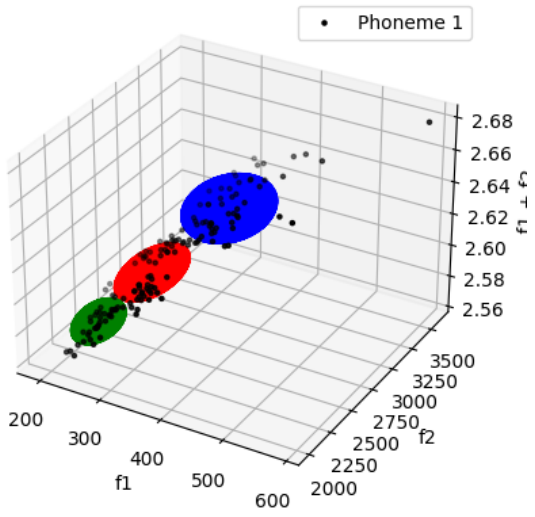
versa), can help us train our model while both avoiding singularity and considering the changes in the values of the linearly dependent feature. Suitable examples of transformation functions are:

- Logarithm
- Cube root

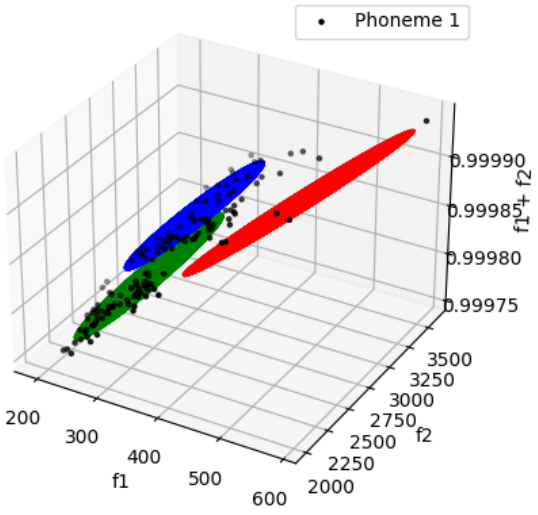
Assuming zero covariance, i.e. retaining only the diagonal of each covariance matrix is a common option in fitting mixtures of Gaussians, and is a valid option since for each point, the values of linearly dependent features are closer together and thus have a similar effect in the updation of parameters, which is similar to not including linearly dependent features.

Adding noise or a small constant or diagonal to the covariance matrix essentially tries to ensure that the rows or columns are not perfect linear combinations of each other, hence ensuring that the determinant of the coovariance matrix is non-zero (allowing for non-zero predictions, as explained in the previous question's response). This is practically the same thing as adding some noise to the data so as to make it not perfectly correlated (even if two fields are still highly correlated).

METHOD 1: Assume zero covariance & retain only variance in a diagonal matrix...



METHOD 2: Transform the training data (for training only) by taking the logarithm of the last column $F_1 + F_2$...



DISCUSSION:

We observe that by converting the convariance matrix to a diagonal matrix (i.e. setting all covariance values to zero), we are able to train the model without encountering singularities, which is expected since the determinant of diagonal matrix with no zeros in the diagonal can never be zero. The mixture of Gaussians is able to cluster (with soft clustering) most of the points of the given data (thereby will be able to appropriately classify new observations), indicating that assuming zero covariance between the features was a suitable assumption to make in this context.

SIDE NOTE: _The diagonal elements are the variances of the features, and we expect there to be no zeros in the diagonal since the feature values are expected to have at least some variance.

We also observe that by transforming the third feature $F_1 + F_2$ using cube root, we are able to remove the linear dependence between this feature and the features F_1 and F_2, resulting in a covariance matrix that does not reflect perfect correlation, and thus, whose determinant is never zero. The mixture of Gaussians is able to cluster (and thereby will be able to appropriately classify) most of the points of the given data. In most iterations of the training loop, we can observe that using this transformation of the last column can lead to the multivariate Gaussians in the mixture to containing higher covariance values between certain features (which is expected, since F_1 and F_2 are closely related to $F_1 + F_2$ and hence to $(F_1 + F_2)^{\frac{1}{3}}$), leading to more tightly-fitting clusters that tend to include more distant points. This indicates that in general, transforming the training data may lead to more accurate models that better reflect the relationship between features.