Assignment 4

Kathan Vyas

EECE 5644

Question 1

**Step 1: Dataset Creation:**

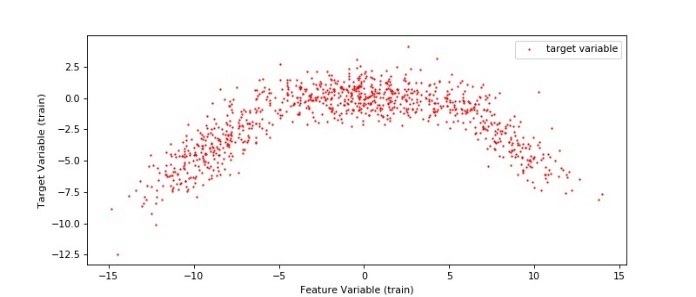
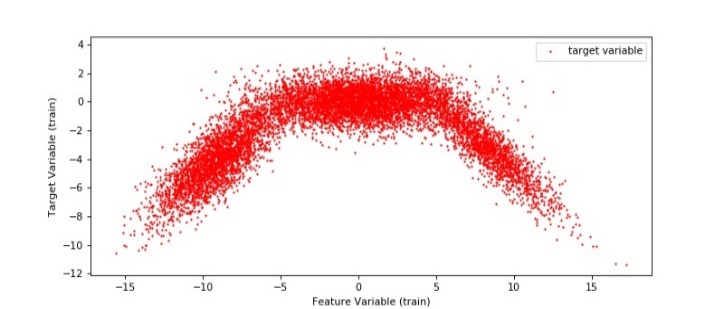
Dataset was created in MATLAB using the function Multi-ring dataset. I created two different datasets with 1000 and10000 datapoints. The visualization of the dataset based on classes is provided below:

Figure 2: Dataset with 10K data points

Figure 1: Dataset with 1000 points

I used MATLAB to create this dataset and then used python to convert the ‘. mat’ datafiles into numpy arrays. I created a function called mat\_to\_array in python that would take in a mat file, number of datapoints and a flag which tells whether the file contains datapoints or labels. The code for this is provided into the section helpful functions.

Dataset structure:  
X = [x1, x2]

Feature Variable: x1  
Target Variable: x2

**Step 2: Constructing MLP for our dataset**

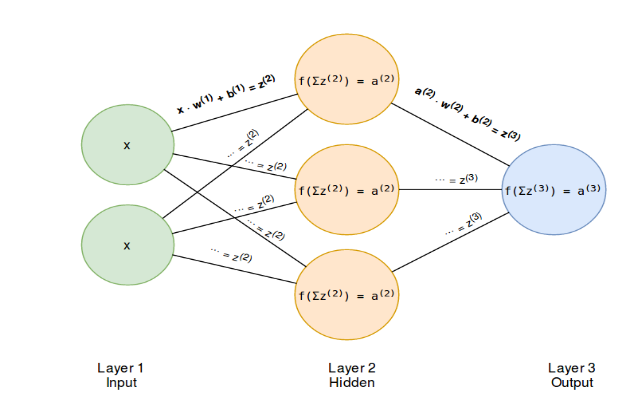
We are going to build a three-layer neural network. Figure 3 shows an example of a three-layered neural network. This is the minimum required number of layers when talking of a multi-layer perceptron network. Every neural net requires an input layer and an output layer. The remaining layers are the so-called hidden layers. The input layers will have data as input and the output layers will make predictions. Predictions here are x2 datapoints from our dataset vectors and the input are x1 datapoints.

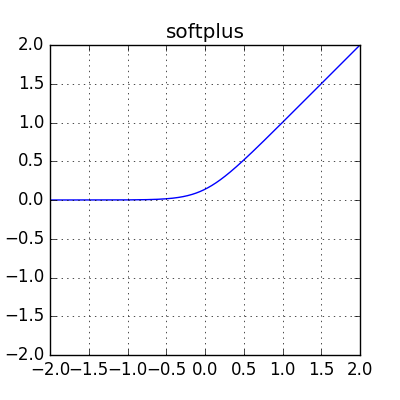
Figure 3: A 3-layer Perceptron, also called feed forward network

The number of nodes in the input layer is determined by the dimensionality of our data, in our case 1000 (for training) or 10,000 (for testing). Similarly, the number of nodes in the output layer is determined by the dimensionality of our output predictions. The input to the network will be x1 data-point and its output will x2 datapoint. The middle layer can have “N” number of neurons. I used cross validation data to select between 1 neuron to 6 neurons. The statistics with that is shown in table 1 (which shows only a few examples. More detailed table could be found in github tables.xlsx).

For implementing MLP regression, I have used MLPRegressor class from Scikit-learn library. Class [MLPRegressor](https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html#sklearn.neural_network.MLPRegressor) implements a multi-layer perceptron (MLP) that trains using backpropagation with no activation function in the output layer, which can also be seen as using the identity function as activation function. Therefore, it uses the square error as the loss function, and the output is a set of continuous values. [MLPRegressor](https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html#sklearn.neural_network.MLPRegressor) also supports multi-output regression, in which a sample can have more than one target.

**Step 3: Activation: Softplus + Identity**

The hidden layer activation function is a softplus activation which is in form of **f(x) = ln (1+exp x).** The derivative of softplus is **f ′(x)=exp(x) / (1+exp(x)) = 1/ (1 +exp (−x))** which is also called the logistic function. The prime version is used in back propagation where I am using Stochastic Gradient Descent.

So, our MLP mathematically looks like this:

Input: x1

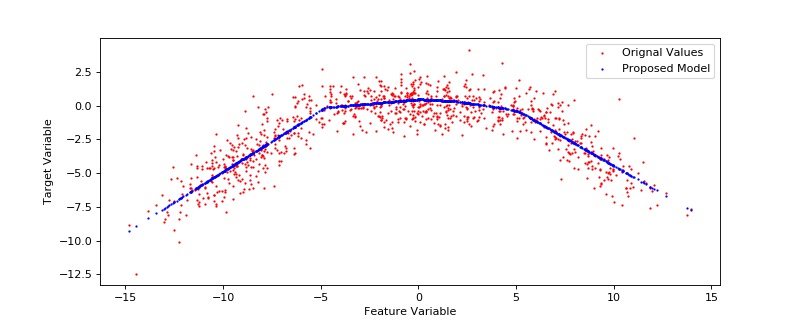
Hidden layer: **ln (1+exp x1)**

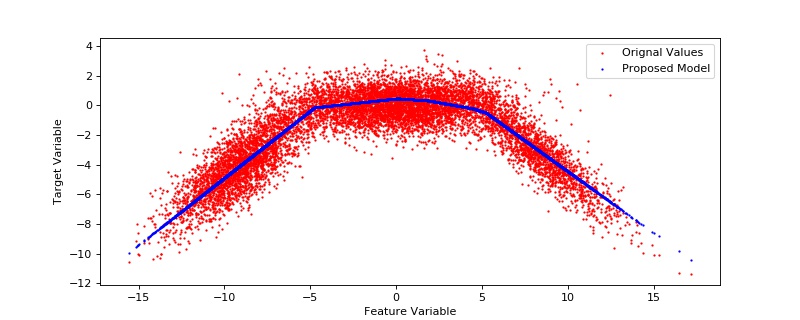
**Output: x2 = w \* ln (1+exp x1) + b**

Tables of experiments:

Table : Cross Validation to select Neurons for hidden layer

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Neurons in hidden layer | Kth fold (with number of legs) | Kth value | MSE LOSS |
| 1 | Train: 1-2,4-10 Validate:3 | 4 | 5.3111 |
| 2 | Train: 1-4,6-10 Validate:5 | 6 | 4.5701 |
| 3 | Train: 2-10 Validate:1 | 2 | 1.2927 |
| **4** | **Train: 1-2,4-10 Validate:3** | **4** | **1.2884** |
| 5 | Train: 1-7,9,10 Validate:8 | 9 | 1.3445 |
| 6 | Train: 2-10 Validate:1 | 10 | 1.3067 |

Results:



Question 2

**Step1: Creating Dataset**

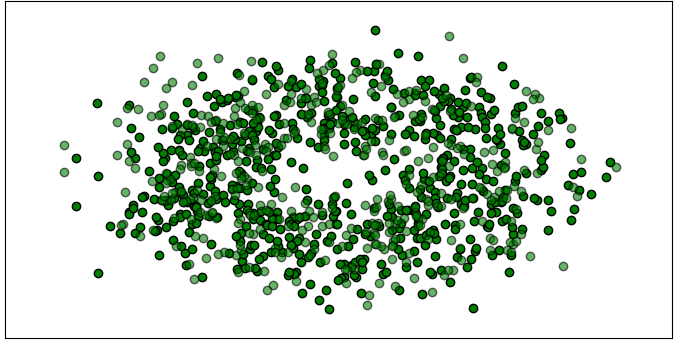
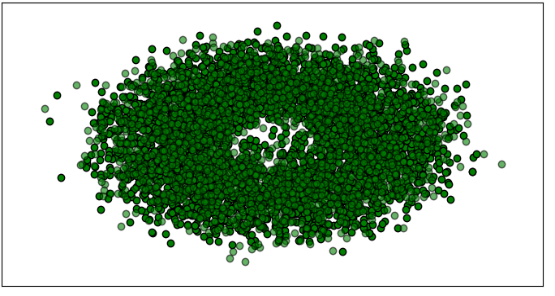
Dataset was created in MATLAB using the function Multi-ring dataset. I created two different datasets with 100 and10000 datapoints. I chose 2 class categories. The visualization of the dataset based on classes is provided below:

Figure 1 : Dataset with 10000 datapoints across 3 classes

Figure 2: Dataset with 1000 datapoints across 3 classes

I used Matlab to create this dataset a nd then used python to convert the ‘.mat’ datafiles into numpy arrays. I created a function called mat\_to\_array in python that would take in a mat file, number of datapoints and a flag which tells whether the file contains datapoints or labels. The code for this is provided into the section helpful functions.

**Step2: Using SKLEARN SVM and Cross Validation of Dataset to set value for C and Sigma**

In order to implement the SVM with RBF kernel, I have used SKLEARN.SVM library’s SVC. The implementation is based on libsvm. The fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples. For large datasets consider using [sklearn.svm.LinearSVC](https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) or [sklearn.linear\_model.SGDClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html#sklearn.linear_model.SGDClassifier) instead, possibly after a [sklearn.kernel\_approximation.Nystroem](https://scikit-learn.org/stable/modules/generated/sklearn.kernel_approximation.Nystroem.html#sklearn.kernel_approximation.Nystroem) transformer. The multiclass support is handled according to a one-vs-one scheme.

The input to the algorithm are:

1)C: Regularization Parameter - The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

2)gamma: Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’.

3)kernel: Specifies the kernel type to be used in the algorithm. It must be one of ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable. If none is given, ‘rbf’ will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape (n\_samples, n\_samples).

The selection of kernel is set to rbf. The selection of C and Gamma is done using cross validation on traning set of 1000 datapoints. The resulting statistics could be found in table 2 (more details in github table.xlsx).

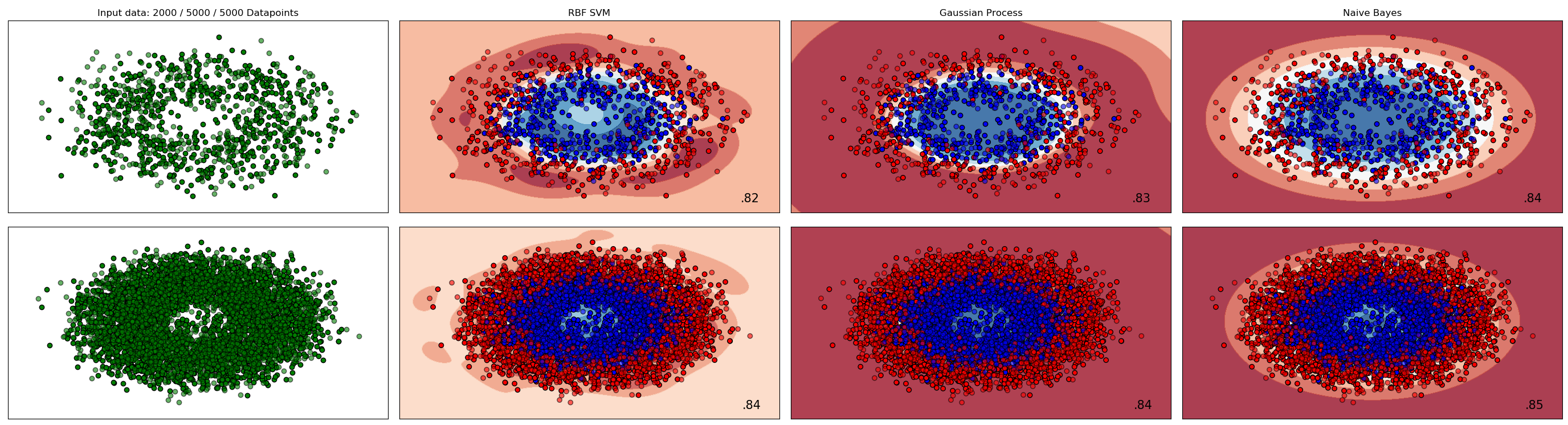
Tables of experiments:

Table : Cross Valid to select C and Gamma

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Neurons in hidden layer | Kth fold (with number of legs) | Kth value | C and Gamma selection |
| 1 | Train: 1-2,4-10 Validate:3 | 4 | 2,3 |
| 2 | Train: 1-4,6-10 Validate:5 | 6 | 1,3 |
| 3 | Train: 2-10 Validate:1 | 2 | 1,3 |
| **4** | **Train: 1-2,4-10 Validate:3** | **4** | **1,2** |
| 5 | Train: 1-7,9,10 Validate:8 | 9 | 2,3 |
| 6 | Train: 2-10 Validate:1 | 10 | 1,2 |

Results:

I have also implemented Gaussian Naïve Bayes and Gaussian Process classification along with SVM.



Question 3

**Step1: Pre-Processing**

For preprocessing, I use OPENCV library in python. First I store the RGB values for each image in a numpy array. Next I appended the row and column indexes in the numpy array to make it a 5 column structure.

Image (numpy array):

Col 1- Row index Col2- Column Index

Col3- Red pixel value Col3- Green pixel value Col3- Blue pixel value

Next in I normalize each columns dividing them with max values and bringing the values close down between 0-1.

This is repeated for both the images.

**Step2: Using Max Likelihood Par with GMM (for 2 guassians) and then implementing MAP to classify**

In here we take the 5 dimensional array and fit a GMM with 2 component with the help of max likelihood parameter estimation. This is implemented similar to Assignment 2 and 3. The results for fitting a gaussian with 2 components and splitting image in two parts is shown here:



Here we have only one gaussian to fit using MLPE with 2 components. The value for mean and sigma are experimentally found our to be :

Plane Image: 6.01 and 1.00 Bird Image: 5.1 and 9.99

**Step3: Using SKLEARN GMM and Cross Validation of Dataset to set value for number of GMMs**

I use the SKLEARN GMM library to set up the number of clusters. I validate it using the SKLEARN Kmeans library and EM algorithm. In order to select the number of GMMs I use 10 fold cross validation. Based on experiments, the number of GMMs to fit are 2. At the end of the GMM experiment, I implement K=2 k=3 and k=4 for both the images to see the exact results.

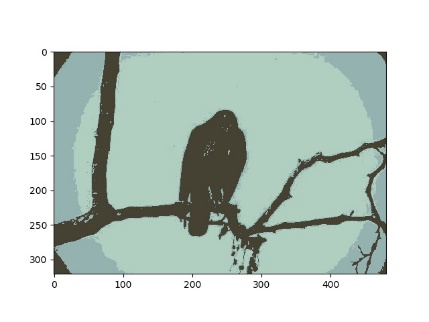
Now for each image, based on the number of gaussians selected, I assign the classes to divide the arrays into corresponding classes.

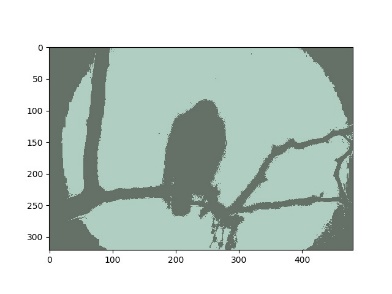
So We will finally have following arrangements:

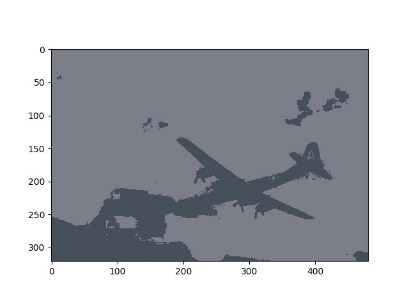
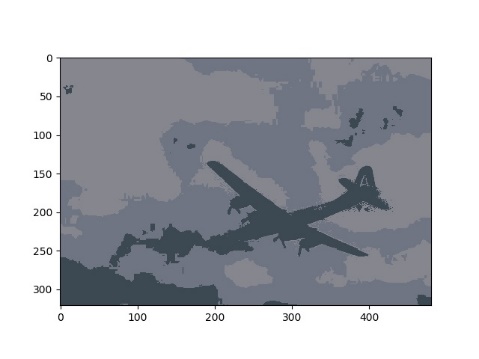
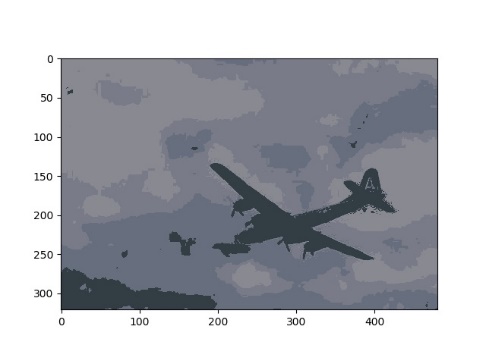
K=2 , classes=2

K=3, classes =3

K=4, Classes =4

Bird image results:

K =2, 3 , 4

plane image results:

K =2, 3, 4

The result of cross validation is provided in the table. More detailed table is provided in github repo.

Tables of experiments:

Table : Cross Validation to select nO. Of clusters in GMM

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Neurons in hidden layer | Kth fold (with number of legs) | Kth value | C and Gamma selection |
| 1 | Train: 1-2,4-10 Validate:3 | 4 | 2,3 |
| 2 | Train: 1-4,6-10 Validate:5 | 6 | 1,3 |
| 3 | Train: 2-10 Validate:1 | 2 | 1,3 |
| **4** | **Train: 1-2,4-10 Validate:3** | **4** | **1,2** |
| 5 | Train: 1-7,9,10 Validate:8 | 9 | 2,3 |
| 6 | Train: 2-10 Validate:1 | 10 | 1,2 |

**Conclusion**

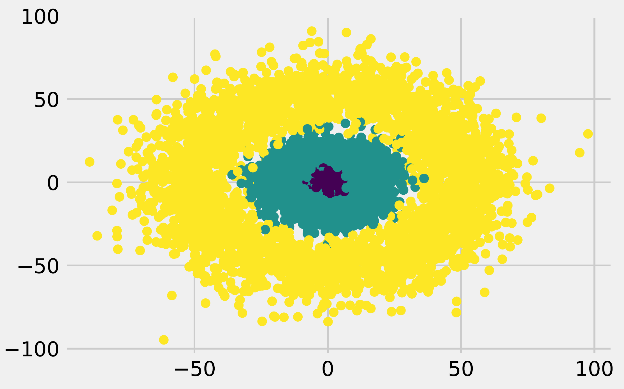
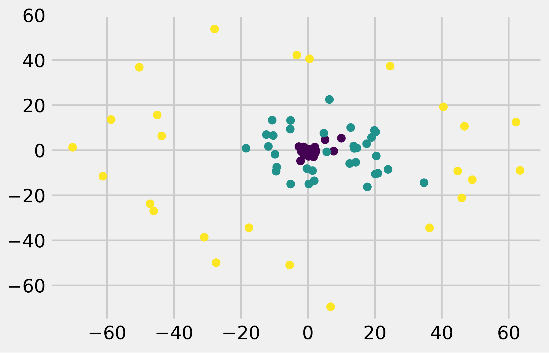
We look at the log likeligood errors for both the MLP (multilayer perceptron) and GMM (Gaussian Mixture Model) to check which is the best model. Details about the same is provided in table below. Note that I have considered loss and accuracy of classification here. This is on 10000 test datapoints

|  |  |  |
| --- | --- | --- |
| **Model** | **Accuracy** | **loss** |
| MLP (100 iterations) | 0.67 | 0.16223 |
| MLP (500 iterations) | 0.78 | 0.10001 |
| MLP (1000 iteration) | 0.88 | 0.059672 |
| GMM (50 iteration) | 0.59 | 0.35030 |
| GMM (100 iteration) | 0.71 | 0.12221 |
| **GMM (150 iteration)** | **0.86** | **0.03122** |

Images of dataset after implementing classification from the bolded models in the table is provided below:

1: dataset with 100 datapoints

2: dataset with 10000 datapoints



Appendix

Helper Functions:

import h5py

import numpy as np

from random import randrange

from numpy import exp, array, random, dot

filepath\_data\_100 = 'c:/Users/Kathan/Desktop/AML/Assignment3/datafile/three\_class\_100.mat'

filepath\_labels\_100 = 'c:/Users/Kathan/Desktop/AML/Assignment3/datafile/three\_label\_100.mat'

def mat\_to\_array(filepath,flag,N,name):

    '''

    Function that converts mat file to numpy array.

    filepath: path to the mat file.

    flag: 0-data |  1-label

    n: number of samples

    name: name of mat file ins string format

    Author: Kathan Vyas

    '''

    if flag == 0:

        x\_y\_numpy\_array = np.zeros((N, 2), dtype=float)

    else:

        x\_y\_numpy\_array = np.zeros((N, 1), dtype=float)

    with h5py.File(filepath, 'r') as f:

        for idx, element in enumerate(f[name]):

            x\_y\_numpy\_array[idx] = element[:]

    return x\_y\_numpy\_array

def min\_P\_error\_classifier(sample\_size,class\_prior0,class\_prior1,dataset,orig\_label,gmean,gcov):

    #As it is min P(error) classifer, we will always take 0/1 loss

    loss = np.array([[0,1], [1,0]])

    size = sample\_size

    prior = [class\_prior0,class\_prior1]

    mean = np.zeros((2,4))

    mean[:,0] = gmean[:,0]

    mean[:,1] = gmean[:,1]

    cov = np.zeros((2,2,4))

    cov[:,:,0] = gcov[:,:,0]

    cov[:,:,1] = gcov[:,:,1]

    # Gamma/ threshold

    gamma = ((loss[1,0]-loss[0,0])/(loss[1,0] - loss[1,1])) \* (prior[0]/prior[1])

    orig\_labels = orig\_label

    new\_labels = np.zeros((1,size))

    # Calculation for discriminant score and decisions

    cond\_pdf\_class0\_log = np.log((multivariate\_normal.pdf(dataset.T,mean=mean[:,0],cov = cov[:,:,0])))

    cond\_pdf\_class1\_log = np.log((multivariate\_normal.pdf(dataset.T,mean=mean[:,1],cov = cov[:,:,1])))

    discriminant\_score = cond\_pdf\_class1\_log - cond\_pdf\_class0\_log

    new\_labels[0,:] = (discriminant\_score >= np.log(gamma)).astype(int)

    # Code to plot the distribution after Classification

    x00 = [i for i in range(new\_labels.shape[1]) if (orig\_labels[0,i] == 0 and new\_labels[0,i] == 0)]

    x01 = [i for i in range(new\_labels.shape[1]) if (orig\_labels[0,i] == 0 and new\_labels[0,i] == 1)]

    x10 = [i for i in range(new\_labels.shape[1]) if (orig\_labels[0,i] == 1 and new\_labels[0,i] == 0)]

    x11 = [i for i in range(new\_labels.shape[1]) if (orig\_labels[0,i] == 1 and new\_labels[0,i] == 1)]

    plt.plot(dataset[0,x00],dataset[1,x00],'.',color ='g')

    plt.plot(dataset[0,x01],dataset[1,x01],'.',color = 'r')

    plt.plot(dataset[0,x11],dataset[1,x11],'+',color ='g')

    plt.plot(dataset[0,x10],dataset[1,x10],'+',color = 'r')

    plt.legend(["class 0 correctly classified",'class 0 wrongly classified','class 1 correctly classified','class 1 wrongly classified'])

    plt.xlabel("Feature x1")

    plt.ylabel("Feature x2")

    plt.title('Distribution after classification')

    plt.show()

    c0 = np.argwhere(orig\_labels[0,:]==0).shape[0]

    c1 = np.argwhere(orig\_labels[0,:]==1).shape[0]

    #print("Class 0:",c0)

    #print("Class 1:",c1)

    TP = 0

    FP = 0

    TN = 0

    FN = 0

    tpr = 0

    fpr = 0

    min\_TPR = 0

    min\_FPR = 0

    TPR = []

    FPR = []

    new\_labels1 = np.zeros((1,size))

    d\_labels1 = np.zeros((1,size))

    r=map(lambda x: x/10.0,range(0,500))

    print(r)

    for i in r:

        gamma1 = i

        #print(gamma)

        new\_labels1[0,:] = (discriminant\_score >= np.log(gamma1)).astype(int)

        #d\_labels1[0,:] = discriminant\_score >= np.log(gamma)

        for i in range(new\_labels1.shape[1]):

            #print("innerforloop")

            if (orig\_labels[0,i] == 1 and new\_labels1[0,i] == 1):

               TP += 1

            if (orig\_labels[0,i] == 0 and new\_labels1[0,i] == 1):

               FP += 1

            if (orig\_labels[0,i] == 0 and new\_labels1[0,i] == 0):

               TN += 1

            if (orig\_labels[0,i] == 1 and new\_labels1[0,i] == 0):

               FN += 1

        tpr = TP / (TP+FN)

        fpr = FP / (FP+TN)

        TPR.append(tpr)

        FPR.append(fpr)

        if gamma1 == 9.00000:

            min\_TPR = tpr

            min\_FPR = fpr

    plt.plot(FPR,TPR,'-',color = 'r')

    plt.plot(min\_FPR,min\_TPR, 'g\*')

    plt.legend(["ROC Curve",'Min P Error'])

    plt.show()

    plt.close()

def cross\_validation\_split(dataset, folds=3):

    '''

    Function performs k-fold cross validation

    dataset: numpy array

    folds: number of folds

    Author: Kathan Vyas

    '''

    dataset\_split = list()

    dataset\_copy = list(dataset)

    fold\_size = int(len(dataset) / folds)

    for i in range(folds):

        fold = list()

        while len(fold) < fold\_size:

            index = randrange(len(dataset\_copy))

            fold.append(dataset\_copy.pop(index))

        dataset\_split.append(fold)

    return dataset\_split

**Code for Q1**

import h5py

import numpy as np

from random import randrange

from numpy import exp, array, random, dot

import sklearn.metrics

from fun\_lib import min\_P\_error\_classifier

np.random.seed(404)

predict\_proba = []

def mat\_to\_array(filepath,flag,N,name):

    '''

    Function that converts mat file to numpy array.

    filepath: path to the mat file.

    flag: 0-data |  1-label

    n: number of samples

    name: name of mat file ins string format

    Author: Kathan Vyas

    '''

    if flag == 0:

        x\_y\_numpy\_array = np.zeros((N, 2), dtype=float)

    else:

        x\_y\_numpy\_array = np.zeros((N, 1), dtype=float)

    with h5py.File(filepath, 'r') as f:

        for idx, element in enumerate(f[name]):

            x\_y\_numpy\_array[idx] = element[:]

    return x\_y\_numpy\_array

def cross\_validation\_split(dataset, folds=3):

    '''

    Function performs k-fold cross validation

    dataset: numpy array

    folds: number of folds

    Author: Kathan Vyas

    '''

    dataset\_split = list()

    dataset\_copy = list(dataset)

    fold\_size = int(len(dataset) / folds)

    for i in range(folds):

        fold = list()

        while len(fold) < fold\_size:

            index = randrange(len(dataset\_copy))

            fold.append(dataset\_copy.pop(index))

        dataset\_split.append(fold)

    return dataset\_split

def expandthree(df,num):

        y = np.zeros((num,3))

        for i in range(num):

            if df[[i]]==1:

                y[i,0]=1

                y[i,1]=0

                y[i,2]=0

            if df[i]==2:

                y[i,0]=0

                y[i,1]=1

                y[i,2]=0

            if df[i]==3:

                y[i,0]=0

                y[i,1]=0

                y[i,2]=1

        return y

class elu:

    @staticmethod

    def activation(z):

        alpha = 0.1

        z[z < 0] = ((alpha) \* (np.exp(z)-1))

        return z

    @staticmethod

    def prime(z):

        alpha = 0.1

        z[z < 0] = (alpha \* (np.exp(a)-1)) + alpha

        z[z > 0] = 1

        return z

class Selu:

    @staticmethod

    def activation(z):

        alpha = 1.67

        lamda = 1.050

        z[z < 0] = (lamda \* (alpha \* (np.exp(z)-1)))

        return z

    @staticmethod

    def prime(z):

        alpha = 1.67

        lamda = 1.050

        z[z < 0] = lamda \* (alpha \* np.exp(z))

        z[z > 0] = lamda

        return z

class Softmax:

    f = True

    @staticmethod

    def activation(z):

        alpha = 0.2

        z[z < 0] = z[z < 0] \* alpha

        return z

    @staticmethod

    def prime(z):

        alpha = 0.2

        # Implementing MAP classifier along with Softmax

        if not f:

            """Compute softmax values for each sets of scores in x."""

            e\_x = np.exp(x - np.max(x))

            return e\_x / e\_x.sum()

            scores = [0.0, 1.0, 2.0]

            print(Softmax(scores))

            q = np.array(3)

            q[0] = np.len(x[y==0]) / np.len(x)

            q[1] = np.len(x[y==1]) / np.len(x)

            q[2]= np.len(x[y==2]) / np.len(x)

            for l in range(2):

                z = np.argmax(scores[l]\*q[l])

        z[z < 0] = alpha

        z[z > 0] = 1

        return z

class LRelu:

    @staticmethod

    def activation(z):

        z[z < 0] = 0

        return z

    @staticmethod

    def prime(z):

        z[z < 0] = 0

        z[z > 0] = 1

        return z

class Sigmoid:

    @staticmethod

    def activation(z):

        return 1 / (1 + np.exp(-z))

    @staticmethod

    def prime(z):

        return Sigmoid.activation(z) \* (1 - Sigmoid.activation(z))

class MSE:

    def \_\_init\_\_(self, activation\_fn=None):

        """

        :param activation\_fn: Class object of the activation function.

        """

        if activation\_fn:

            self.activation\_fn = activation\_fn

        else:

            self.activation\_fn = NoActivation

    def activation(self, z):

        return self.activation\_fn.activation(z)

    @staticmethod

    def loss(y\_true, y\_pred):

        """

        :param y\_true: (array) One hot encoded truth vector.

        :param y\_pred: (array) Prediction vector

        :return: (flt)

        """

        return np.mean((y\_pred - y\_true)\*\*2)

    @staticmethod

    def prime(y\_true, y\_pred):

        return y\_pred - y\_true

    def delta(self, y\_true, y\_pred):

        """

        Back propagation error delta

        :return: (array)

        """

        return self.prime(y\_true, y\_pred) \* self.activation\_fn.prime(y\_pred)

class Network:

    def \_\_init\_\_(self, dimensions, activations):

        """

        :param dimensions: (tpl/ list) Dimensions of the neural net. (input, hidden layer, output)

        :param activations: (tpl/ list) Activations functions.

        Example of one hidden layer with

        - 2 inputs

        - 3 hidden nodes

        - 3 outputs

        layers -->    [1,        2,          3]

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

        dimensions =  (2,     3,          3)

        activations = (      Relu,      Sigmoid)

        """

        self.n\_layers = len(dimensions)

        self.loss = None

        self.learning\_rate = None

        # Weights and biases are initiated by index. For a one hidden layer net you will have a w[1] and w[2]

        self.w = {}

        self.b = {}

        # Activations are also initiated by index. For the example we will have activations[2] and activations[3]

        self.activations = {}

        for i in range(len(dimensions) - 1):

            self.w[i + 1] = np.random.randn(dimensions[i], dimensions[i + 1]) / np.sqrt(dimensions[i])

            self.b[i + 1] = np.zeros(dimensions[i + 1])

            self.activations[i + 2] = activations[i]

    def \_feed\_forward(self, x):

        """

        Execute a forward feed through the network.

        :param x: (array) Batch of input data vectors.

        :return: (tpl) Node outputs and activations per layer. The numbering of the output is equivalent to the layer numbers.

        """

        # w(x) + b

        z = {}

        # activations: f(z)

        a = {1: x}  # First layer has no activations as input. The input x is the input.

        for i in range(1, self.n\_layers):

            # current layer = i

            # activation layer = i + 1

            z[i + 1] = np.dot(a[i], self.w[i]) + self.b[i]

            a[i + 1] = self.activations[i + 1].activation(z[i + 1])

        return z, a

    def \_back\_prop(self, z, a, y\_true):

        """

        The input dicts keys represent the layers of the net.

        a = { 1: x,

              2: f(w1(x) + b1)

              3: f(w2(a2) + b2)

              }

        :param z: (dict) w(x) + b

        :param a: (dict) f(z)

        :param y\_true: (array) One hot encoded truth vector.

        :return:

        """

        # Determine partial derivative and delta for the output layer.

        # delta output layer

        delta = self.loss.delta(y\_true, a[self.n\_layers])

        dw = np.dot(a[self.n\_layers - 1].T, delta)

        update\_params = {

            self.n\_layers - 1: (dw, delta)

        }

        # In case of three layer net will iterate over i = 2 and i = 1

        # Determine partial derivative and delta for the rest of the layers.

        # Each iteration requires the delta from the previous layer, propagating backwards.

        for i in reversed(range(2, self.n\_layers)):

            delta = np.dot(delta, self.w[i].T) \* self.activations[i].prime(z[i])

            dw = np.dot(a[i - 1].T, delta)

            update\_params[i - 1] = (dw, delta)

        for k, v in update\_params.items():

            self.\_update\_w\_b(k, v[0], v[1])

    def \_update\_w\_b(self, index, dw, delta):

        """

        Update weights and biases.

        :param index: (int) Number of the layer

        :param dw: (array) Partial derivatives

        :param delta: (array) Delta error.

        """

        self.w[index] -= self.learning\_rate \* dw

        self.b[index] -= self.learning\_rate \* np.mean(delta, 0)

    def fit(self, x, y\_true, loss, epochs, batch\_size, learning\_rate=1e-3):

        """

        :param x: (array) Containing parameters

        :param y\_tue: (array) Containing one hot encoded labels.

        :param loss: Loss class (MSE, CrossEntropy etc.)

        :param epochs: (int) Number of epochs.

        :param batch\_size: (int)

        :param learning\_rate: (flt)

        """

        if not x.shape[0] == y\_true.shape[0]:

            raise ValueError("Length of x and y arrays don't match")

        # Initiate the loss object with the final activation function

        self.loss = loss(self.activations[self.n\_layers])

        self.learning\_rate = learning\_rate

        for i in range(epochs):

            # Shuffle the data

            seed = np.arange(x.shape[0])

            np.random.shuffle(seed)

            x\_ = x[seed]

            y\_ = y\_true[seed]

            for j in range(x.shape[0] // batch\_size):

                k = j \* batch\_size

                l = (j + 1) \* batch\_size

                z, a = self.\_feed\_forward(x\_[k:l])

                self.\_back\_prop(z, a, y\_[k:l])

            if (i + 1) % 10 == 0:

                \_, a = self.\_feed\_forward(x)

                print("Loss:", self.loss.loss(y\_true, a[self.n\_layers]))

    def predict(self, x):

        """

        :param x: (array) Containing parameters

        :return: (array) A 2D array of shape (n\_cases, n\_classes).

        """

        \_, a = self.\_feed\_forward(x)

        return a[self.n\_layers]

if \_\_name\_\_ == "\_\_main\_\_":

    filepath\_data\_100 = 'c:/Users/Kathan/Desktop/AML/Assignment3/datafile/three\_class\_100.mat'

    filepath\_labels\_100 = 'c:/Users/Kathan/Desktop/AML/Assignment3/datafile/three\_label\_100.mat'

    filepath\_data\_10k = 'c:/Users/Kathan/Desktop/AML/Assignment3/datafile/three\_class\_10k.mat'

    x = mat\_to\_array(filepath\_data\_100,0,100,'three\_class\_100')

    \_y\_ = mat\_to\_array(filepath\_labels\_100,1,100,'three\_label\_100')

    x\_test = mat\_to\_array(filepath\_data\_10k,0,10000,'three\_class\_10k')

    # one hot encoding

    y = expandthree(\_y\_,100)

    nn = Network((2, 6, 3), (Sigmoid, Softmax))

    nn.fit(x, y, loss=MSE, epochs=1000, batch\_size=1, learning\_rate=0.01)

    prediction = nn.predict(x)

    y\_true = []

    y\_pred = []

    for i in range(len(y)):

        y\_pred.append(np.argmax(prediction[i]))

        y\_true.append(np.argmax(y[i]))

    print(sklearn.metrics.classification\_report(y\_true, y\_pred))

    preds = nn.predict(x\_test)

    y\_pred1 = []

    for i in range(len(x\_test)):

        y\_pred1.append(np.argmax(preds[i]))

    y\_test = y\_pred1

    #print(np.shape(y\_test))

    #print(np.unique(y\_test))

**Code for Q2**

#%%

import matplotlib.pyplot as plt

import h5py

from matplotlib import style

style.use('fivethirtyeight')

from sklearn.datasets.samples\_generator import make\_blobs

import numpy as np

from scipy.stats import multivariate\_normal

np.random.seed(404)

filepath\_data\_100 = 'c:/Users/Kathan/Desktop/AML/Assignment3/datafile/three\_class\_100.mat'

filepath\_labels\_100 = 'c:/Users/Kathan/Desktop/AML/Assignment3/datafile/three\_label\_100.mat'

filepath\_data\_10k = 'c:/Users/Kathan/Desktop/AML/Assignment3/datafile/three\_class\_10k.mat'

def mat\_to\_array(filepath,flag,N,name):

    '''

    Function that converts mat file to numpy array.

    filepath: path to the mat file.

    flag: 0-data |  1-label

    n: number of samples

    name: name of mat file ins string format

    Author: Kathan Vyas

    '''

    if flag == 0:

        x\_y\_numpy\_array = np.zeros((N, 2), dtype=float)

    else:

        x\_y\_numpy\_array = np.zeros((N, 1), dtype=float)

    with h5py.File(filepath, 'r') as f:

        for idx, element in enumerate(f[name]):

            x\_y\_numpy\_array[idx] = element[:]

    return x\_y\_numpy\_array

def cross\_validation\_split(dataset, folds=3):

    '''

    Function performs k-fold cross validation

    dataset: numpy array

    folds: number of folds

    Author: Kathan Vyas

    '''

    dataset\_split = list()

    dataset\_copy = list(dataset)

    fold\_size = int(len(dataset) / folds)

    for i in range(folds):

        fold = list()

        while len(fold) < fold\_size:

            index = randrange(len(dataset\_copy))

            fold.append(dataset\_copy.pop(index))

        dataset\_split.append(fold)

    return dataset\_split

x\_test = mat\_to\_array(filepath\_data\_10k,0,10000,'three\_class\_10k')

# 0. Create dataset

X = mat\_to\_array(filepath\_data\_100,0,100,'three\_class\_100')

Y = mat\_to\_array(filepath\_labels\_100,1,100,'three\_label\_100')

# Stratch dataset to get ellipsoid data

#X = np.dot(X,np.random.RandomState(0).randn(2,2))

#%%

class GMM:

    def \_\_init\_\_(self,X,number\_of\_sources,iterations):

        self.iterations = iterations

        self.number\_of\_sources = number\_of\_sources

        self.X = X

        self.mu = None

        self.pi = None

        self.cov = None

        self.XY = None

    """Define a function which runs for iterations, iterations"""

    def run(self):

        self.reg\_cov = 1e-6\*np.identity(len(self.X[0]))

        x,y = np.meshgrid(np.sort(self.X[:,0]),np.sort(self.X[:,1]))

        self.XY = np.array([x.flatten(),y.flatten()]).T

        """ 1. Set the initial mu, covariance and pi values"""

        self.mu = np.random.randint(min(self.X[:,0]),max(self.X[:,0]),size=(self.number\_of\_sources,len(self.X[0]))) # This is a nxm matrix since we assume n sources (n Gaussians) where each has m dimensions

        self.cov = np.zeros((self.number\_of\_sources,len(X[0]),len(X[0]))) # We need a nxmxm covariance matrix for each source since we have m features --> We create symmetric covariance matrices with ones on the digonal

        for dim in range(len(self.cov)):

            np.fill\_diagonal(self.cov[dim],5)

        self.pi = np.ones(self.number\_of\_sources)/self.number\_of\_sources # Are "Fractions"

        log\_likelihoods = [] # In this list we store the log likehoods per iteration and plot them in the end to check if

                             # if we have converged

        """Plot the initial state"""

        fig = plt.figure(figsize=(10,10))

        ax0 = fig.add\_subplot(111)

        ax0.scatter(self.X[:,0],self.X[:,1])

        ax0.set\_title('Initial state')

        for m,c in zip(self.mu,self.cov):

            c += self.reg\_cov

            multi\_normal = multivariate\_normal(mean=m,cov=c)

            ax0.contour(np.sort(self.X[:,0]),np.sort(self.X[:,1]),multi\_normal.pdf(self.XY).reshape(len(self.X),len(self.X)),colors='black',alpha=0.3)

            ax0.scatter(m[0],m[1],c='grey',zorder=10,s=100)

        for i in range(self.iterations):

            """E Step"""

            r\_ic = np.zeros((len(self.X),len(self.cov)))

            for m,co,p,r in zip(self.mu,self.cov,self.pi,range(len(r\_ic[0]))):

                co+=self.reg\_cov

                mn = multivariate\_normal(mean=m,cov=co)

                r\_ic[:,r] = p\*mn.pdf(self.X)/np.sum([pi\_c\*multivariate\_normal(mean=mu\_c,cov=cov\_c).pdf(X) for pi\_c,mu\_c,cov\_c in zip(self.pi,self.mu,self.cov+self.reg\_cov)],axis=0)

            """

            The above calculation of r\_ic is not that obvious why I want to quickly derive what we have done above.

            First of all the nominator:

            We calculate for each source c which is defined by m,co and p for every instance x\_i, the multivariate\_normal.pdf() value.

            For each loop this gives us a 100x1 matrix (This value divided by the denominator is then assigned to r\_ic[:,r] which is in

            the end a 100x3 matrix).

            Second the denominator:

            What we do here is, we calculate the multivariate\_normal.pdf() for every instance x\_i for every source c which is defined by

            pi\_c, mu\_c, and cov\_c and write this into a list. This gives us a 3x100 matrix where we have 100 entrances per source c.

            Now the formula wants us to add up the pdf() values given by the 3 sources for each x\_i. Hence we sum up this list over axis=0.

            This gives us then a list with 100 entries.

            What we have now is FOR EACH LOOP a list with 100 entries in the nominator and a list with 100 entries in the denominator

            where each element is the pdf per class c for each instance x\_i (nominator) respectively the summed pdf's of classes c for each

            instance x\_i. Consequently we can now divide the nominator by the denominator and have as result a list with 100 elements which we

            can then assign to r\_ic[:,r] --> One row r per source c. In the end after we have done this for all three sources (three loops)

            and run from r==0 to r==2 we get a matrix with dimensionallity 100x3 which is exactly what we want.

            If we check the entries of r\_ic we see that there mostly one element which is much larger than the other two. This is because

            every instance x\_i is much closer to one of the three gaussians (that is, much more likely to come from this gaussian) than

            it is to the other two. That is practically speaing, r\_ic gives us the fraction of the probability that x\_i belongs to class

            c over the probability that x\_i belonges to any of the classes c (Probability that x\_i occurs given the 3 Gaussians).

            """

            """M Step"""

            # Calculate the new mean vector and new covariance matrices, based on the probable membership of the single x\_i to classes c --> r\_ic

            self.mu = []

            self.cov = []

            self.pi = []

            log\_likelihood = []

            for c in range(len(r\_ic[0])):

                m\_c = np.sum(r\_ic[:,c],axis=0)

                mu\_c = (1/m\_c)\*np.sum(self.X\*r\_ic[:,c].reshape(len(self.X),1),axis=0)

                self.mu.append(mu\_c)

                # Calculate the covariance matrix per source based on the new mean

                self.cov.append(((1/m\_c)\*np.dot((np.array(r\_ic[:,c]).reshape(len(self.X),1)\*(self.X-mu\_c)).T,(self.X-mu\_c)))+self.reg\_cov)

                # Calculate pi\_new which is the "fraction of points" respectively the fraction of the probability assigned to each source

                self.pi.append(m\_c/np.sum(r\_ic)) # Here np.sum(r\_ic) gives as result the number of instances. This is logical since we know

                                                # that the columns of each row of r\_ic adds up to 1. Since we add up all elements, we sum up all

                                                # columns per row which gives 1 and then all rows which gives then the number of instances (rows)

                                                # in X --> Since pi\_new contains the fractions of datapoints, assigned to the sources c,

                                                # The elements in pi\_new must add up to 1

            """Log likelihood"""

            log\_likelihoods.append(np.log(np.sum([k\*multivariate\_normal(self.mu[i],self.cov[j]).pdf(X) for k,i,j in zip(self.pi,range(len(self.mu)),range(len(self.cov)))])))

            #print(log\_likelihoods)

            print(np.average(log\_likelihoods))

            """

            This process of E step followed by a M step is now iterated a number of n times. In the second step for instance,

            we use the calculated pi\_new, mu\_new and cov\_new to calculate the new r\_ic which are then used in the second M step

            to calculat the mu\_new2 and cov\_new2 and so on....

            """

        fig2 = plt.figure(figsize=(10,10))

        ax1 = fig2.add\_subplot(111)

        ax1.set\_title('Log-Likelihood')

        ax1.plot(range(0,self.iterations,1),log\_likelihoods)

        #plt.show()

    """Predict the membership of an unseen, new datapoint"""

    def predict(self,Y):

        # PLot the point onto the fittet gaussians

        fig3 = plt.figure(figsize=(10,10))

        ax2 = fig3.add\_subplot(111)

        ax2.scatter(self.X[:,0],self.X[:,1])

        for m,c in zip(self.mu,self.cov):

            multi\_normal = multivariate\_normal(mean=m,cov=c)

            ax2.contour(np.sort(self.X[:,0]),np.sort(self.X[:,1]),multi\_normal.pdf(self.XY).reshape(len(self.X),len(self.X)),colors='black',alpha=0.3)

            ax2.scatter(m[0],m[1],c='grey',zorder=10,s=100)

            ax2.set\_title('Final state')

            for y in Y:

                ax2.scatter(y[0],y[1],c='orange',zorder=10,s=100)

        prediction = []

        for m,c in zip(self.mu,self.cov):

            #print(c)

            prediction.append(multivariate\_normal(mean=m,cov=c).pdf(Y)/np.sum([multivariate\_normal(mean=mean,cov=cov).pdf(Y) for mean,cov in zip(self.mu,self.cov)]))

        #plt.show()

        return prediction

#%%

GMM = GMM(X,3,150)

GMM.run()

y\_test = GMM.predict(x\_test)

# %%

p = GMM.predict(X)

# %%

Citations and References:

Code and explanation for the code have been largely influenced by from following blogs and code linked here:

* Question 1:
  + <https://machinelearningmastery.com/k-fold-cross-validation/>
  + <https://www.ritchievink.com/blog/2017/07/10/programming-a-neural-network-from-scratch/>
* Question 2:
  + <https://towardsdatascience.com/gaussian-mixture-modelling-gmm-833c88587c7f>

Github code link: <https://github.com/kathanvyas/EECE5644_Assign3.git> (additional figures provided here)