EECE5644 Fall 2021 – Take Home Exam 3

Balaji Sundareshan

Question 1

Data Distribution:

In this problem, I trained a 2-layer multi-layer perceptron (MLP) to approximate class label posteriors using maximum likelihood parameter estimation. The trained models were used to achieve minimum probability of error on a validation dataset.

A 3-dimensional real-valued random vector was generated from 4 Gaussian class conditional pdfs with uniform priors. The mean and covariance used to generate gaussian distributions for each class are shown below:

$$m_1 = \begin{bmatrix} 9.75 \\ 0.00 \\ 0.00 \end{bmatrix} \quad C_1 = \begin{bmatrix} 10.00 & 0.00 & 0.00 \\ 0.00 & 40.0 & 0.0 \\ 0.0 & 0.0 & 15.00 \end{bmatrix}$$

$$m_2 = \begin{bmatrix} 7.5 \\ 0.0 \\ 9.0 \end{bmatrix} \quad C_2 = \begin{bmatrix} 20.00 & 0.00 & 0.00 \\ 0.00 & 5.00 & 0.0 \\ 0.00 & 0.0 & 10.00 \end{bmatrix}$$

$$m_3 = \begin{bmatrix} 0.0 \\ 7.5 \\ 9.0 \end{bmatrix} \quad C_3 = \begin{bmatrix} 20.00 & 0.00 & 0.00 \\ 0.00 & 10.00 & 0.00 \\ 0.00 & 0.00 & 40.00 \end{bmatrix}$$

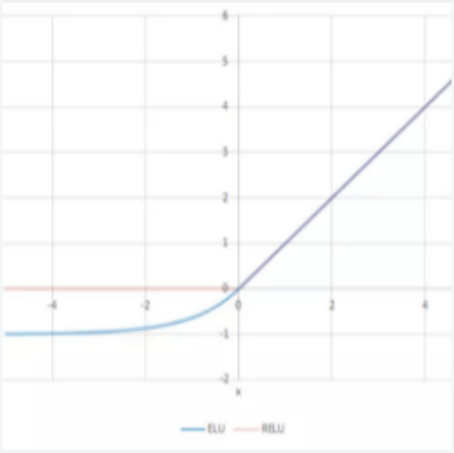
$$m_4 = \begin{bmatrix} 8.25 \\ 9. \\ 7.5 \end{bmatrix} \quad C_4 = \begin{bmatrix} 5.00 & 0.00 & 0.00 \\ 0.00 & 20.00 & 0.00 \\ 0.00 & 0.00 & 5.00 \end{bmatrix}$$

$$P(L=0) = 0.25 \; P(L=1) = 0.25 \; P(L=2) = 0.25 \; P(L=3) = 0.25$$

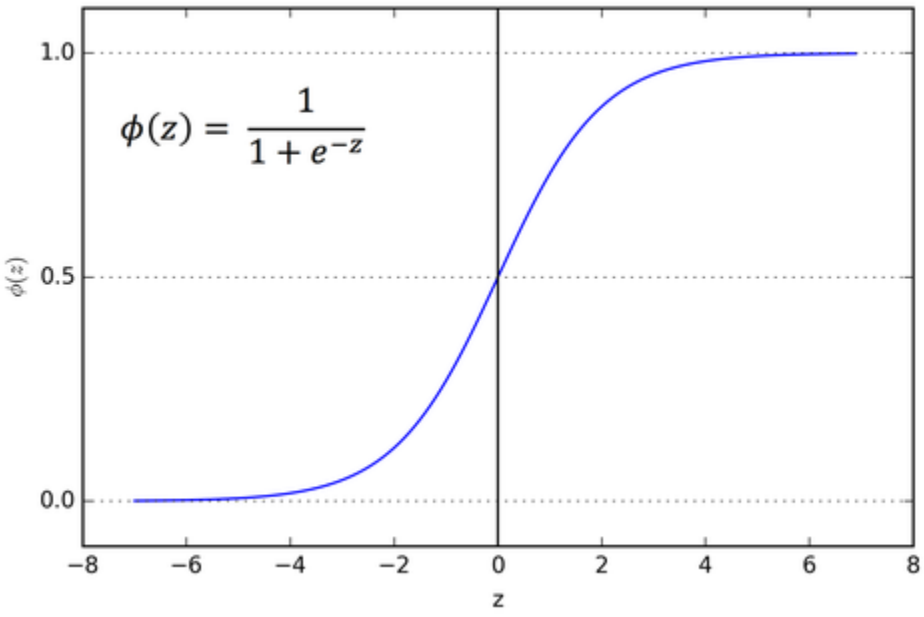
MLP Structure:

I implemented a 2-layer MLP with one hidden layer and one output layer. For a smooth-ramp style activation function as the activation layer for the hidden layer, I used ELU activation. ELU activation function and the plot is provided below.

$$R(z) = \begin{cases} z & z > 0 \\ \alpha.(e^z - 1) & z <= 0 \end{cases}$$



Softmax was used as the activation function for the output layer in order to make sure that all outputs are positive and add up to 1. Softmax activation function and the plot is provided below



Generate data:

Following datasets were generated for training.

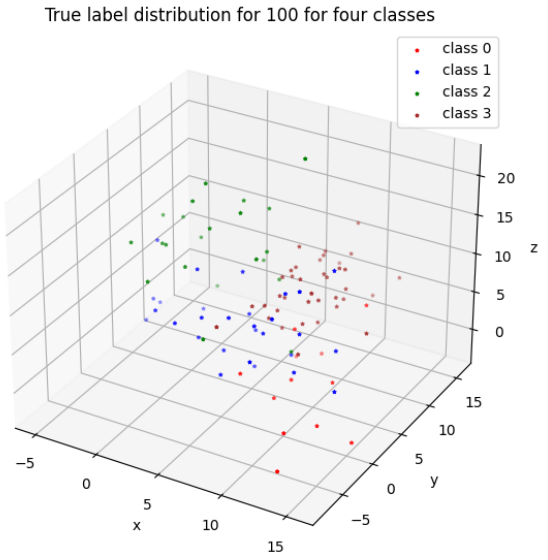
1. T100 contains 100 training data samples
2. T200 contains 200 training data samples
3. T500 contains 500 training data samples

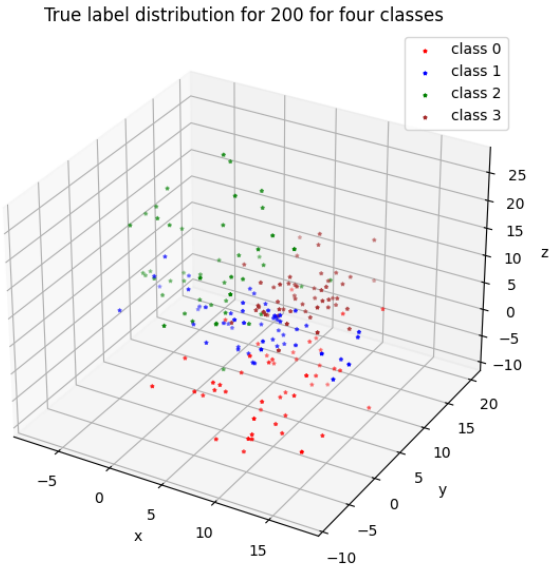
4. T1k contains 1000 training data samples
5. T2k contains 2000 training data samples
6. T5k contains 5000 training data samples

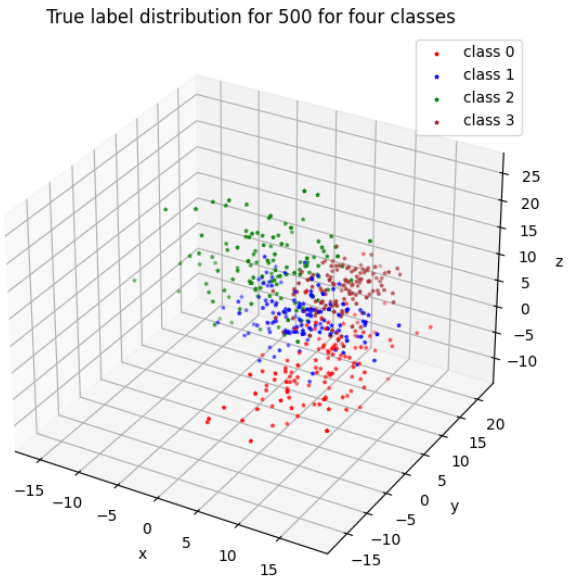
Following datasets were generated for validation.

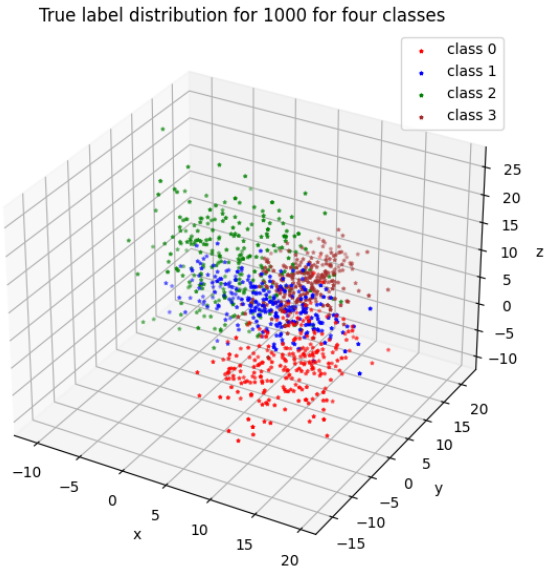
1. V100k contains 100,000 training data samples

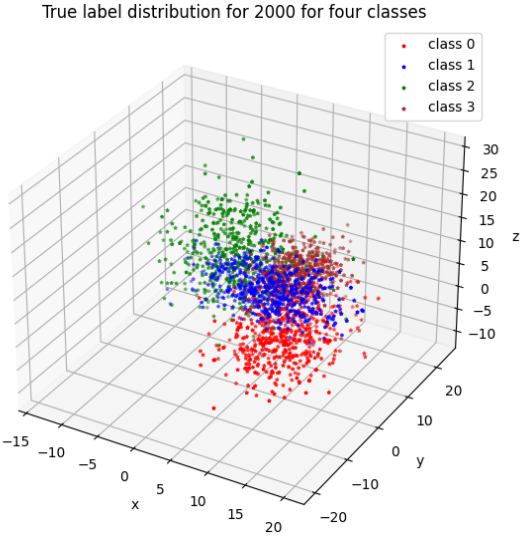
Plots generated for the above mentioned training and validation datasets are shown below.

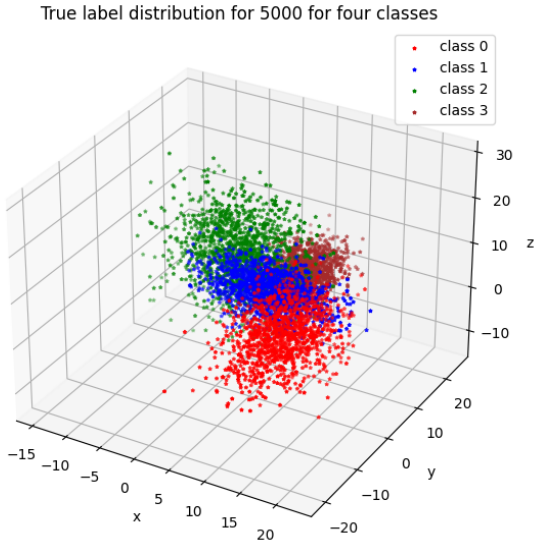


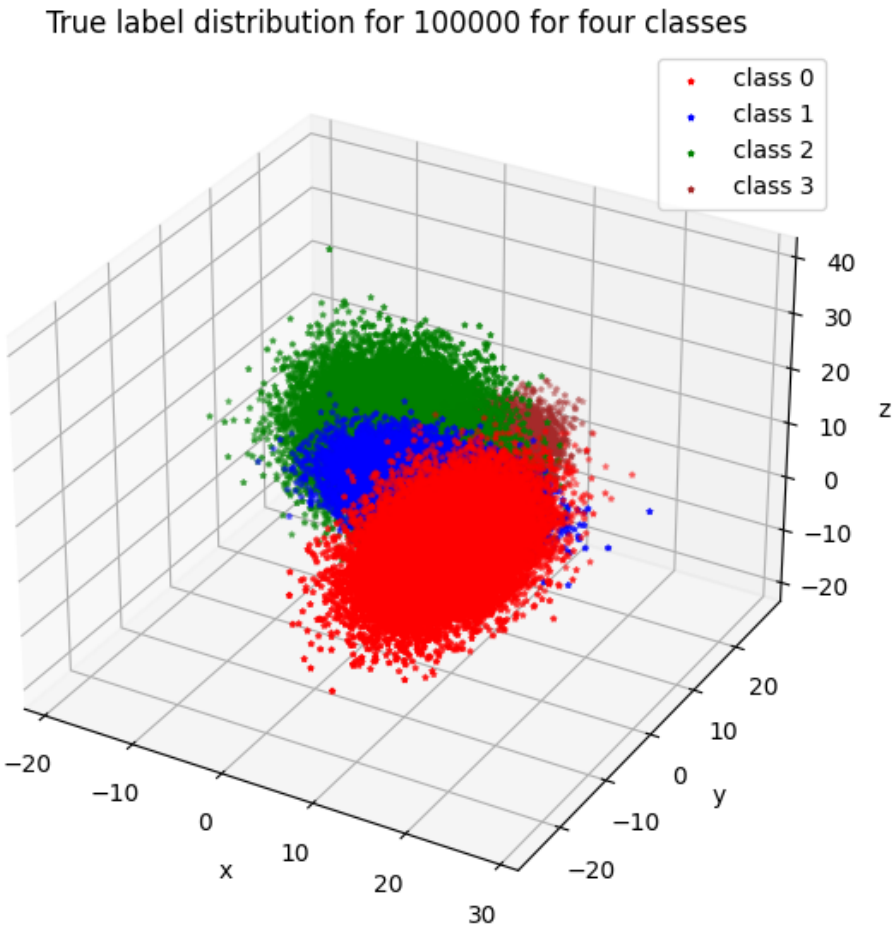












Theoretically optimal classifier:

Below is the decision rule that achieves minimum probability of error:

$$D(x) = argmin \sum_{d,l=1}^{3} \lambda_{dl} p(L = l|x)$$

λ_{dl} is the loss for classifying a point from label l in class d
p(L = l|x) is the class posteriors which are calculated as shown below

$$p(L = l|x) = \frac{p(x|L = l)p(L = l)}{p(x)}$$

where *p(x|L = l)* is the class conditional pdf; *p(L = l)* is the class prior and

$$p(x) = \sum_{j=1}^{C} p(x|L = j)p(L = j)$$

The cost function used in this classification was chosen to minimize the probability of error and the loss matrix is shown below

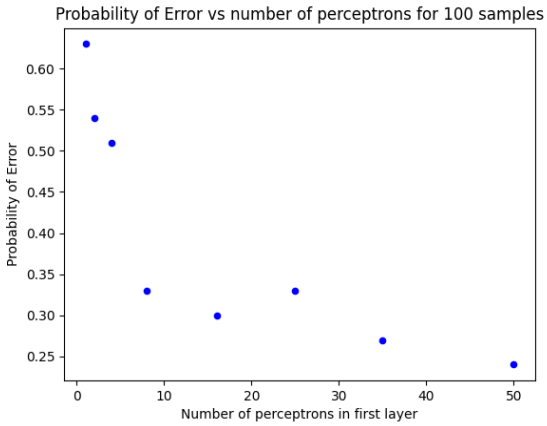
$$\lambda = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

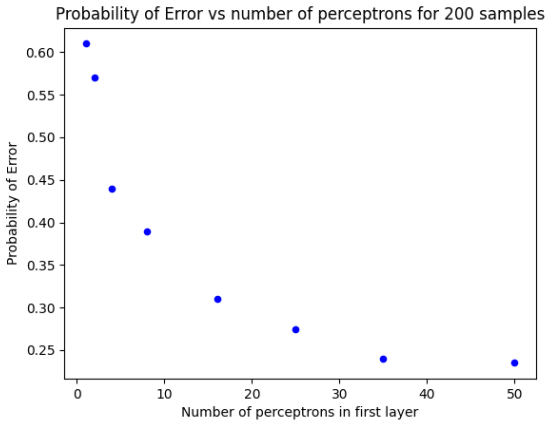
The minimum probability of error using the theoretically optimal classifier on the test data of 100, 000 samples is **0.137.**

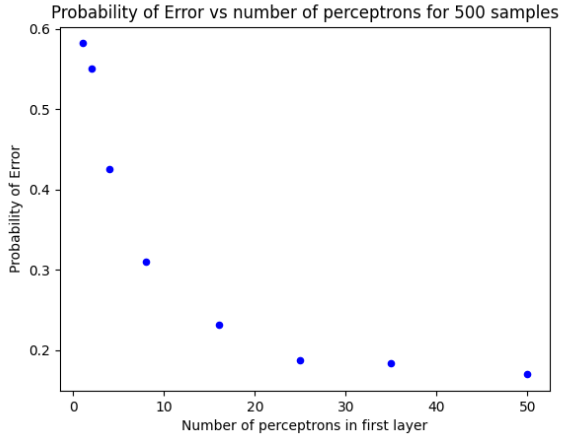
Model Order Selection:

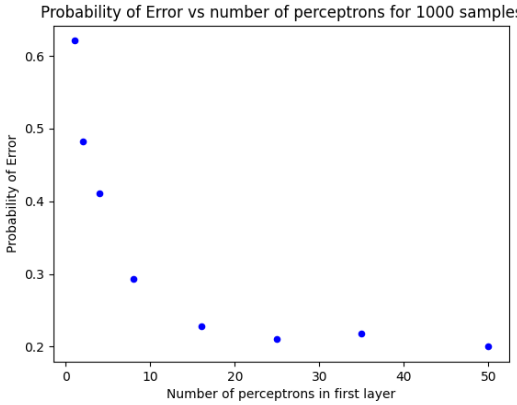
I performed 10-fold cross-validation to find the best number of perceptrons for each of the above-mentioned datasets. The objective function used to minimize was classification error probability.

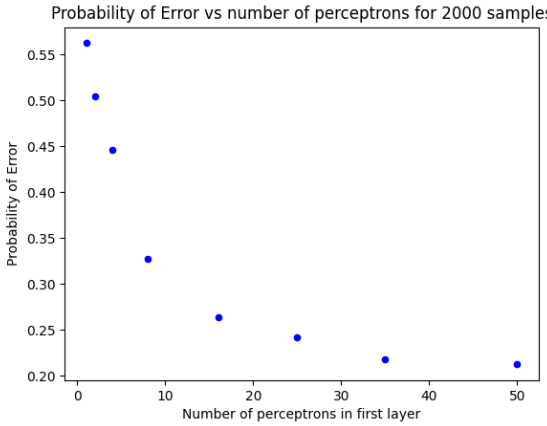
Below plot shows variation in probability of error with increase in number of perceptrons for all the above mentioned datasets.

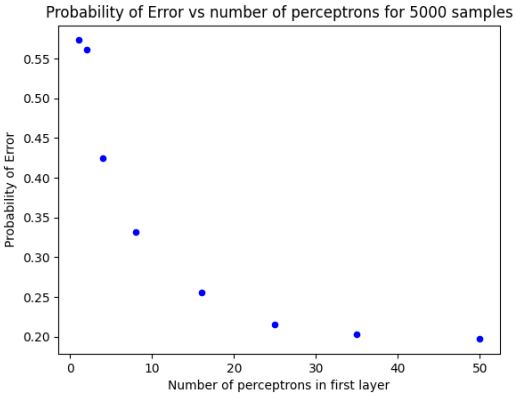












Observation:

1. Though the probability of error is lowest for the number of perceptrons = 50 across each of the dataset, the difference in mean probability of error among perceptrons 35 and 50 is less significant.

2. This difference in mean probability of error among perceptrons 35 and 50 reduces considerably as the number of training samples increases.
3. Below table shows the mean and standard deviation of probability of error for number of perceptrons 35 and 50 for each of the training dataset.
4. The standard deviation provides us with the confidence of probability of error across the mean. Lower the standard deviation, more confident the model is.
5. As the difference in the standard deviation is less, the optimal number of perceptrons were chosen based on the difference in the mean.
6. To decide how close two numbers a and b are, the following measure was used.
   (a-b)/(a+b)/2
7. Low values from this measure are highlighted in the below table.

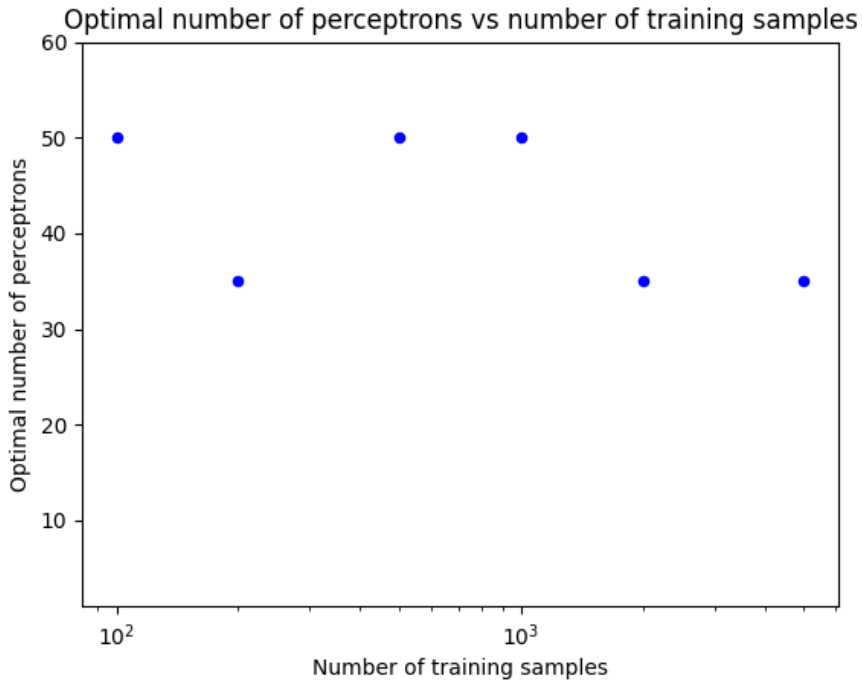
Mean of probability of error:

Training Datasets	Number of perceptrons = 35	Number of perceptrons = 50	Measure (a-b)/(a+b)/2	Optimal number of perceptrons
T100	0.27	0.24	0.11	50
T200	0.24	0.235	**0.021**	35
T500	0.184	0.17	0.07	50
T1k	0.218	0.2	0.08	50
T2k	0.218	0.212	**0.02**	35
T5k	0.202	0.197	**0.025**	35

Standard deviation of probability of error:

Training Datasets	Number of perceptrons = 35	Number of perceptrons = 50
T100	0.11	0.08
T200	0.08	0.08
T500	0.04	0.03
T1k	0.020	0.019
T2k	0.030	0.025
T5k	0.024	0.023

Below plot shows the optimal number of perceptrons for each of the training dataset.



Model training, Performance Assessment and Results:

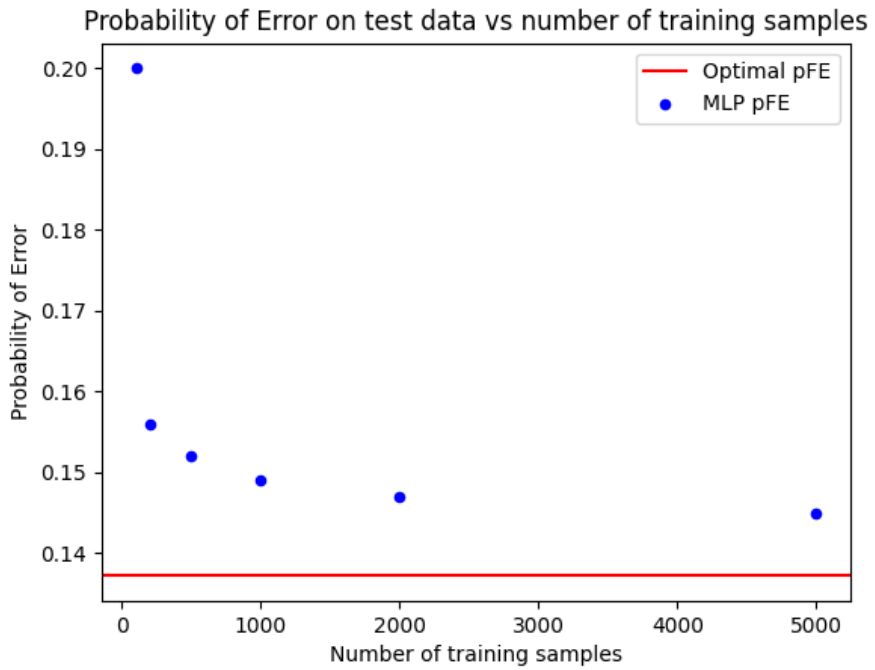
Using model order selection, the number of perceptrons in the first layer which provides minimum probability of error was identified for each dataset. Using this optimal number of perceptrons, MLP was trained for each dataset and validated on the test dataset containing 100, 000 samples.

To mitigate the chances of getting stuck at a local optimum, MLP for each dataset was trained 10 times with an optimal number of perceptrons in the first layer of the MLP, each time randomly initializing the weights of the MLP layers. For each dataset, trained MLP which provides minimum cross entropy loss (highest training data log-likelihood) across 10 experiments was used to validate on the test data.

Below table contains minimum cross entropy loss achievable across 10 training and their corresponding training accuracy for each of the training dataset.

Datasets	Cross Entropy Loss	Training Accuracy
100	0.3109	89%
200	0.3685	87.5%
500	0.3504	87.4%
1000	0.3718	87.3%
2000	0.3859	85.35%
5000	0.3874	85.7%

Below plot contains variation in probability of error for MLPs trained on each of the above datasets with optimal number of perceptrons in the first layer of the MLP and validated on the test set.



Observation:

1. The probability of error is well correlated with the number of data samples in the training dataset.
2. As the number of data samples increases, the probability of error decreases and approaches the theoretically optimal probability of error which was estimated using the true pdf of the underlying data.

3. This shows that training MLP on more number of samples increases the accuracy of the model estimate.

Implementation and Tools used:

Keras library was used to create 2 layer MLP. The kernel weights were uniformly initialized for both the layers. Stochastic Gradient Descent (SGD) was used as the optimizer. Each MLP model was trained for 100 epochs with a batch size of 10. For each epoch, the training loss is calculated using cross entropy loss. The cross entropy loss is the same as negative log likelihood of the data.

Question 2

In this problem, a 2-dimensional real-valued random vector was generated using a Gaussian Mixture Model (GMM) with 4 components. Each of these components have different mean vectors, different covariance matrices and different prior.

The prior, mean and covariance for these four components are mentioned below

$$m_1 = \begin{bmatrix} 35 \\ 0 \end{bmatrix} \quad C_1 = \begin{bmatrix} 10.00 & 20.00 \\ 0.00 & 40.00 \end{bmatrix}$$

$$m_2 = \begin{bmatrix} 35 \\ 52.5 \end{bmatrix} \quad C_2 = \begin{bmatrix} 20.00 & 0.00 \\ 40.00 & 5.00 \end{bmatrix}$$

$$m_3 = \begin{bmatrix} 0.00 \\ 0.00 \end{bmatrix} \quad C_3 = \begin{bmatrix} 20.00 & 30.00 \\ 0.00 & 10.00 \end{bmatrix}$$

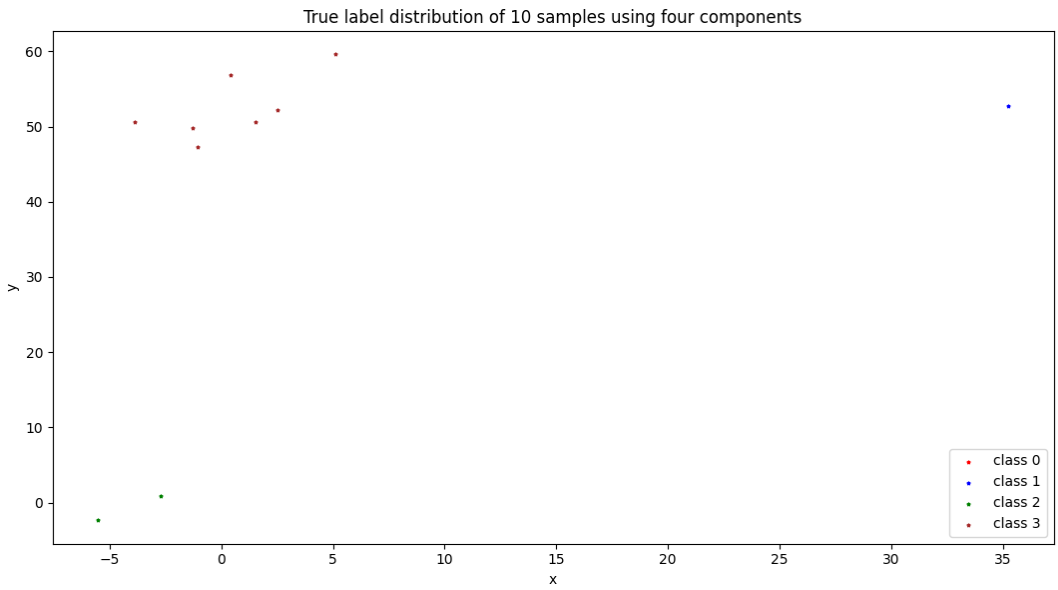
$$m_4 = \begin{bmatrix} 0.00 \\ 52.5 \end{bmatrix} \quad C_4 = \begin{bmatrix} 5.00 & 0.00 \\ 10.00 & 20.00 \end{bmatrix}$$

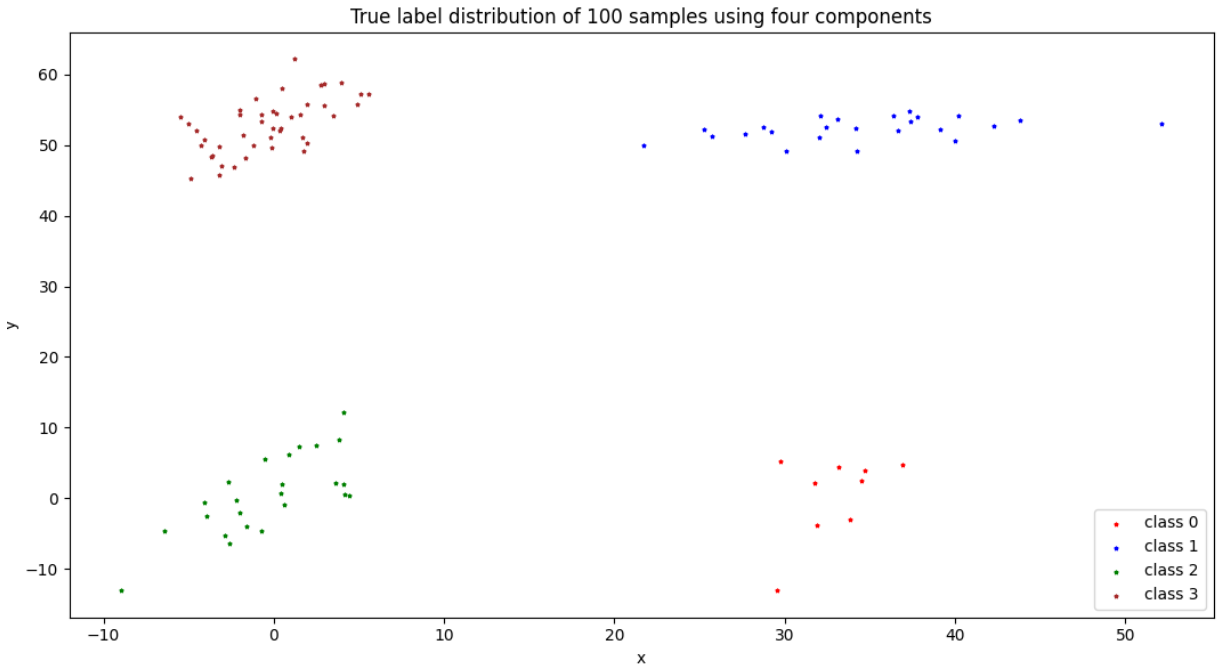
$$P(L = 0) = 0.1 \; P(L = 1) = 0.2 \; P(L = 2) = 0.3 \; P(L = 3) = 0.4$$

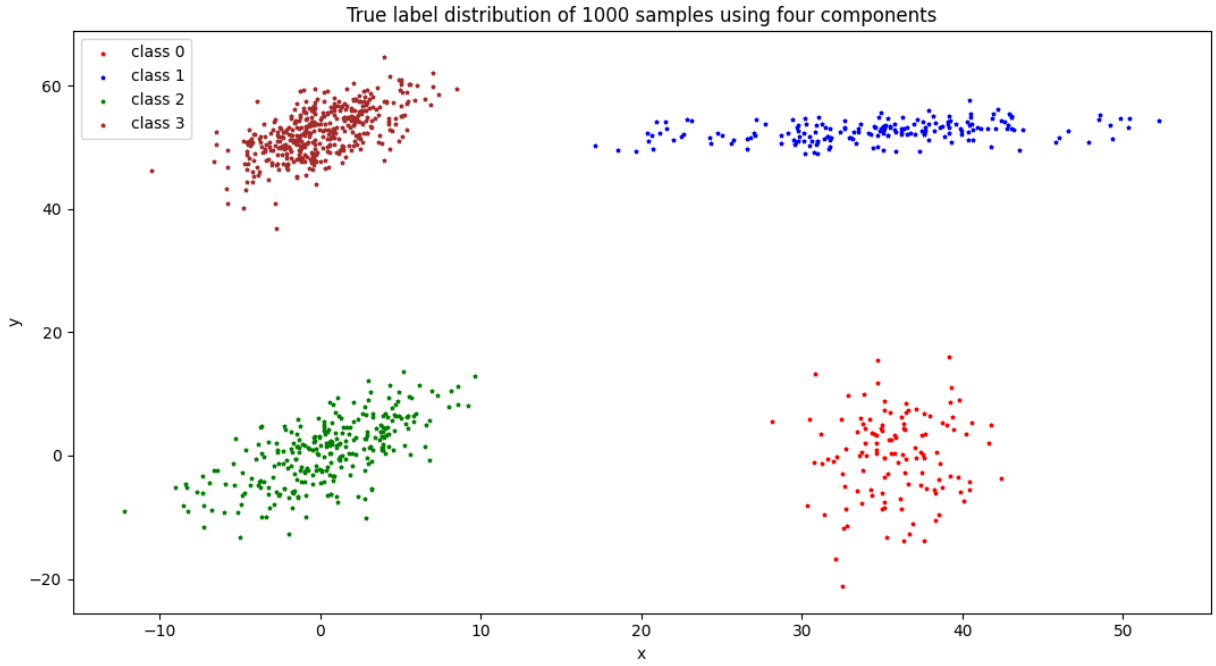
Following datasets were generated using this true GMM for training.

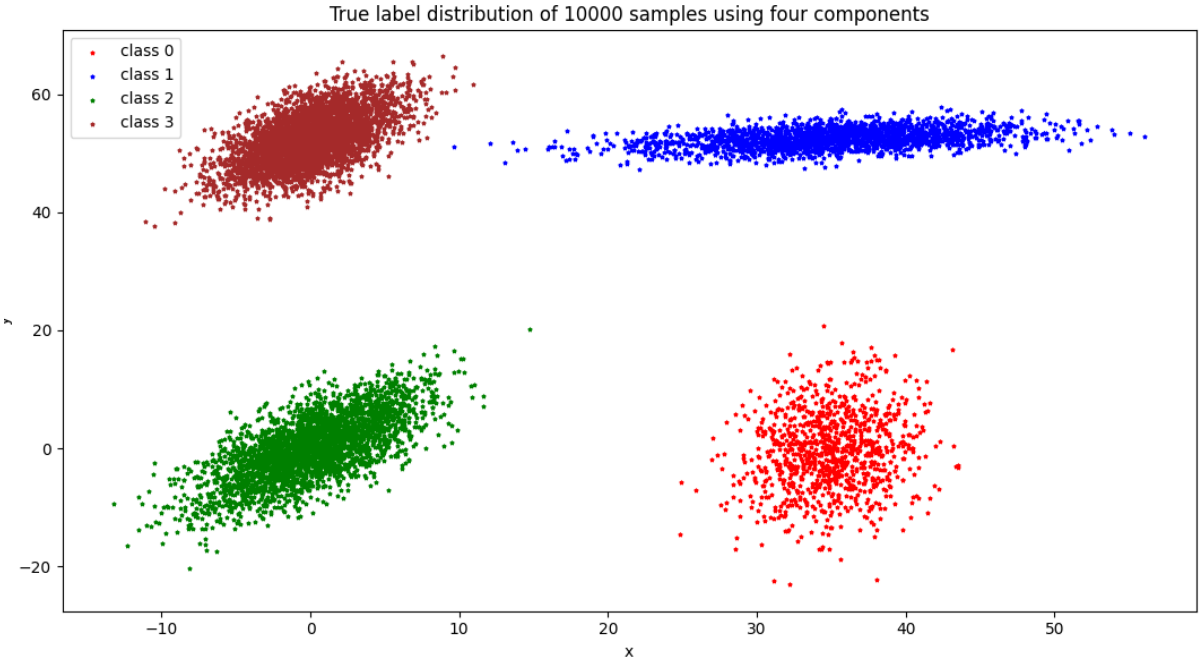
1. T10 contains 10 training data samples
2. T100 contains 100 training data samples
3. T1k contains 1000 training data samples
4. T10k contains 10000 training data samples

Plots generated for the above mentioned training datasets are shown below.


True label distribution of 10 samples using four components
class 0
class 1
class 2
class 3
x
y


True label distribution of 100 samples using four components
class 0
class 1
class 2
class 3
x
y





Model Order Selection:

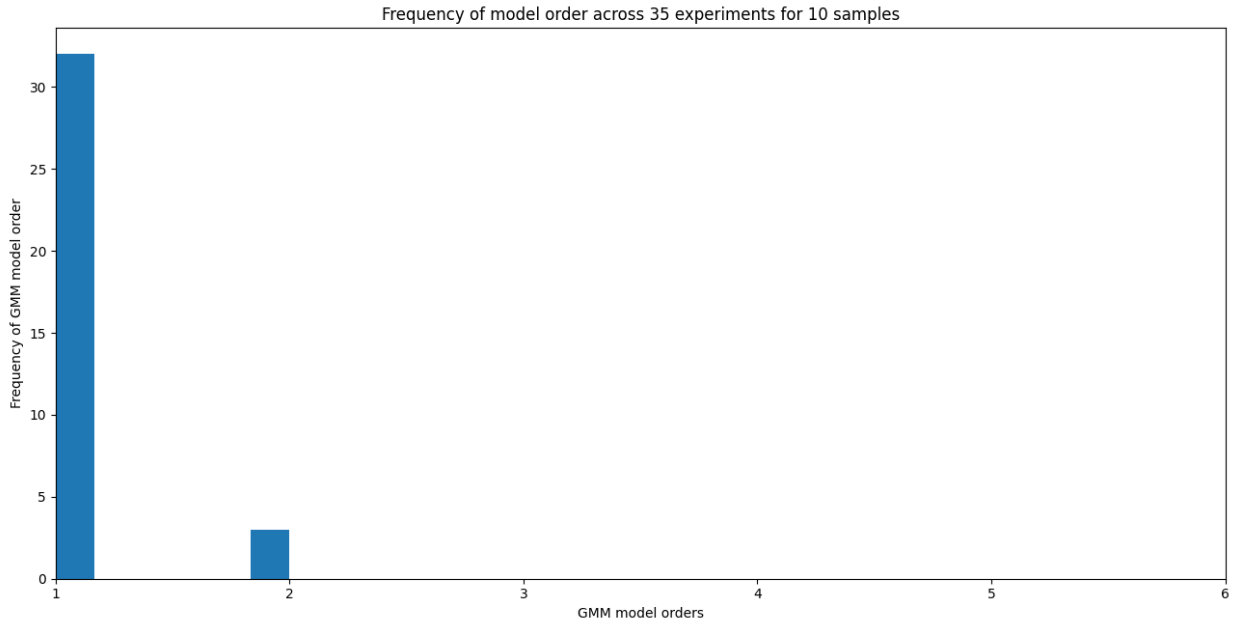
I performed 10-fold cross-validation to find the best number of GMM components for each of the above-mentioned datasets. The number of GMM components used for model selection were 1, 2, 3, 4, 5, and 6.

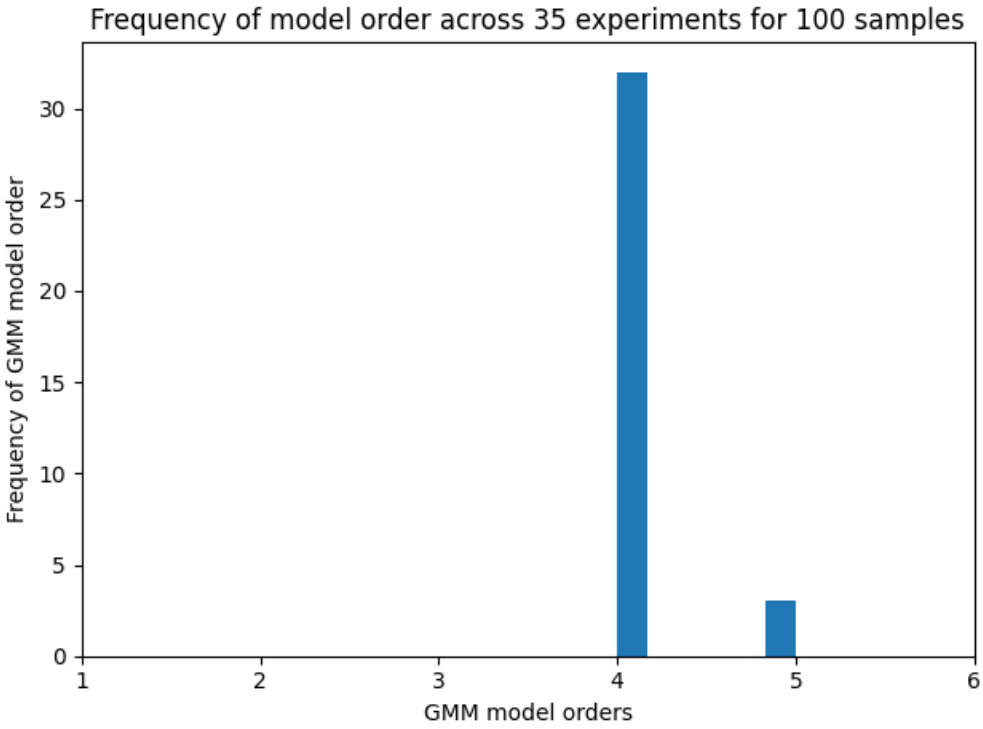
Parameter estimation for each Gaussian component was performed using the built-in function sklearn.mixture.GaussianMixture in python. The iterative numerical optimization method used here is **Expectation-Maximization**(EM) algorithm. This algorithm maximizes the expected value of the log likelihood function of θ as shown below.

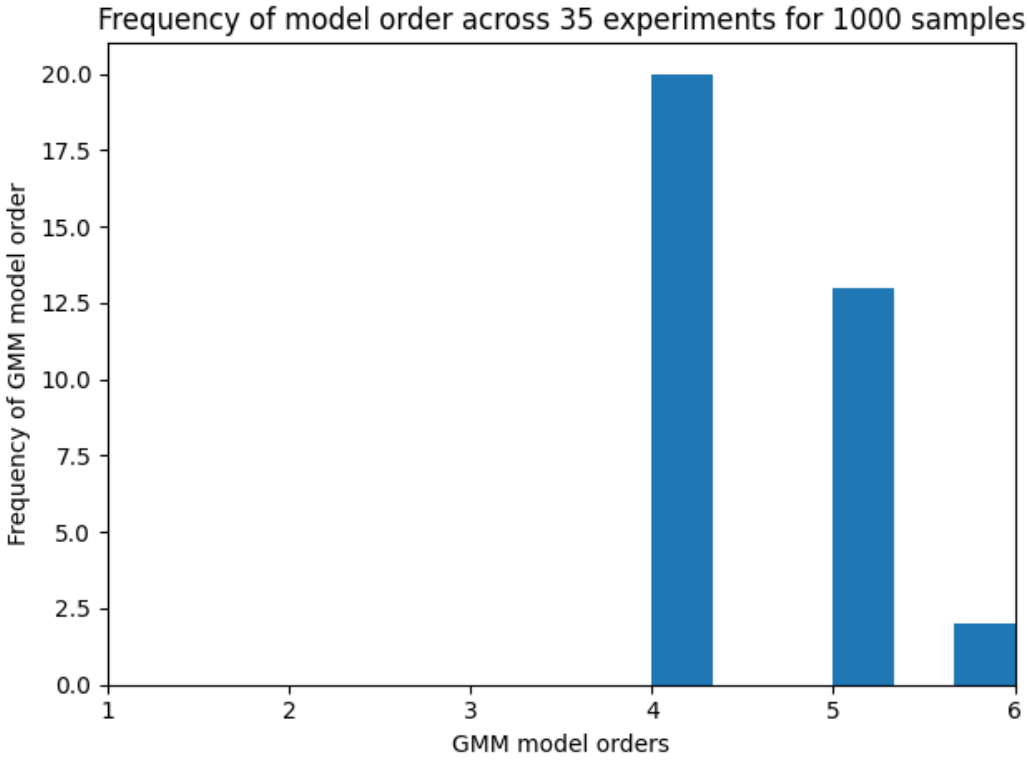
$$Q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{(t)}) = \mathrm{E}_{\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{(t)}} \left[\log L(\boldsymbol{\theta}; \mathbf{X}, \mathbf{Z}) \right]$$

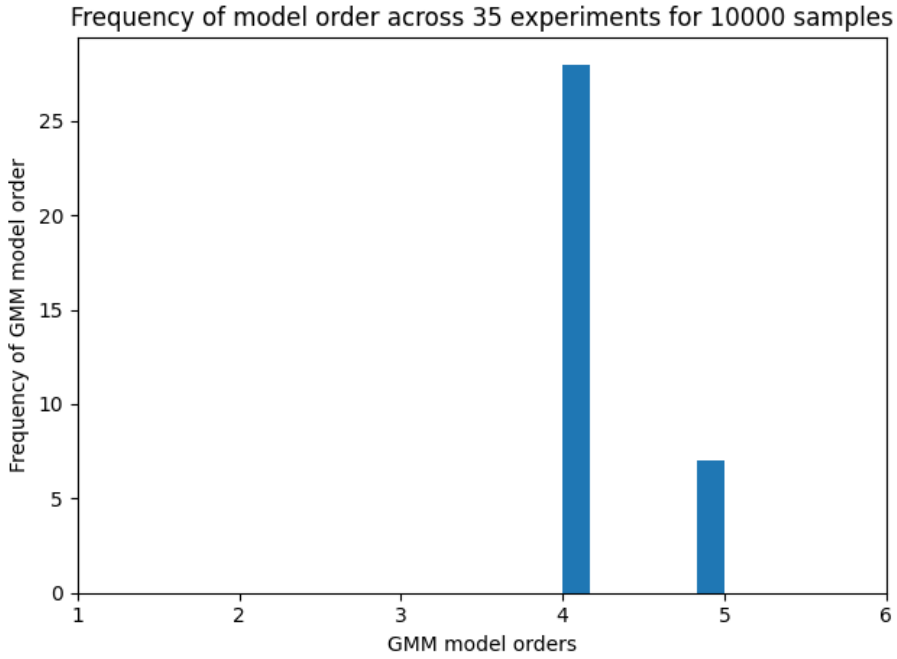
$$\boldsymbol{\theta}^{(t+1)} = \arg\max_{\boldsymbol{\theta}} Q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{(t)})$$

This log likelihood function was used to measure the model parameter estimation performance for each of the above mentioned datasets. These experiments were repeated **35** times and the frequency at which each of these six GMM orders gets selected for each of the datasets were calculated and plotted as shown below




Frequency of model order across 35 experiments for 100 samples
Frequency of GMM model order
GMM model orders
0
5
10
15
20
25
30
1
2
3
4
5
6


Frequency of model order across 35 experiments for 1000 samples
Frequency of GMM model order
GMM model orders
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
1
2
3
4
5
6



The rate at which each of the model orders gets selected for each dataset is provided below. These values are calculated by dividing the frequency with the number of experiments.

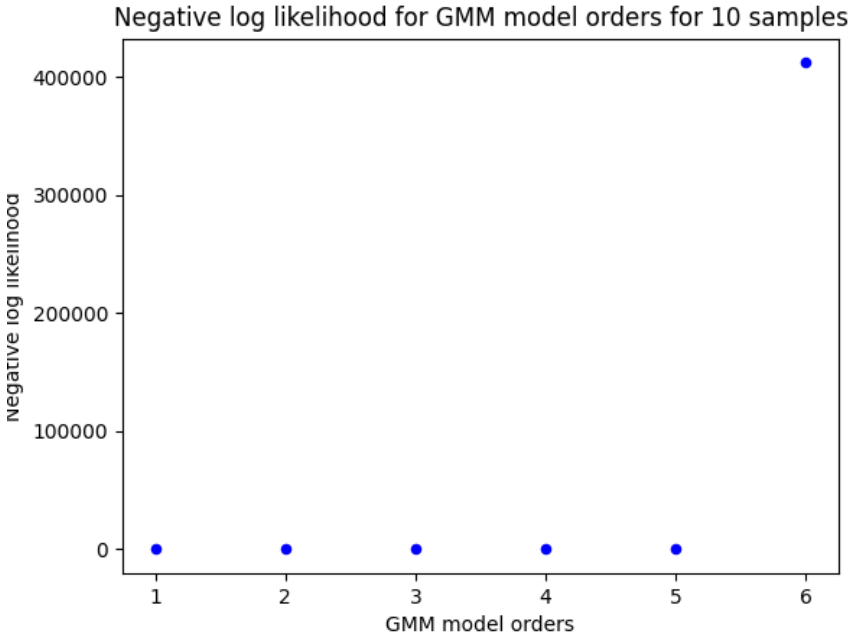
Dataset	1	2	3	4	5	6
T10	0.91	0.9	0.0	0.0	0.0	0.0
T100	0.0	0.0	0.0	0.94	0.06	0.0
T1k	0.0	0.0	0.0	0.54	0.34	0.11
T10k	0.0	0.0	0.0	0.85	0.15	0.0

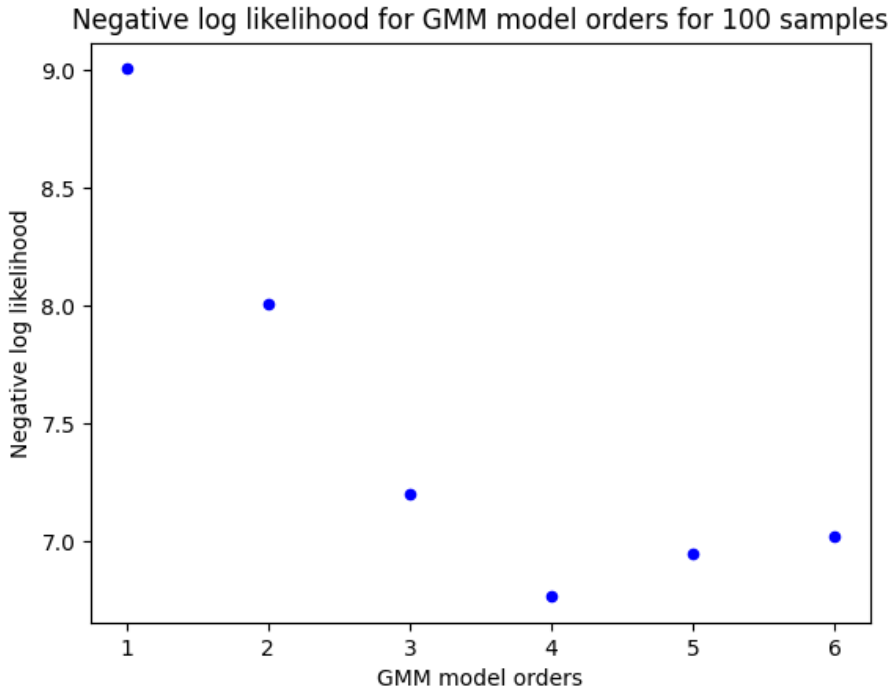
Observation:

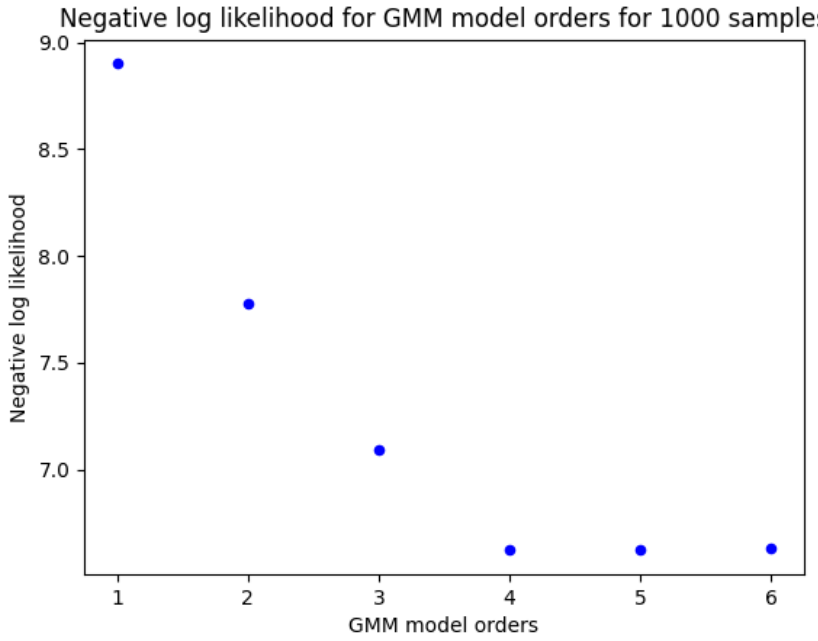
1. In 10 training samples, as there are not enough samples to fit for higher gaussian components, all these samples were grouped into a single gaussian distribution. As a result, the number of Gaussian components equal to 1 gets selected the highest compared to the other model orders.
2. With the increase in the number of samples, the number of Gaussian components equal to 4 fits the data distribution better and as a result gets selected the highest compared to the other model orders.
3. If the gaussian distribution is closer to each other, the chances are higher that an extra gaussian component gets fit at the intersection combining some samples from one distribution and some samples from the other. As a result, higher model orders(5 and 6) also fit well with an increase in number of data samples.

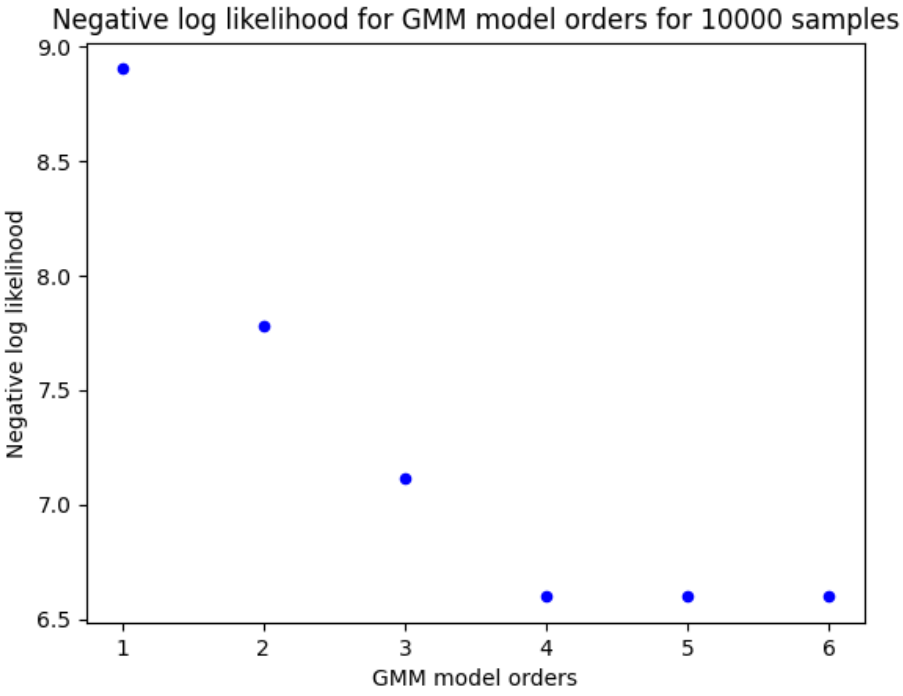
4. As expected, from the below plots, we can observe that the difference in negative log-likelihood between model order 4 and model orders 5 and 6 reduces as we increase the number of data samples, resulting in an increase in the frequency of model orders 5 and 6.
5. For higher samples, the negative log-likelihood reduces as we increase the model order. This behaviour is seen from model order 1 to model order 4.

The plots containing negative log-likelihood for each of the above mentioned datasets across all these 6 GMM orders are shown below




Negative log likelihood for GMM model orders for 100 samples
Negative log likelihood
GMM model orders
9.0
8.5
8.0
7.5
7.0
1
2
3
4
5
6


Negative log likelihood for GMM model orders for 1000 samples
Negative log likelihood
GMM model orders
9.0
8.5
8.0
7.5
7.0
1
2
3
4
5
6



Implementation and Tools used:

Parameter estimation for each Gaussian component was performed using the built-in function sklearn.mixture.GaussianMixture in python. The score provided by the sklearn.model_selection.cross_val_score is the log likelihood of the data. This score was used to measure the performance of the Gaussian mixture model fitting.

Code:

Question 1

```
import numpy as np
import scipy.stats
import random
import matplotlib.pyplot as plt
import sys
import keras
from keras.models import Sequential
from keras.layers import Dense
np.set_printoptions(suppress=True)

def set_mean_cov():

    mval = 7.5
    m0 = np.array([1.3, 0, 0])*mval
    m1 = np.array([1, 0, 1.2])*mval
```

```
    m2 = np.array([0, 1, 1.2])*mval
    m3 = np.array([1.1, 1.2, 1])*mval
    means = [m0, m1, m2, m3]

    C0 = np.array([[10, 0, 0], [0, 40, 0], [0, 0, 15]])
    C1 = np.array([[20, 0, 0], [0, 5, 0], [0, 0, 10]])
    C2 = np.array([[20, 0, 0], [0, 10, 0], [0, 0, 40]])
    C3 = np.array([[5, 0, 0], [0, 20, 0], [0, 0, 5]])
    covs = [C0, C1, C2, C3]

    return means, covs

def gen_class_samples(num_samples, label_ids):

    ##equal prior
    num_labels = len(label_ids)
    class_dist = np.random.randint(num_labels, size=num_samples)
    class_samples = [np.sum(class_dist==label_id).astype('int') for label_id in label_ids]

    return class_samples

def generate_data_pxgl(priors, means, covs, num_samples, label_ids):

    class_samples = gen_class_samples(num_samples, label_ids)
    print('class_samples: ',class_samples, ' sum ', sum(class_samples))

    # generate class data
    pxgls = np.array([], dtype=float).reshape(3,0)
    labels = []
    for label_id in label_ids:
        num_cls_samples = class_samples[label_id]
        mean = means[label_id]
        cov = covs[label_id]
        pxgl = np.random.multivariate_normal(mean, cov, num_cls_samples).T
        pxgls = np.concatenate((pxgls, pxgl), axis=1)
        class_label = [label_id]*num_cls_samples
        labels += class_label

    labels = np.array(labels).reshape((1, -1))
    data = np.concatenate((pxgls, labels), axis=0)

    return data, class_samples

def generate_data_pxgl_samples(samples_type, priors, means, covs, label_ids):

    for i, key in enumerate(samples_type.keys()):

        sample_type = samples_type[key]
        num_samples = int(sample_type[0][0])
```

```
        data_wt_labels, cls_samples = generate_data_pxgl(priors, means, covs, num_samples,
label_ids)
        sample_type[1] = cls_samples
        sample_type[2] = data_wt_labels

        label_names = ["True label distribution for " + str(num_samples) + " for four classes", "x",
"y", "z"]
        plot_dist(data_wt_labels, label_names, label_ids)

    return samples_type

def plot_dist(data, label_names, label_ids):

    tname, xname, yname, zname = label_names

    print('***** plot *****')
    samples = split_data(data, label_ids)

    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')

    colors = ['red', 'blue', 'green', 'brown']
    for label_id, sample in enumerate(samples):
        ax.scatter(sample[0, :], sample[1, :], sample[2, :], s=5, color = colors[label_id], label =
'class ' + str(label_id), marker='*')

    ax.set_title(tname)
    ax.set_xlabel(xname)
    ax.set_ylabel(yname)
    ax.set_zlabel(zname)
    plt.legend()
    plt.show()

def split_data(data_wt_labels, label_ids):

    samples = []

    for label_id in label_ids:
        class_ids = np.where(data_wt_labels[-1,:]==label_id)[0]
        cls_samples = data_wt_labels[:,class_ids]
        samples.append(cls_samples)

    return samples

def calc_theoretical_classifier(sample_type):

    num_samples = sample_type[0][0]
    cls_samples = sample_type[1]
    data_wt_labels = sample_type[2]
```

```
    data = data_wt_labels[:3,:].T #(N, 3)
    labels = data_wt_labels[3,:]

    eval_pxgls = np.zeros((num_labels, num_samples), dtype=float) ##(4, N)
    for label_id in label_ids:
        eval_pxgl = scipy.stats.multivariate_normal.pdf(data, mean=means[label_id],
cov=covs[label_id])
        eval_pxgls[label_id] = eval_pxgl

    priors_np = np.array(priors)
    px = np.matmul(priors_np.reshape(1,-1), eval_pxgls)  ##(1, N)

    stack_px = np.zeros((num_labels, num_samples), dtype=float)
    for label_id in label_ids:
        stack_px[label_id] = px

    plgx = priors_np.reshape(-1, 1)*eval_pxgls/stack_px ## class posterior(4, N)
    risks = np.matmul(loss_mat, plgx)

    decisions = np.argmin(risks, axis=0)

    correct_ids, incorrect_ids = [], []
    for label_id in label_ids:
        label_pids = (labels == label_id)
        correct_cls_bool = ((label_pids)*(decisions == label_id)).astype('int')
        incorrect_cls_bool = ((label_pids)*(decisions != label_id)).astype('int')
        correct_class_ids = np.where(correct_cls_bool == 1)[0]
        incorrect_class_ids = np.where(incorrect_cls_bool == 1)[0]
        correct_ids.append(correct_class_ids)
        incorrect_ids.append(incorrect_class_ids)

    prob_error = 1.0*np.sum((decisions != labels).astype('int'))/num_samples
    prob_error = np.round(prob_error, 4)
    print('prob_error: ',prob_error)

    plot_decision(data_wt_labels, correct_ids, incorrect_ids)

def plot_decision(data, correct_ids, incorrect_ids):

    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')

    markers = ['o', 'v', 's', 'P']
    for label_id in label_ids:
        ax.scatter(data[0, correct_ids[label_id]], data[1, correct_ids[label_id]], data[2,
correct_ids[label_id]], s=5, color = 'green', label = 'correct class' + str(label_id),
marker=markers[label_id])
        ax.scatter(data[0, incorrect_ids[label_id]], data[1, incorrect_ids[label_id]], data[2,
incorrect_ids[label_id]], s=5, color = 'red', label = 'incorrect class' + str(label_id),
marker=markers[label_id])
```

```
    ax.set_title('MAP classification Results')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    plt.legend()
    plt.show()

def get_model(first_num_nodes):

    model = Sequential()

    # first layer
    fc1_act = Dense(units = first_num_nodes, kernel_initializer = 'random_uniform', activation =
'elu')
    model.add(fc1_act)

    # Second layer
    fc2_act = Dense(units = num_labels, kernel_initializer = 'random_uniform', activation =
'softmax')
    model.add(fc2_act)

    model.compile(optimizer='SGD', loss='sparse_categorical_crossentropy', metrics =
['accuracy'])

    return model

def calc_pe(label, prediction):

    num_samples = label.shape[0]
    acc = np.sum((label == prediction).astype('int'))/num_samples
    error = 1 - acc

    return error

def MOS(sample_type, kfold, num_perc_lst):

    num_samples = sample_type[0][0]
    cls_samples = sample_type[1]
    data_wt_labels = sample_type[2]
    data_wt_labels = data_wt_labels[:, np.random.permutation(data_wt_labels.shape[1])]
#shuffle

    data = data_wt_labels[:3,:].T #(N, 3)
    labels = data_wt_labels[3,:].T

    data = data.reshape((kfold, -1, 3))
    labels = labels.reshape((kfold, -1))

    num_val = num_samples/kfold
```

```
    num_train = num_samples - num_val

    perc_lst = []
    for num_perc in num_perc_lst:

        err_lst = []
        for val_idx in range(kfold):

            # train
            train_data = np.concatenate((data[0:val_idx], data[val_idx+1:]), axis=0)
            train_labels = np.concatenate((labels[0:val_idx], labels[val_idx+1:]), axis=0)

            # val
            val_data = data[val_idx].reshape((1, -1, 3))
            val_labels = labels[val_idx]

            #data shape summary
            # print('train data shape ',train_data.shape)
            # print('train label shape ',train_labels.shape)
            # print('val data shape ',val_data.shape)
            # print('val labels shape ',val_labels.shape)

            # get model
            model = get_model(num_perc)

            # train
            model.fit(train_data, train_labels, batch_size = 10, epochs = 100, verbose=0)

            # validate
            val_pred = model.predict(val_data)
            val_pred = np.argmax(val_pred, axis=2)
            val_pred = np.squeeze(val_pred, axis=0)

            err = calc_pe(val_labels, val_pred)
            #print('num_samples:', num_samples,' num_perc: ',num_perc,' val idx: ', val_idx, '
error: ', np.round(err, 4))
            err_lst.append(err)

        mean_err = np.mean(np.array(err_lst))
        std_err = np.std(np.array(err_lst))
        print('num_samples:', num_samples, ' num_perc: ',num_perc,' mean error: ',
np.round(mean_err, 4), ' std error: ',np.round(std_err, 4))
        perc_lst.append(mean_err)

    perc_lst = np.array(perc_lst)
    print('pe for each perceptron: ', perc_lst)
    desired_num_perc = num_perc_lst[np.argmin(perc_lst)]

    return desired_num_perc
```

```
def train_kfoldMLP(train_sample_type, val_sample_type, kfold, num_perc_lst):

    num_samples = train_sample_type[0][0]
    cls_samples = train_sample_type[1]
    data_wt_labels = train_sample_type[2]

    data_wt_labels = data_wt_labels[:, np.random.permutation(data_wt_labels.shape[1])]
#shuffle
    data = data_wt_labels[:3,:].T #(N, 3)
    labels = data_wt_labels[3,:].T

    # Model Order Selection
    desired_num_perc = MOS(train_sample_type, kfold, num_perc_lst)

    for num in range(num_train):

        print('Train ',num_samples)

        # get model
        model = get_model(desired_num_perc)

        # train
        model.fit(data, labels, batch_size = 10, epochs = 100, verbose=1)

        print('model summary')
        print(model.summary())

        # validate
        val_err = validate(val_sample_type, model)

        print('num_samples: ',num_samples,' desired_num_perc: ',desired_num_perc,' val_err: ',
val_err)

def validate(sample_type, model):

    num_samples = sample_type[0][0]
    cls_samples = sample_type[1]
    data_wt_labels = sample_type[2]
    data = data_wt_labels[:3,:].T #(N, 3)
    labels = data_wt_labels[3,:].T

    prediction = model.predict(data, workers=4, use_multiprocessing = True)

    prediction = np.argmax(prediction, axis=1)
    err = calc_pe(labels, prediction)

    return err

if __name__ == "__main__":
```

```
    dim = 3
    label_ids = [0, 1, 2, 3]
    num_labels = len(label_ids)
    priors = [0.25, 0.25, 0.25, 0.25]
    loss_mat = np.ones((num_labels, num_labels)) - np.eye(num_labels)
    kfold = 10
    num_perc_lst = [1, 2, 4, 8, 16, 25, 35, 50]
    num_train = 10

    samples_type = {
        'D100': [[100], [], []],
        'D200': [[200], [], []],
        'D500': [[500], [], []],
        'D1k': [[1000], [], []],
        'D2k': [[2000], [], []],
        'D5k': [[5000], [], []],
        'D100k': [[100000], [], []],
    }

    means, covs = set_mean_cov()
    print('mean: ',means)
    print('covs: ',covs)

    # generate data
    generate_data_pxgl_samples(samples_type, priors, means, covs, label_ids)

    ##theoretical classifier
    calc_theoretical_classifier(samples_type['D100k'])

    # train MLP
    for i, key in enumerate(list(samples_type.keys())[:-1]):
        train_kfoldMLP(samples_type[key], samples_type['D100k'], kfold, num_perc_lst)
```

Question 2

```
import numpy as np
import scipy.stats
import random
import matplotlib.pyplot as plt
import sys
from sklearn.mixture import GaussianMixture
from sklearn.model_selection import cross_val_score
np.set_printoptions(suppress=True)

def set_mean_cov():

    mval = 35
    m0 = np.array([1, 0])*mval
```

```
    m1 = np.array([1, 1.5])*mval
    m2 = np.array([0, 0])*mval
    m3 = np.array([0, 1.5])*mval

    C0 = np.array([[10, 20], [0, 40]])
    C1 = np.array([[20, 0], [40, 5]])
    C2 = np.array([[20, 30], [0, 10]])
    C3 = np.array([[5, 0], [10, 20]])

    return [m0, m1, m2, m3], [C0, C1, C2, C3]

def gen_class_samples(num_samples, priors, label_ids):

    num_labels = len(label_ids)
    class_samples = np.array([0]*num_labels)

    for num_sample in range(num_samples):

        pr = random.uniform(0.0, 1.0)
        if  pr <= priors[0]:
            class_samples[0] += 1
        elif pr <= priors[0] + priors[1]:
            class_samples[1] += 1
        elif pr <= priors[0] + priors[1] + priors[2]:
            class_samples[2] += 1
        else:
            class_samples[3] += 1

    return class_samples

def generate_data_pxgl(priors, means, covs, num_samples, label_ids):

    class_samples = gen_class_samples(num_samples, priors, label_ids)
    print('class_samples: ',class_samples, ' sum ', sum(class_samples))

    # generate class data
    pxgls = np.array([], dtype=float).reshape(2,0)
    labels = []
    for label_id in label_ids:
        num_cls_samples = class_samples[label_id]
        mean = means[label_id]
        cov = covs[label_id]
        pxgl = np.random.multivariate_normal(mean, cov, num_cls_samples).T
        pxgls = np.concatenate((pxgls, pxgl), axis=1)
        class_label = [label_id]*num_cls_samples
        labels += class_label

    labels = np.array(labels).reshape((1, -1))
    data = np.concatenate((pxgls, labels), axis=0)
```

```
    return data, class_samples

def generate_data_pxgl_samples(samples_type, priors, means, covs, label_ids):

    for i, key in enumerate(samples_type.keys()):

        sample_type = samples_type[key]
        num_samples = int(sample_type[0][0])

        data_wt_labels, cls_samples = generate_data_pxgl(priors, means, covs, num_samples,
label_ids)
        sample_type[1] = cls_samples
        sample_type[2] = data_wt_labels

        label_names = ["True label distribution of " + str(num_samples) + " samples using four
components", "x", "y"]
        plot_dist(data_wt_labels, label_names, label_ids)

    return samples_type

def plot_dist(data, label_names, label_ids):

    tname, xname, yname = label_names

    print('***** plot *****')
    samples = split_data(data, label_ids)

    colors = ['red', 'blue', 'green', 'brown']
    for label_id, sample in enumerate(samples):
        plt.scatter(sample[0, :], sample[1, :], s=5, color = colors[label_id], label = 'class ' +
str(label_id), marker='*')

    plt.title(tname)
    plt.xlabel(xname)
    plt.ylabel(yname)
    plt.legend()
    plt.show()

def split_data(data_wt_labels, label_ids):

    samples = []

    for label_id in label_ids:
        class_ids = np.where(data_wt_labels[-1,:]==label_id)[0]
        cls_samples = data_wt_labels[:,class_ids]
        samples.append(cls_samples)

    return samples

def MOS(sample_type, kfold, num_repeat):
```

```
    num_samples = sample_type[0][0]

    num_gmm_freq = np.zeros((num_repeat,), dtype=int)

    for num_time in range(num_repeat):

        print('Iteration: ',num_time)

        data_wt_labels, cls_samples = generate_data_pxgl(priors, means, covs, num_samples,
label_ids)

        data_wt_labels = data_wt_labels[:, np.random.permutation(data_wt_labels.shape[1])]
#shuffle

        data = data_wt_labels[:2,:].T #(N, 2)
        labels = data_wt_labels[2,:].T

        gmm_mean = np.zeros((len(num_gmm_lst),), dtype=float)
        gmm_std = np.zeros((len(num_gmm_lst),), dtype=float)

        for num_gmm in num_gmm_lst:
            GMM = GaussianMixture(num_gmm, covariance_type='full',
                    random_state=0)
            scores = cross_val_score(GMM, data, labels, cv=kfold)
            mean_scores = np.mean(scores)
            std_scores = np.std(scores)

            gmm_mean[num_gmm-1] = mean_scores
            gmm_std[num_gmm-1]  = std_scores
            print('num_samples: ', num_samples, ' num_gmm: ', num_gmm,
            ' mean_scores: ',np.round(mean_scores, 4), ' std_scores: ', np.round(std_scores, 4))

        desired_num_gmm = np.argmin(abs(gmm_mean)) + 1
        print('desired_num_gmm ',desired_num_gmm)
        num_gmm_freq[num_time] = desired_num_gmm

    plot_hist(num_gmm_freq, num_samples, num_time+1)

def plot_hist(num_gmm_freq, num_samples, num_time):

    print('num_gmm_freq: ',num_gmm_freq)
    n_bins = len(num_gmm_lst)
    fig, ax = plt.subplots(tight_layout=True)
    ax.set_xlim([1, 6])
    ax.hist(num_gmm_freq, bins=n_bins)
    plt.title('Frequency of model order across 35 experiments for ' + str(num_samples) + '
samples')
    plt.xlabel('GMM model orders')
    plt.ylabel('Frequency of GMM model order')
```

```
    #plt.show()
    plt.savefig(str(num_samples) + '_' + str(num_time) + '.png')

if __name__ == "__main__":

    dim = 2
    label_ids = [0, 1, 2, 3]
    num_labels = len(label_ids)
    priors = [0.1, 0.2, 0.3, 0.4]
    kfold = 10
    num_repeat = 35
    num_gmm_lst = [1, 2, 3, 4, 5, 6]

    samples_type = {
        'D10': [[10], [], []],
        'D100': [[100], [], []],
        'D1k': [[1000], [], []],
        'D10k': [[10000], [], []],
    }

    means, covs = set_mean_cov()
    print('means ',means)
    print('covs ',covs)

    #generate_data_pxgl_samples(samples_type, priors, means, covs, label_ids)

    #Model order selection
    for i, key in enumerate(list(samples_type.keys())):
        MOS(samples_type[key], kfold, num_repeat)
```

References:

1. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#elu
2. https://stats.stackexchange.com/questions/198038/cross-entropy-or-log-likelihood-in-output-layer
3. https://scikit-learn.org/stable/modules/cross_validation.html
4. https://glassboxmedicine.com/2019/12/07/connections-log-likelihood-cross-entropy-kl-divergence-logistic-regression-and-neural-networks/
5. https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html