NORTHEASTERN UNIVERSITY

Department of Electrical and Computer Engineering

EECE 5644 Machine Learning and Pattern Recognition

Homework Assignment – 3

Mansi Rao Mudrakola
NUID: 002702946

Imports:

```
# Widget to manipulate plots in Jupyter notebooks
%matplotlib widget

import matplotlib.pyplot as plt # For general plotting
import numpy as np
from scipy.stats import norm, multivariate_normal as mvn
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
import torch.nn.functional as F

# Utility to visualize PyTorch network and shapes
from torchsummary import summary

np.set_printoptions(suppress=True)

# Set seed to generate reproducible "pseudo-randomness" (handles scipy's "randomness" too)
np.random.seed(7)

# Customize font sizes for better visualization
plt.rc('font', size=20)          # controls default text sizes
plt.rc('axes', titlesize=16)     # fontsize of the axes title
plt.rc('axes', labelsize=16)     # fontsize of the x and y labels
plt.rc('xtick', labelsize=14)    # fontsize of the tick labels
plt.rc('ytick', labelsize=14)    # fontsize of the tick labels
plt.rc('legend', fontsize=18)    # legend fontsize
plt.rc('figure', titlesize=20)   # fontsize of the figure title
```

Utility Functions:

```
def generate_data(sample, gmmparam):
    n = gmmparam['meanvectors'].shape[1]
    X = np.zeros([sample, n])
    labels = np.zeros(sample)
    u = np.random.rand(sample)
    threshold = np.cumsum(gmmparam['priors'])
    threshold = np.insert(threshold, 0, 0)
    L = np.array(range(len(gmmparam['priors'])))
    for l in L:
        indices = np.argwhere((threshold[l] <= u) & (u <= threshold[l+1]))[:, 0]
        N_labels = len(indices)
        labels[indices] = l * np.ones(N_labels)
        X[indices, :] = mvn.rvs(gmmparam['meanvectors'][l], gmmparam['covariancematrices'][l], N_labels)
    return X, labels
```

Question 1

In this exercise, you will train many multilayer perceptrons (MLP) to approximate the class label posteriors, using maximum likelihood parameter estimation (equivalently, with minimum average cross-entropy loss) to train the MLP. Then, you will use the trained models to approximate a MAP classification rule in an attempt to achieve minimum probability of error (i.e. to minimize expected loss with 0-1 loss assignments to correct-incorrect decisions).

Data Distribution: For C = 4 classes with uniform priors, specify Gaussian class-conditional pdfs for a 3-dimensional real-valued random vector x (pick your own mean vectors and covariance matrices for each class). Try to adjust the parameters of the data distribution so that the MAP classifier that uses the true data pdf achieves between 10% − 20% probability of error.

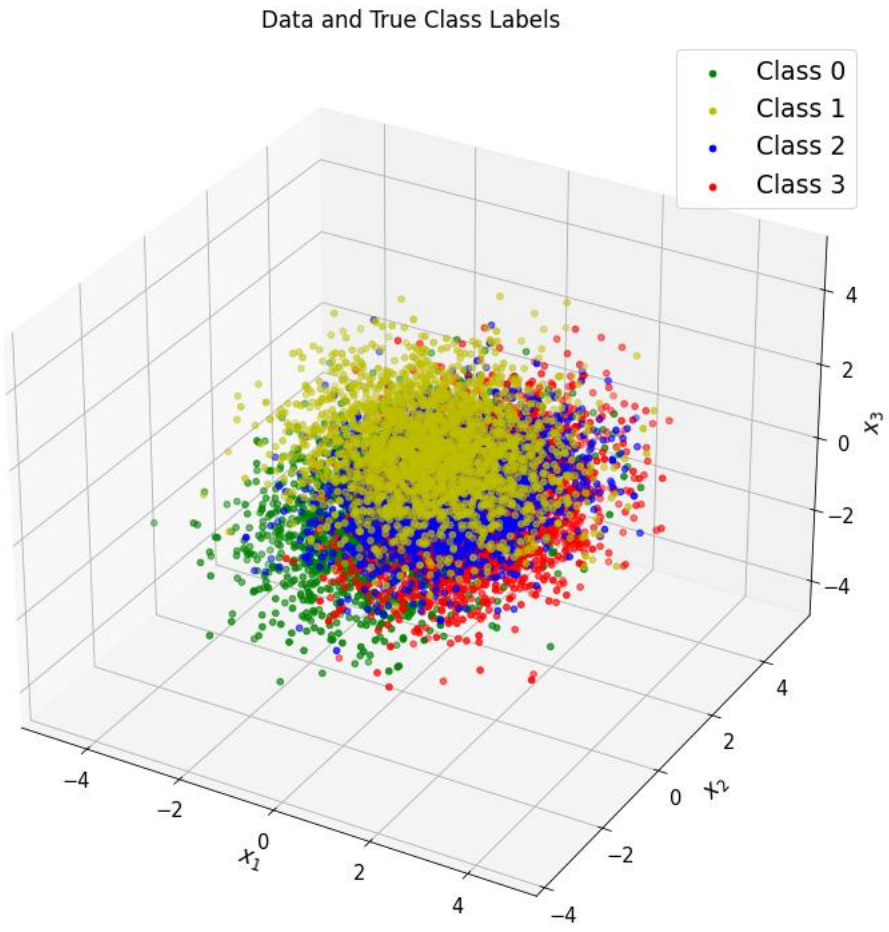
The Gaussian class-conditional PDF's parameters are:

$$\mu_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \mu_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \quad \Sigma_0 = \begin{bmatrix} 1.5 & 0.3 & 0.3 \\ 0.3 & 1 & 0.5 \\ 0.3 & 0.5 & 1 \end{bmatrix}$$

$$\mu_2 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad \mu_3 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \quad \Sigma_2 = \begin{bmatrix} 1 & 0.2 & 0.2 \\ 0.2 & 1 & 0 \\ 0.2 & 0 & 1 \end{bmatrix}$$

$$\Sigma_1 = \begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \Sigma_3 = \begin{bmatrix} 1 & 0 & 0.2 \\ 0 & 1.5 & 0 \\ 0.2 & 0 & 1.5 \end{bmatrix}$$

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

%matplotlib widget



samples = 10000
classes = 4

gmmparam = {
    'priors': np.ones(classes) / classes,
    'meanvectors': np.array([[0, 0, 0],
                [0, 1, 1],
                [1, 0, 1],
                [1, 1, 0]]),
    'covariancematrices': np.array([[[1.5, 0.3, 0.3],
                    [0.3, 1, 0.5],
                    [0.3, 0.5, 1]],

                   [[1.5, 0, 0],
                    [0, 1, 0],
                    [0, 0, 1]],

                   [[1, 0.2, 0.2],
                    [0.2, 1, 0],
                    [0.2, 0, 1]],

                   [[1, 0, 0.2],
                    [0, 1.5, 0],
                    [0.2, 0, 1.5]]])
}

fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111, projection='3d')

X,labels=generate_data(samples, gmmparam)
ax.scatter(X[labels == 0, 0], X[labels == 0, 1], X[labels== 0, 2], c='g', label="Class 0")
ax.scatter(X[labels == 1, 0], X[labels == 1, 1], X[labels == 1, 2], c='y', label="Class 1")
ax.scatter(X[labels == 2, 0], X[labels == 2, 1], X[labels == 2, 2], c='b', label="Class 2")
ax.scatter(X[labels == 3, 0], X[labels == 3, 1], X[labels == 3, 2], c='r', label="Class 3")
ax.set_xlabel(r"$x_1$")
ax.set_ylabel(r"$x_2$")
ax.set_zlabel(r"$x_3$")
plt.title("Data and True Class Labels")
plt.legend()
plt.tight_layout()
plt.show()
```



MLP Structure: Use a 2-layer MLP (one hidden layer of perceptrons) that has P perceptrons in the first (hidden) layer with smooth-ramp style activation functions (e.g., ISRU, Smooth-ReLU, ELU, etc). At the second/output layer use a softmax function to ensure all outputs are positive and add up to 1. The best number of perceptrons for your custom problem will be selected using cross-validation.

```
class TwolayerMLP(nn.Module):
    def __init__(self, n, P, C):
        super(TwolayerMLP, self).__init__()
        self.input_fc = nn.Linear(n, P)
        self.output_fc = nn.Linear(P, C)

    def forward(self, X):
        X = self.input_fc(X)
        X = F.relu(X)
        y = self.output_fc(X)
        return y
```

Generate Data: Using your specified data distribution, generate multiple datasets: Training datasets with 100, 500, 1000, 5000, 10000 samples and a test dataset with 100000 samples. You will use the test dataset only for performance evaluation.

```
train_sample = [100, 500, 1000, 5000,10000]

test_sample = 100000

# Lists to hold the corresponding input matrices, target vectors and sample label counts
X_train = []
Y_train = []

for N_i in train_sample:
    print("Generating the Training data set for sample = {}".format(N_i))

    X_i, Y_i = generate_data(N_i, gmmparam)

    X_train.append(X_i)
    Y_train.append(Y_i)

print("Generating the Test set = {}".format(test_sample))
X_test, Y_test = generate_data(test_sample, gmmparam)
```

```
Generating the Training data set for sample = 100
Generating the Training data set for sample = 500
Generating the Training data set for sample = 1000
Generating the Training data set for sample = 5000
Generating the Training data set for sample = 10000
Generating the Test set = 100000
```

Theoretically Optimal Classifier: Using the knowledge of your true data pdf, construct the minimum-probability-of-error classification rule, apply it on the test dataset, and empirically esti- mate the probability of error for this theoretically optimal classifier. This provides the aspirational performance level for the MLP classfier.

```
class_condition_likelihoods = np.array([mvn.pdf(X_test, gmmparam['meanvectors'][i], gmmparam['covariancematrices'][i]) for i in range(classes)])
decisions = np.argmax(class_condition_likelihoods, axis=0)
wrong_samples = sum(decisions != Y_test)
minimum_probability_error = (wrong_samples / test_sample)
print("Probability of Error on Test Set using the true Data pdf = {}".format(minimum_probability_error))
```

```
Probability of Error on Test Set using the true Data pdf = 0.44829
```

Model Order Selection: For each of the training sets with different number of samples, perform 10-fold cross-validation, using minimum classification error probability as the objective function, to select the best number of perceptrons (that is justified by available training data).

```python
def train_model(model, data, labels, optimizer, criterion=nn.CrossEntropyLoss(), num_epochs=100):
    model.train()
    for epoch in range(num_epochs):
        outputs = model(data)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return model, loss

def predict_model(model, data):
    model.eval()
    with torch.no_grad():
        predicted_labels = model(data)
        predicted_labels = predicted_labels.detach().numpy()
        return np.argmax(predicted_labels, 1)
```

K-fold CV algorithm

1. Partition D into $D_1, D_2, \ldots, D_K$ with equal partition sizes (or close)
2. for model $m \in \{1, \ldots, M\}$
   for fold $k \in \{1, \ldots, K\}$

$$D_{valid-k} = D_k \; ; \; D_{train-k} = D - D_k$$

$$\theta^*_{m,k} = \underset{\theta_m}{\operatorname{argmin}} \mathcal{L}_{train}(\theta_m ; D_{train-k})$$

$$\epsilon_{m,k} = \mathcal{L}_{valid}(\theta^*_{m,k} ; D_{valid-k})$$

$$\epsilon_m = \frac{1}{K} \sum_{k=1}^{K} \epsilon_{m,k}$$

3. best model has smallest average error:

$$m^* = \underset{m \in \{1, \ldots, M\}}{\operatorname{argmin}} \epsilon_m$$

4. Given m^*, train on entire dataset:

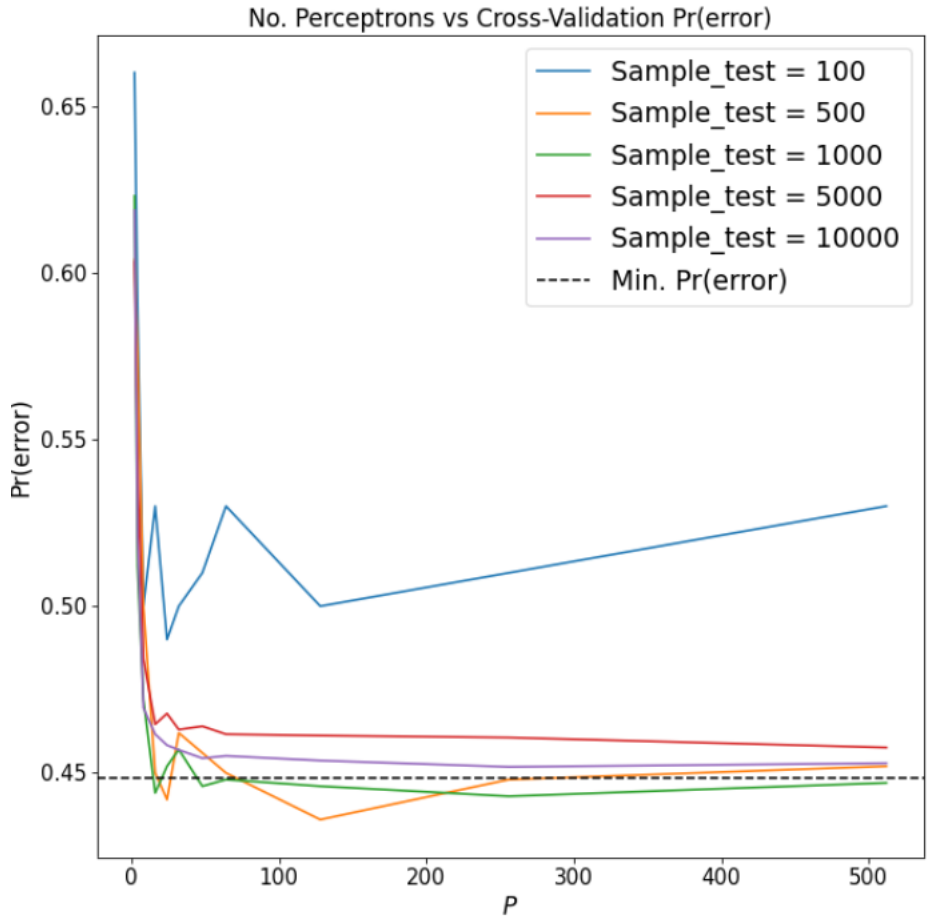
$$\theta^*_m = \underset{\theta^*_m}{\operatorname{argmin}} \mathcal{L}_{train}(\theta_m ; D)$$

```python
def k_fold_cv_perceptrons(K, P_list, data, labels):
    """
    Performs k-fold cross-validation to select the optimal number of perceptrons for a two-layer MLP
    model based on the minimum validation error.

    Args:
    K (int): The number of folds for k-fold cross-validation.
    P_list (list): A list of integers representing the number of perceptrons to test.
    data (numpy.ndarray): The dataset features.
    labels (numpy.ndarray): The dataset labels.

    Returns:
    optimal_P (int): The optimal number of perceptrons.
    error_valid_m (numpy.ndarray): The mean validation error across K folds for each value of P.
    """
    kf = KFold(n_splits=K, shuffle=True)

    error_valid_mk = np.zeros((len(P_list), K))

    for p_idx, p in enumerate(P_list):
        for fold_idx, (train_indices, valid_indices) in enumerate(kf.split(data)):
            X_train, y_train = torch.FloatTensor(data[train_indices]), torch.LongTensor(labels[train_indices])
            X_valid, y_valid = torch.FloatTensor(data[valid_indices]), labels[valid_indices]

            model = TwolayerMLP(X_train.shape[1], p, classes)
            optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
            model, _ = train_model(model, X_train, y_train, optimizer)

            predictions = predict_model(model, X_valid)
            error_valid_mk[p_idx, fold_idx] = np.sum(predictions != y_valid) / len(y_valid)

    error_valid_m = np.mean(error_valid_mk, axis=1)
    optimal_P = P_list[np.argmin(error_valid_m)]

    return optimal_P, error_valid_m
```

Reporting the optimal number of perceptrons P^* per training set:

```python
K = 10
P_list = [2, 4, 8, 16, 24, 32, 48, 64, 128, 256, 512]
best_P_list = []

fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111)
print("\t# of Training Samples \tBest # of Perceptrons \tPr(error)")
for i in range(len(X_train)):
    P_best, P_CV_err = k_fold_cv_perceptrons(K, P_list, X_train[i], Y_train[i])
    best_P_list.append(P_best)
    print("\t\t %d \t\t\t %d \t\t  %.3f" % (train_sample[i], P_best, np.min(P_CV_err)))
    ax.plot(P_list, P_CV_err, label="Sample_test = {}".format(train_sample[i]))

plt.axhline(y=minimum_probability_error, color="black", linestyle="--", label="Min. Pr(error)")
ax.set_title("No. Perceptrons vs Cross-Validation Pr(error)")
ax.set_xlabel(r"$P$")
ax.set_ylabel("Pr(error)")
ax.legend()
plt.show()
```

# of Training Samples	Best # of Perceptrons	Pr(error)
100	24	0.490
500	128	0.436
1000	256	0.443
5000	512	0.458
10000	256	0.452



Since we make no attempt to reduce these effects via regularization, one would likely settle on fewer perceptrons than the best result of the CV method, as this is likely overfitting. P=128 or even less could be an appropriate choice because the data is not very complicated or high-dimensional. Furthermore, the larger datasets (e.g., 2000 or 5000 samples) show a plateau in performance increases around p=128 and are therefore better representations of the genuine data distribution. Therefore, we would probably follow Occam's razor approach and choose the less complex, resource-efficient model, i.e., selecting a model that is lower than the CV result but still provides a decent estimate of the chance of mistake.

Seeing how erratic and noisy the MLP's performance is at low data levels is also intriguing. Given that the trend for P vs. Pr(error) does not ensure smoothness, we would not want to commit to a result in the N=100 or N=200 configurations. In spite of any deceptive performance gains made by N<1000 trained models, such as the top rates associated with N=500, you could choose to only believe the MLP

performance if it was trained on N>=1000 samples. More generally, this misleading impact highlights how dependent neural networks are on the availability of data, as it is soon demonstrated to be false when assessed on the test set in the following phase.

Model Training: For each training set, having identified the best number of perceptrons using cross-validation, using maximum likelihood parameter estimation (minimum cross-entropy loss) train an MLP using each training set with as many perceptrons as you have identified as optimal for that training set. These are your final trained MLP models for class posteriors (possibly each with different number of perceptrons and different weights). Make sure to mitigate the chances of getting stuck at a local optimum by randomly reinitializing each MLP training routine multiple times and getting the highest training-data log-likelihood solution you encounter.

```
trained_models = []
num_restarts = 10
for i in range(len(X_train)):
    print("Training model for N = {}".format(X_train[i].shape[0]))
    X_i = torch.FloatTensor(X_train[i])
    y_i = torch.LongTensor(Y_train[i])
    best_models = []
    best_losses = []
    # Remove chances of falling into suboptimal local minima
    for r in range(num_restarts):
        model = TwolayerMLP(X_i.shape[1], best_P_list[i], classes)
        optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
        # Trained model
        model, loss = train_model(model, X_i, y_i, optimizer)
        best_models.append(model)
        best_losses.append(loss.detach().item())

    # Add best model from multiple restarts to list
    trained_models.append(best_models[np.argmin(best_losses)])
```

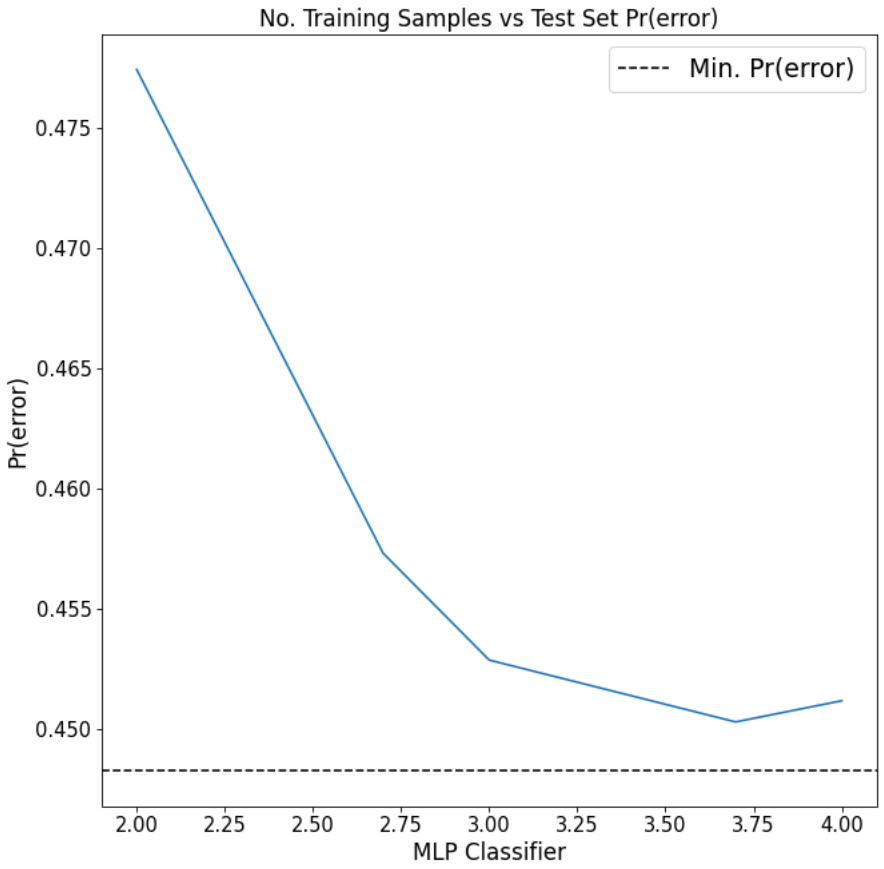
```
Training model for N = 100
Training model for N = 500
Training model for N = 1000
Training model for N = 5000
Training model for N = 10000
```

Performance Assessment: Using each trained MLP as a model for class posteriors, and using the MAP decision rule (aiming to minimize the probability of error) classify the samples in the test set and for each trained MLP empirically estimate the probability of error.

Report Process and Results: Describe your process of developing the solution; numerically and visually report the test set empirical probability of error estimates for the theoretically optimal and multiple trained MLP classifiers. For instance show a plot of the empirically estimated test P(error) for each trained MLP versus number of training samples used in optimizing it (with semilog-x axis), as well as a horizontal line that runs across the plot indicating the empirically estimated test P(error) for the theoretically optimal classifier.

```
# First conver test set data to tensor suitable for PyTorch models
X_test_tensor = torch.FloatTensor(X_test)
pr_error_list = []

fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111)
# Estimate loss (probability of error) for each trained MLP model by testing on the test data set
print("Probability of error results summarized below per trained MLP: \n")
print("\t # of Training Samples \t Pr(error)")
for i in range(len(X_train)):
    # Evaluate the neural network on the test set
    predictions = predict_model(trained_models[i], X_test_tensor)
    # Compute the probability of error estimates
    prob_error = np.sum(predictions != Y_test) / len(Y_test)
    print("\t\t %d \t\t   %.3f" % (train_sample[i], prob_error))
    pr_error_list.append(prob_error)

plt.axhline(y=minimum_probability_error, color="black", linestyle="--", label="Min. Pr(error)")
ax.plot(np.log10(train_sample), pr_error_list)
ax.set_title("No. Training Samples vs Test Set Pr(error)")
ax.set_xlabel("MLP Classifier")
ax.set_ylabel("Pr(error)")


ax.legend()
plt.show()
```

```
Probability of error results summarized below per trained MLP:

         # of Training Samples   Pr(error)
                100               0.477
                500               0.457
                1000              0.453
                5000              0.450
                10000             0.451
```



An anticipated outcome of increasing data supplied to the MLP classifier for model training is an improvement in the chance of mistake. We should expect the MLP classifier to perform closer to the theoretically ideal classifier's estimate of the least probability of error as additional data become available.

Question 2

Conduct the following model order selection exercise using 10-fold cross-validation procedure and report your procedure and results in a comprehensive, convincing, and rigorous fashion:

1. Select a Gaussian Mixture Model as the true probability density function for 2-dimensional real-valued data synthesis. This GMM will have 4 components with different mean vectors, different covariance matrices, and different probability for each Gaussian to be selected as the generator for each sample. Specify the true GMM that generates data in a way that has two of the Gaussian components overlap significantly (e.g. set the distance between mean vectors comparable to the sum of their average covariance matrix eigenvalues).

2. Generate multiple data sets with independent identically distributed samples using this true GMM; these datasets will have respectively 10, 100, 1000 samples.

3. For each data set, using maximum likelihood parameter estimation principle (e.g. with the EM algorithm), within the framework of K-fold (e.g., 10-fold) cross-validation, evaluate GMMs with different model orders; specifically evaluate candidate GMMs with 1, 2, . . . , 10 Gaussian components. Note that both model parameter estimation and validation perfor- mance measures to be used is log-likelihood of data.

4. Repeat the experiment multiple times (e.g., at least 100 times) and report your results, indicating the rate at which each of the the six GMM orders get selected for each of the datasets you produced. Develop a good way to describe and summarize your experiment results in the form of tables/figures.

```python
def generate_data(n_samples):
    data_1 = []
    max_order = max(model_orders)
    for k in range(len(true_gmm_param['weights'])):
        samples = np.random.multivariate_normal(
            true_gmm_param['means'][k],
            true_gmm_param['covariancematrices'][k],
            max(n_samples, max_order)  # Ensure at least max_order samples for each component
        )
        data_1.append(samples)
    data = np.vstack(data_1)
    return data
```

This function generates synthetic data for a Gaussian Mixture Model (GMM). It iterates over each component of the GMM specified by mean, covariance matrix, and weight. For each component, it generates a number of samples proportional to the weight of that component. The samples are drawn from a multivariate normal distribution defined by the mean and covariance matrix of the component. Finally, the generated samples are stacked vertically to form the complete dataset.

```
true_gmm_param = {
    'means': np.array([[0, 0], [2, 2], [0, 4], [4, 0]]),
    'covariancematrices':np.array([[[1, 0], [0, 1]],
                [[1, 0.5], [0.5, 1]],
                [[1, -0.7], [-0.5, 1]],
                [[1, 0], [0, 1]]]),
    'weights': np.array([0.2, 0.3, 0.1, 0.4])
}

n_experiments = 100
n_splits = 10
n_samples = [10, 100, 1000]
model_orders = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

freq = np.zeros((len(n_samples), len(model_orders)))
```

These are the true parameters of the GMM used to generate synthetic data. It consists of means, covariance matrices, and weights for each component.

```
def kfold_validation(freq):
    for exp in range(n_experiments):
        for i, n in enumerate(n_samples):
            # Ensure that n is greater than or equal to the maximum model order
            max_order = max(model_orders)
            if n < max_order:
                raise ValueError(f"Number of samples ({n}) should be greater than or equal to the maximum model order ({max_order}).")

            X = generate_data(n)
            cv_scores = np.zeros((n_splits, len(model_orders)))
            kf = KFold(n_splits=n_splits)
            for j, (train_index, test_index) in enumerate(kf.split(X)):
                X_train, X_test = X[train_index], X[test_index]
                for k, order in enumerate(model_orders):
                    gmm = GaussianMixture(n_components=order, covariance_type='full')
                    gmm.fit(X_train)
                    cv_scores[j, k] = gmm.score(X_test)
            freq[i] = np.bincount(np.argmax(cv_scores, axis=1), minlength=len(model_orders))

        avg_freq = freq / ((exp + 1) * n_splits)
        print(f"Frequencies for experiment = {exp + 1}")
        for i, n in enumerate(n_samples):
            print(f"n_samples={n}")
            for j, order in enumerate(model_orders):
                print(f"  order = {order} : Frequency = {avg_freq[i, j]:.4f}")

    freq /= (n_experiments * n_splits)
    return freq
```

This function performs k-fold cross-validation for different numbers of samples (‘n_samples’) and model orders (‘model_orders’). It uses the ‘generate_data’ function to generate synthetic data, then fits a Gaussian Mixture Model (GMM) with varying model orders using k-fold cross-validation. The frequencies of selecting each model order are stored in the ‘freq’ array.

```
import numpy as np
from sklearn.mixture import GaussianMixture
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")


def plot_bar(freq):
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111)
    width = 0.1
    x = np.arange(len(model_orders))
    for i, n in enumerate(n_samples):
        ax.bar(x+i*width, freq[i], width, label=f"{n} samples")
    ax.set_xticks(x+2*width)
    ax.set_xticklabels(model_orders)
    ax.set_xlabel('Model Order')
    ax.set_ylabel('Frequency')
    ax.legend()
    plt.show()
freqs = kfold_validation(freq)
plot_bar(freqs)
```

This function plots the results of the k-fold cross-validation, showing the frequencies of selecting each model order for different numbers of samples.

Outputs:

```
Frequencies for experiment = 1
n_samples=10
  order = 1 : Frequency = 0.2000
  order = 2 : Frequency = 0.4000
  order = 3 : Frequency = 0.1000
  order = 4 : Frequency = 0.1000
  order = 5 : Frequency = 0.0000
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.1000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.1000
  order = 10 : Frequency = 0.0000
n_samples=100
  order = 1 : Frequency = 0.3000
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.4000
  order = 4 : Frequency = 0.1000
  order = 5 : Frequency = 0.1000
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.1000
  order = 10 : Frequency = 0.0000
n_samples=1000
  order = 1 : Frequency = 0.2000
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.2000
  order = 4 : Frequency = 0.3000
  order = 5 : Frequency = 0.1000
  order = 6 : Frequency = 0.1000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.1000
```

```
Frequencies for experiment = 2
n_samples=10
  order = 1 : Frequency = 0.1000
  order = 2 : Frequency = 0.1500
  order = 3 : Frequency = 0.0500
  order = 4 : Frequency = 0.1000
  order = 5 : Frequency = 0.0500
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0500
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.0000
n_samples=100
  order = 1 : Frequency = 0.1000
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.0500
  order = 4 : Frequency = 0.1500
  order = 5 : Frequency = 0.0500
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.1000
  order = 10 : Frequency = 0.0500
n_samples=1000
  order = 1 : Frequency = 0.1000
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.0500
  order = 4 : Frequency = 0.1500
  order = 5 : Frequency = 0.0000
  order = 6 : Frequency = 0.0500
  order = 7 : Frequency = 0.0500
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0500
  order = 10 : Frequency = 0.0500
```

```
Frequencies for experiment = 3
n_samples=10
  order = 1 : Frequency = 0.1333
  order = 2 : Frequency = 0.0333
  order = 3 : Frequency = 0.1000
  order = 4 : Frequency = 0.0667
  order = 5 : Frequency = 0.0000
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.0000
n_samples=100
  order = 1 : Frequency = 0.0667
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.0667
  order = 4 : Frequency = 0.0667
  order = 5 : Frequency = 0.0333
  order = 6 : Frequency = 0.0333
  order = 7 : Frequency = 0.0333
  order = 8 : Frequency = 0.0333
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.0000
n_samples=1000
  order = 1 : Frequency = 0.0667
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.1000
  order = 4 : Frequency = 0.1667
  order = 5 : Frequency = 0.0000
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.0000
```

```
Frequencies for experiment = 4
n_samples=10
  order = 1 : Frequency = 0.0750
  order = 2 : Frequency = 0.0500
  order = 3 : Frequency = 0.1000
  order = 4 : Frequency = 0.0000
  order = 5 : Frequency = 0.0000
  order = 6 : Frequency = 0.0250
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.0000
n_samples=100
  order = 1 : Frequency = 0.0500
  order = 2 : Frequency = 0.0500
  order = 3 : Frequency = 0.0500
  order = 4 : Frequency = 0.0250
  order = 5 : Frequency = 0.0500
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.0250
n_samples=1000
  order = 1 : Frequency = 0.0500
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.0500
  order = 4 : Frequency = 0.0750
  order = 5 : Frequency = 0.0500
  order = 6 : Frequency = 0.0250
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.0000
```

```
Frequencies for experiment = 5
n_samples=10
  order = 1 : Frequency = 0.0200
  order = 2 : Frequency = 0.0400
  order = 3 : Frequency = 0.0600
  order = 4 : Frequency = 0.0400
  order = 5 : Frequency = 0.0200
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0200
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.0000
n_samples=100
  order = 1 : Frequency = 0.0600
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.0600
  order = 4 : Frequency = 0.0400
  order = 5 : Frequency = 0.0000
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0200
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0200
  order = 10 : Frequency = 0.0000
n_samples=1000
  order = 1 : Frequency = 0.0400
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.0600
  order = 4 : Frequency = 0.0200
  order = 5 : Frequency = 0.0400
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0200
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.0200
```

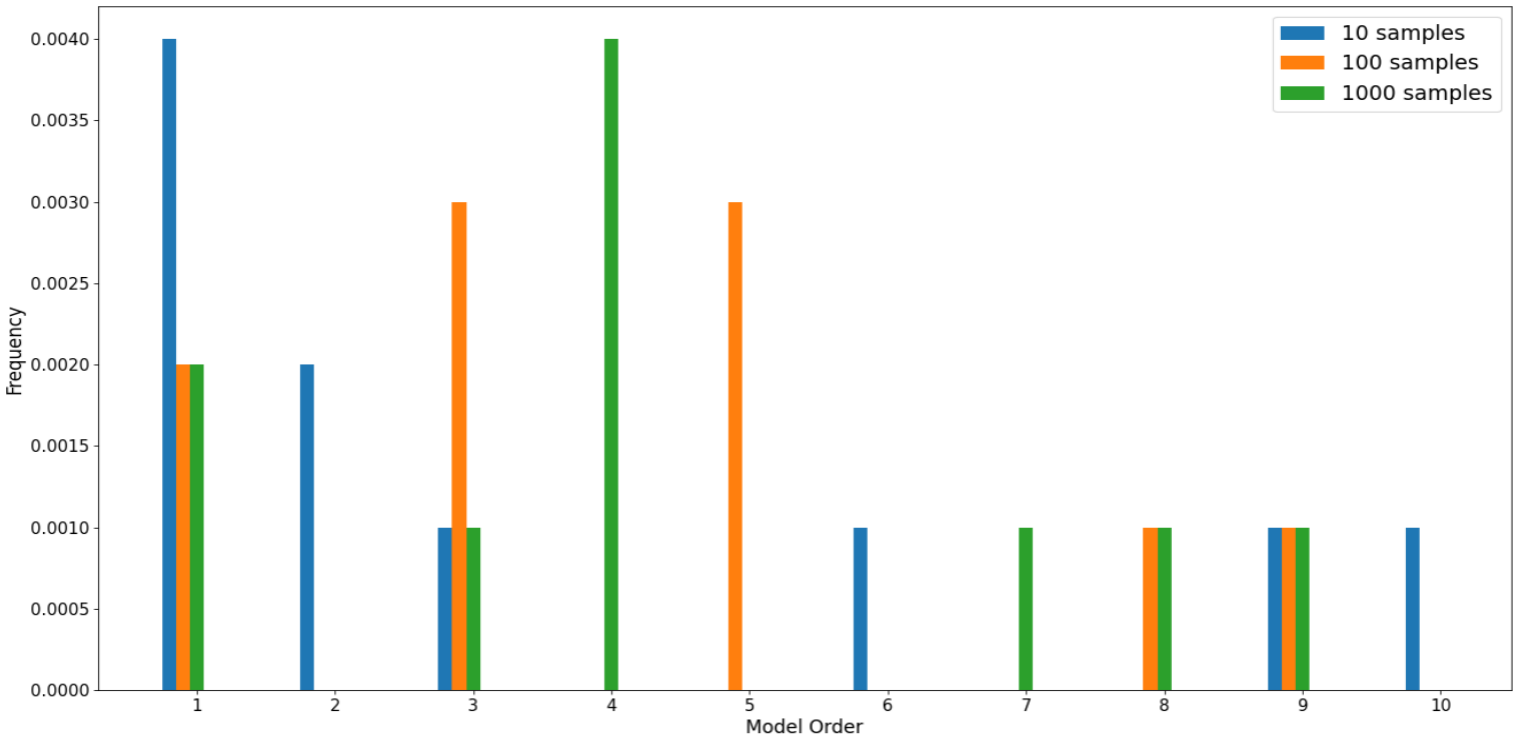
```
Frequencies for experiment = 6
n_samples=10
  order = 1 : Frequency = 0.0667
  order = 2 : Frequency = 0.0500
  order = 3 : Frequency = 0.0000
  order = 4 : Frequency = 0.0333
  order = 5 : Frequency = 0.0167
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.0000
n_samples=100
  order = 1 : Frequency = 0.0333
  order = 2 : Frequency = 0.0167
  order = 3 : Frequency = 0.0333
  order = 4 : Frequency = 0.0167
  order = 5 : Frequency = 0.0333
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0167
  order = 9 : Frequency = 0.0167
  order = 10 : Frequency = 0.0000
n_samples=1000
  order = 1 : Frequency = 0.0333
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.0000
  order = 4 : Frequency = 0.0500
  order = 5 : Frequency = 0.0333
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0333
  order = 10 : Frequency = 0.0167
```

And so on…

The code can be run and the rest outputs for frequency of experiments ranging from 7,8,9,10,……100 can be obtained. Last 2 outputs are as below:

```
Frequencies for experiment = 99
n_samples=10
  order = 1 : Frequency = 0.0030
  order = 2 : Frequency = 0.0010
  order = 3 : Frequency = 0.0040
  order = 4 : Frequency = 0.0020
  order = 5 : Frequency = 0.0000
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0000
  order = 10 : Frequency = 0.0000
n_samples=100
  order = 1 : Frequency = 0.0020
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.0051
  order = 4 : Frequency = 0.0010
  order = 5 : Frequency = 0.0010
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0010
  order = 10 : Frequency = 0.0000
n_samples=1000
  order = 1 : Frequency = 0.0020
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.0010
  order = 4 : Frequency = 0.0040
  order = 5 : Frequency = 0.0010
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0010
  order = 10 : Frequency = 0.0010
```

```
Frequencies for experiment = 100
n_samples=10
  order = 1 : Frequency = 0.0040
  order = 2 : Frequency = 0.0020
  order = 3 : Frequency = 0.0010
  order = 4 : Frequency = 0.0000
  order = 5 : Frequency = 0.0000
  order = 6 : Frequency = 0.0010
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0000
  order = 9 : Frequency = 0.0010
  order = 10 : Frequency = 0.0010
n_samples=100
  order = 1 : Frequency = 0.0020
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.0030
  order = 4 : Frequency = 0.0000
  order = 5 : Frequency = 0.0030
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0000
  order = 8 : Frequency = 0.0010
  order = 9 : Frequency = 0.0010
  order = 10 : Frequency = 0.0000
n_samples=1000
  order = 1 : Frequency = 0.0020
  order = 2 : Frequency = 0.0000
  order = 3 : Frequency = 0.0010
  order = 4 : Frequency = 0.0040
  order = 5 : Frequency = 0.0000
  order = 6 : Frequency = 0.0000
  order = 7 : Frequency = 0.0010
  order = 8 : Frequency = 0.0010
  order = 9 : Frequency = 0.0010
  order = 10 : Frequency = 0.0000
```



The experiment results indicate the frequencies with which different model orders are selected in each experiment and for each number of samples. The output pattern shows the following information for each experiment:

- **Frequencies for experiment = X:** X represents the experiment number.

For each value of `n_samples` (10, 100, 1000), the frequencies for different model orders (1 to 10) are reported. Each line provides the frequencies for a specific `n_samples` value.

- **n_samples = Y**: Y represents the number of samples.

For each `n_samples` value, the frequencies are reported for model orders 1 through 10.

- **order = Z : Frequency = W**: Z represents the model order, and W represents the frequency with which that order was selected.

The frequencies represent the proportion of times each model order was chosen during the cross-validation experiments. These frequencies are calculated over multiple experiments (100 in this case) to provide a more robust assessment of the model selection process. The output pattern allows you to observe how the model order selection varies across different numbers of samples and experiments.

For example, in the first experiment (X=1), for `**n_samples**` = 10, the model order 2 has the highest frequency (0.4), followed by order 1 (0.2), and so on. This pattern is repeated for each experiment, providing insights into the stability and variability of the model order selection process under different conditions.