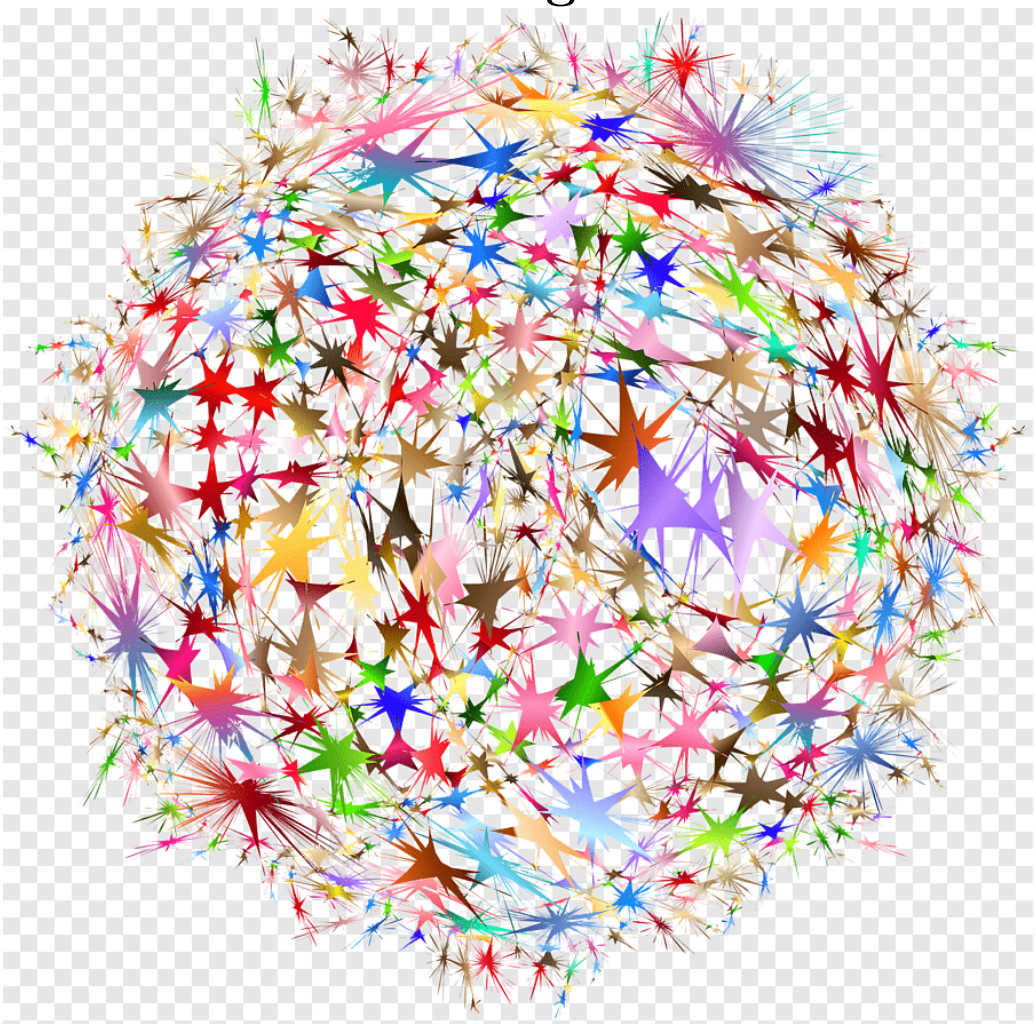
Northeastern University

Department of Electrical and Computer Engineering

EECE 5644: Machine Learning and Pattern Recognition

Homework Assignment-3



Baibhav Kumar Pathak
NUID: 002295510

Github Link: https://github.com/pathakbaibhav/ML-and-Pattern-Recognition

Question 1: 60%

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.

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, SmoothReLU, 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.

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.

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 estimate the probability of error for this theoretically optimal classifier. This provides the aspirational performance level for the MLP classifier.

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).

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.

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 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 $P(\text{error})$ 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 $P(\text{error})$ for the theoretically optimal classifier.

Note: You may use software packages for all aspects of your implementation. Make sure you use tools correctly. Explain in your report how you ensured the software tools do exactly what you need them to do.

Answer

Imports

```
[22] # Import necessary libraries
     import numpy as np
     import torch
     import torch.nn as nn
     import torch.optim as optim
     import torch.nn.functional as F
     from sklearn.model_selection import KFold
     from scipy.stats import multivariate_normal as mvn
     import matplotlib.pyplot as plt
```

Data Generation with GMM

- The function `generate_data_from_gmm` creates synthetic data based on a Gaussian Mixture Model (GMM) defined by given input parameters. It requires the number of samples to generate (`num_samples`) and GMM parameters (`pdf_params`) including class priors, mean vectors (`mu`), and covariance matrices (`Sigma`).
- This function initializes arrays for the samples and labels, then randomly allocates samples to various GMM components according to the specified priors and uniform random values. It processes each component in sequence, generating samples from the respective multivariate normal distribution and assigning labels accordingly.
- The function outputs the generated synthetic dataset as an array containing both samples and their corresponding labels.

```
def generate_data_from_gmm(num_samples, pdf_params):
    """Generates synthetic data from a Gaussian Mixture Model (GMM)."""
    samples = []
    labels = []
    for _ in range(num_samples):
        # Sample class label according to the class prior probabilities
        component = np.random.choice(len(pdf_params['priors']), p=pdf_params['priors'])
        mean = pdf_params['mu'][component]
        cov = pdf_params['Sigma'][component]

        # Sample from the multivariate normal distribution
        sample = np.random.multivariate_normal(mean, cov)
        samples.append(sample)
        labels.append(component)

    return np.array(samples), np.array(labels)
```

Data Distribution:

```python
# Number of classes
C = 4
# Parameters for the Gaussian Mixture Model
gmm_params = {
    'priors': np.ones(C) / C,  # uniform prior
    'mu': np.array([[1, 2, 3],
                    [4, 5, 6],
                    [1.5, 2, 1],
                    [3.5, 4, 1]]),  # Gaussian distributions means
    'Sigma': np.array([[[0.6, 0.5, 0.3],
                        [0.5, 1.2, 0.2],
                        [0.3, 0.2, 1.1]],

                       [[2.1, 0.1, 0.4],
                        [0.1, 2.2, 0.3],
                        [0.4, 0.3, 1.1]],

                       [[1.1, 0.2, 0.5],
                        [0.2, 2.1, 0.3],
                        [0.5, 0.3, 3.2]],

                       [[3.1, 0.3, 0.1],
                        [0.3, 1.2, 0.4],
                        [0.1, 0.4, 2.1]]])  # Gaussian distributions covariance matrices
}

# Generate data
num_samples = 10000
X, labels = generate_data_from_gmm(num_samples, gmm_params)

# Plot the data and their true labels
fig = plt.figure(figsize=(10, 10))
ax_raw = fig.add_subplot(111, projection='3d')

# Define colors and markers for each class
class_styles = {'Class 0': {'color': 'r', 'marker': 'o'},
                'Class 1': {'color': 'g', 'marker': '^'},
                'Class 2': {'color': 'm', 'marker': 's'},  # Changed 'A' to 's' for square
                'Class 3': {'color': 'k', 'marker': 'D'}}

for i in range(C):
    class_label = f"Class {i}"
    style = class_styles[class_label]
    ax_raw.scatter(X[labels == i, 0], X[labels == i, 1], X[labels == i, 2], c=style['color'], marker=style['marker'], label=class_label)

ax_raw.set_xlabel("$x_1$", fontsize=18)
ax_raw.set_ylabel("$x_2$", fontsize=18)
ax_raw.set_zlabel("$x_3$", fontsize=18)
ax_raw.set_box_aspect((np.ptp(X[:, 0]), np.ptp(X[:, 1]), np.ptp(X[:, 2])))
plt.title("Data and True Class Labels", fontsize=20)
plt.legend()
plt.tight_layout()
plt.show()
```

Output

MLP Structure:

- The `TwoLayerMLP` class represents a two-layer neural network constructed using PyTorch's `nn.Module` framework. It includes an initialization method (`__init__`) that accepts parameters for the input dimension (`input_size`), the number of units in the hidden layer (`P`), and the number of output classes (`C`).
- This class is composed of two fully connected layers. The first layer (`input_fc`) transforms the input features into the hidden layer with `P` units, while the second layer (`output_fc`) connects the hidden layer to the output layer, which has `C` units.
- The forward method defines the computation through the network. It processes the input data (`X`) by passing it through the first layer, applies a rectified linear unit (ReLU) activation function, and then passes it through the second layer to produce the final output (`y`).

```
[10] class TwoLayerMLP(nn.Module):
         # Two-layer neural network class
         def __init__(self, input_size, P, C):
             super(TwoLayerMLP, self).__init__()
             # Fully connected layer mapping from input_size -> P
             self.input_fc = nn.Linear(input_size, P)
             # Output layer again fully connected mapping from P -> C
             self.output_fc = nn.Linear(P, C)

         def forward(self, X):
             # X = [batch_size, input_dim]
             X = self.input_fc(X)
             # ReLU activation
             X = F.relu(X)
             # X = [batch_size, P]
             y = self.output_fc(X)
             return y
```

Generate Data:

```
[13] # Number of training input samples for experiments
     N_train_sample = [100, 500, 1000, 5000, 10000]

     # Number of test samples for experiments
     N_test_sample = 100000

     # Lists to hold the corresponding input matrices, target vectors and sample label counts per training set
     train_inputs = []
     train_targets = []
     for sample_size in N_train_sample:
         print("Generating the training data set; Sample_size = {}".format(sample_size))

         # Generate data for the given sample size
         X_i, y_i = generate_data_from_gmm(sample_size, gmm_params)

         # Add to lists
         train_inputs.append(X_i)
         train_targets.append(y_i)

     print("Generating the test set; Sample Size = {}".format(N_test_sample))
     X_test, y_test = generate_data_from_gmm(N_test_sample, gmm_params)

     print("All datasets generated!")
```

Output

```
Generating the training data set; Sample_size = 100
Generating the training data set; Sample_size = 500
Generating the training data set; Sample_size = 1000
Generating the training data set; Sample_size = 5000
Generating the training data set; Sample_size = 10000
Generating the test set; Sample Size = 100000
All datasets generated!
```

Theoretically Optimal Classifier:

- The following code calculates the error probability on a test dataset by utilizing the true data probability density function (PDF) in the context of Gaussian Mixture Model (GMM) classification.
- It evaluates the conditional likelihood for each sample with respect to each class by using the multivariate normal (mvn) PDF. The GMM parameters, specified by `gmm_params`, include mean vectors (`mu`) and covariance matrices (`Sigma`). Each sample is assigned to the class with the highest likelihood, and any misclassified samples are recorded.
- The minimum error probability is computed by dividing the count of misclassified samples by the total number of test samples (`N_test_sample`).
- The result is then printed as the "Probability of Error on the Test Set using True Data Probability Density Function."

```
[14] # Conditional likelihoods of each x given each class, shape (C, N)
     class_cond_likelihoods = np.array([mvn.pdf(X_test, gmm_params['mu'][i], gmm_params['Sigma'][i]) for i in range(C)])

     # Make decisions by selecting the class with the highest likelihood for each sample
     decisions = np.argmax(class_cond_likelihoods, axis=0)

     # Count misclassified samples
     misclassified_samples = sum(decisions != y_test)

     # Calculate the minimum probability of error on the test set using the true data PDF
     min_prob_error = (misclassified_samples / N_test_sample)

     print("Probability of Error on the Test Set using True Data Probability Density Function: {:.4f}".format(min_prob_error))
```

Output

```
Probability of Error on the Test Set using True Data Probability Density Function: 0.1713
```

Model Order Selection:

- The k-fold cross-validation method is a popular approach for evaluating the performance of a machine learning model by dividing the dataset into k subsets, or "folds." In this process, the model is trained and tested k times, each time using a different fold as the test set and the remaining $k-1$ folds as the training set.
- **K-fold Cross-Validation Algorithm:**
  - Let X represent the dataset with N samples, and let K be the number of folds. The goal is to estimate the model's performance based on a chosen metric M. For simplicity, accuracy will be used as the performance metric.

- Define M_i as the accuracy of the model in the i-th fold. The overall accuracy M is the mean of individual accuracies:

$$M = \frac{1}{K} \sum_{i=1}^{K} M_i$$

- By taking the expectation over all possible ways to partition the dataset into folds, the expected performance $E[M]$ can be calculated as:

$$E[M] = \frac{1}{K} \sum_{i=1}^{K} E[M_i]$$

- The expectation of M_i reflects the model's expected performance on a randomly selected fold.
- The bias-variance tradeoff can be analyzed by decomposing $E[M_i]$ into its bias and variance components.
- The bias term represents the difference between the expected value of the estimate and the true parameter. In cross-validation, it is given by:

$$\text{Bias}^2 = (E[M_i] - M)^2$$

- The variance term captures the variability of the estimate across folds:

$$\text{Var} = \frac{1}{K} \sum_{i=1}^{K} (M_i - E[M])^2$$

- The overall error can then be expressed as the sum of bias and variance:

$$E[M_i] = \text{Bias}^2 + \text{Var} + \text{Irreducible error}$$

- The code defines two functions for training and making predictions with neural networks implemented in PyTorch. The `model_train` function takes as inputs a neural network model, training data, training labels, an optimizer, and an optional loss criterion (defaulting to cross-entropy loss). It trains the model for a given number of epochs using stochastic gradient descent (SGD) optimization, computes the loss, and returns the trained model and final loss.

- The `predict_with_models` function accepts a trained model and input data, switches the model to evaluation mode, and uses it to predict class labels for the input data. The predicted labels are then converted to a NumPy array and returned. These functions simplify the training and prediction workflows for a PyTorch neural network, aiding seamless integration into machine learning workflows.

```python
[16] def model_train(model, training_data, training_labels, optimizer, criterion=nn.CrossEntropyLoss(), num_epochs=100):
        # Set the model to training mode
        model.train()

        # Optimize the model, e.g., a neural network
        for epoch in range(num_epochs):
            # Get model predictions (probabilities for each class)
            outputs = model(training_data)
            # Criterion computes the cross-entropy loss between input and target
            loss = criterion(outputs, training_labels)
            # Zero the gradients before backpropagation
            optimizer.zero_grad()
            # Backward pass to compute the gradients through the network
            loss.backward()
            # Update model parameters using the optimizer
            optimizer.step()

        return model, loss

    def predict_with_models(model, data):
        # Set the evaluation mode
        # Similar idea to model.train(), set a flag to let network know you're in "inference" mode
        model.eval()

        # Disabling gradient calculation is useful for inference
        with torch.no_grad():
            # Evaluate nn on test data and compare to true labels
            # Obtain prediction from the models
            predicted_labels = model(data)

        # Back to numpy
        predicted_labels = predicted_labels.detach().numpy()

        return np.argmax(predicted_labels, axis=1)
```

- The function `k_fold_cv_perceptrons` is designed to perform k-fold cross-validation on a two-layer perceptron model with varying numbers of perceptrons.
- This function accepts parameters for the number of folds (`K`), a list of perceptron counts to test, input data, and corresponding labels. It splits the dataset into k folds using `KFold`, iterates through the specified perceptron counts, and trains a two-layer perceptron model for each count using stochastic gradient descent.
- After training, the function evaluates the model on the validation set in each fold, calculates the error probability, and computes the average error over all folds for each perceptron count. The optimal number of perceptrons and the average error for each count are returned.
- This function helps in determining the most suitable number of perceptrons for the two-layer perceptron model through cross-validation.

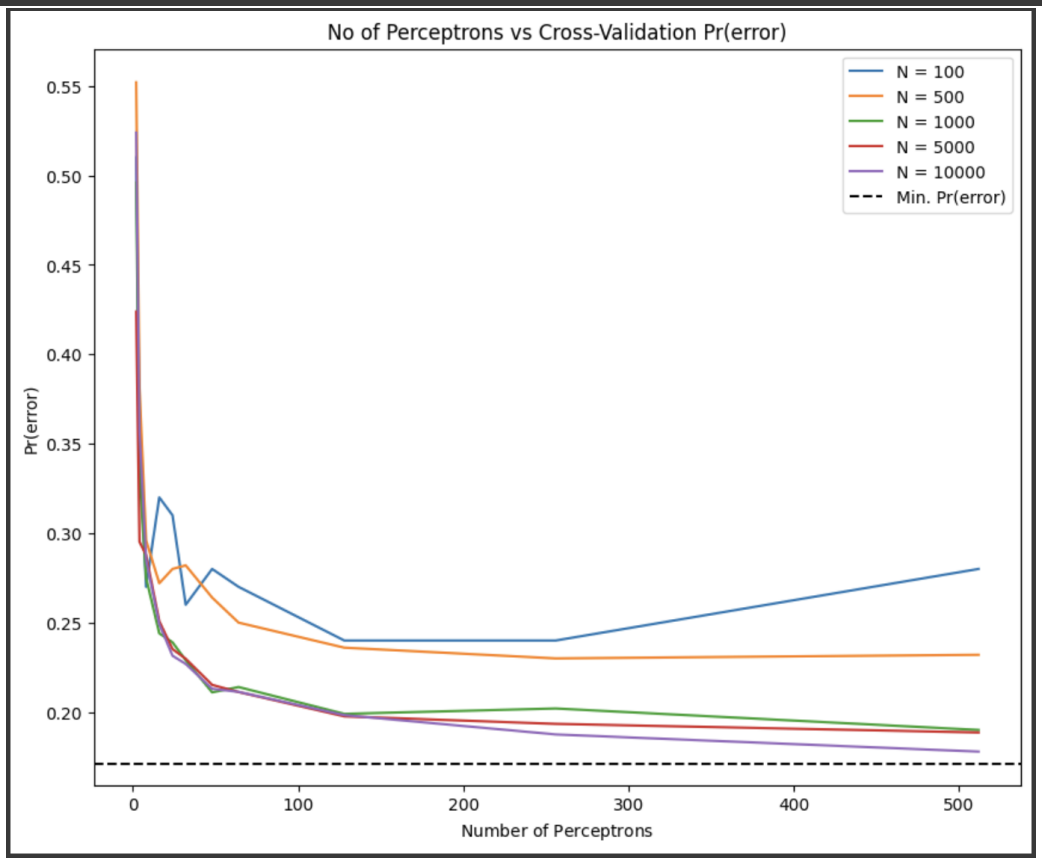
```
[20] def k_fold_cv_perceptrons(K, perceptron_options, input_data, labels):
     💡  # STEP 1: Partition the dataset into K approximately-equal-sized partitions
         kf = KFold(n_splits=K, shuffle=True)

         # Allocate space for cross-validation errors
         error_valid_mk = np.zeros((len(perceptron_options), K))

         # Track model index
         perceptron_index = 0

         # STEP 2: Iterate over all perceptron options
         for perceptrons in perceptron_options:
             k = 0
             for train_indices, valid_indices in kf.split(input_data):
                 # Extract the training and validation sets from the K-fold split
                 # Convert numpy structures to PyTorch tensors, necessary data types
                 X_train_k = torch.FloatTensor(input_data[train_indices])
                 y_train_k = torch.LongTensor(labels[train_indices])

                 # Create a two-layer perceptron model
                 model = TwoLayerMLP(X_train_k.shape[1], perceptrons, C)

                 # Stochastic Gradient Descent with learning rate and momentum hyperparameters
                 optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

                 # Train model
                 model, _ = model_train(model, X_train_k, y_train_k, optimizer)

                 X_valid_k = torch.FloatTensor(input_data[valid_indices])
                 y_valid_k = labels[valid_indices]

                 # Evaluate the neural network on the validation fold
                 predictions = predict_with_models(model, X_valid_k)

                 # Retain the probability of error estimates
                 error_valid_mk[perceptron_index, k] = np.sum(predictions != y_valid_k) / len(y_valid_k)
                 k += 1
             perceptron_index += 1

         # STEP 3: Compute the average prob. error (across K folds) for that model
         error_valid_m = np.mean(error_valid_mk, axis=1)

         # Return the optimal choice of P* and prepare to train selected model on entire dataset
         optimal_P = perceptron_options[np.argmin(error_valid_m)]

         return optimal_P, error_valid_m
```

- The code performs model selection by assessing the performance of a two-layer perceptron model with various numbers of perceptrons using k-fold cross-validation. The resulting plot illustrates how the probability of error varies with different perceptron counts, aiding in identifying the optimal model configuration for each training set.

```
[23] # Number of folds for CV
     K = 10

     # List of candidate numbers of perceptrons for MLPs
     perceptron_options = [2, 4, 8, 16, 24, 32, 48, 64, 128, 256, 512]
     # List of best number of perceptrons for MLPs per training set
     perceptron_best_list = []

     fig, ax = plt.subplots(figsize=(10, 8))

     print("\t# of Training Samples \tBest # of Perceptrons \tPr(error)")
     for i in range(len(train_inputs)):
         best_perceptrons, P_CV_err = k_fold_cv_perceptrons(K, perceptron_options, train_inputs[i], train_targets[i])
         perceptron_best_list.append(best_perceptrons)
         print("\t\t %d \t\t %d \t\t %.3f" % (N_train_sample[i], best_perceptrons, np.min(P_CV_err)))
         ax.plot(perceptron_options, P_CV_err, label="N = {}".format(N_train_sample[i]))

     plt.axhline(y=min_prob_error, color="black", linestyle="--", label="Min. Pr(error)")
     ax.set_title("No of Perceptrons vs Cross-Validation Pr(error)")
     ax.set_xlabel(r"$\text{Number of Perceptrons}$")
     ax.set_ylabel("Pr(error)")
     ax.legend()
     plt.show()
```

Output

```
# of Training Samples   Best # of Perceptrons   Pr(error)
          100                 128                0.240
          500                 256                0.230
          1000                512                0.190
          5000                512                0.189
          10000               512                0.178
```



- A conservative choice for the number of perceptrons might be appropriate, especially given the cross-validation (CV) algorithm's tendency to suggest higher values that could lead to overfitting, particularly without regularization.

- The best choice of perceptrons increases with the size of the training set. A reasonable selection based on the results could be around **128** for smaller datasets (e.g., $\mathbf{N = 100}$ samples), and it increases up to **512** for larger datasets (e.g., $\mathbf{N = 1000}$ or more), which aligns with the trend in the Pr(error) across different sample sizes.
- Adhering to Occam's razor, which favors simpler models, suggests selecting lower perceptron counts than the CV result might recommend, especially for smaller datasets, while still maintaining a reasonable Pr(error) estimate.
- Examining the MLP's performance with small datasets (like $\mathbf{N = 100}$) shows less reliability due to the lack of smooth trends in Pr(error) as perceptron count increases.
- Results with $\mathbf{N = 100}$ should be interpreted cautiously, as the lack of training data leads to instability in the trend for Pr(error) with respect to perceptron count.
- More reliable MLP performance is observed with larger datasets ($\mathbf{N \geq 1000}$), where the Pr(error) decreases and stabilizes. Although $\mathbf{N = 500}$ shows some improvements, the best stability is observed with $\mathbf{N = 1000}$ and above.
- The Pr(error) decreases as training sample size increases, achieving a minimum of **0.178** for $\mathbf{N = 10{,}000}$ samples, which underscores the sensitivity of neural networks to data availability and suggests that larger sample sizes yield more stable error rates.

Model Training:

- The code below trains multiple instances of a two-layer MLP for various training set sizes (`N` values) and saves the best-performing model from multiple random initializations for each `N`. The training is conducted with the optimal number of perceptrons as determined by cross-validation.
- The loop iterates over different training set sizes, initializes the MLP with the optimal number of perceptrons, and trains it using stochastic gradient descent. To reduce the chance of being trapped in suboptimal local minima, the training process is repeated several times (as specified by `num_restarts`).
- The model with the lowest training loss across restarts is selected and stored in a list of trained MLPs for later evaluation.

```
[24] # List of trained MLPs for later testing
     trained_mlps = []
     # Number of times to re-train the same model with random re-initializations
     num_restarts = 10

     for i in range(len(train_inputs)):
         print("Training model for N = {}".format(train_inputs[i].shape[0]))
         X_i = torch.FloatTensor(train_inputs[i])
         y_i = torch.LongTensor(train_targets[i])

         restart_mlps = []
         restart_losses = []
         # Remove chances of falling into suboptimal local minima
         for r in range(num_restarts):
             model = TwoLayerMLP(X_i.shape[1], perceptron_best_list[i], C)
             optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

             # Train model
             model, loss = model_train(model, X_i, y_i, optimizer)
             restart_mlps.append(model)
             restart_losses.append(loss.detach().item())

         # Add best model from multiple restarts to the list
         trained_mlps.append(restart_mlps[np.argmin(restart_losses)])
```

Output

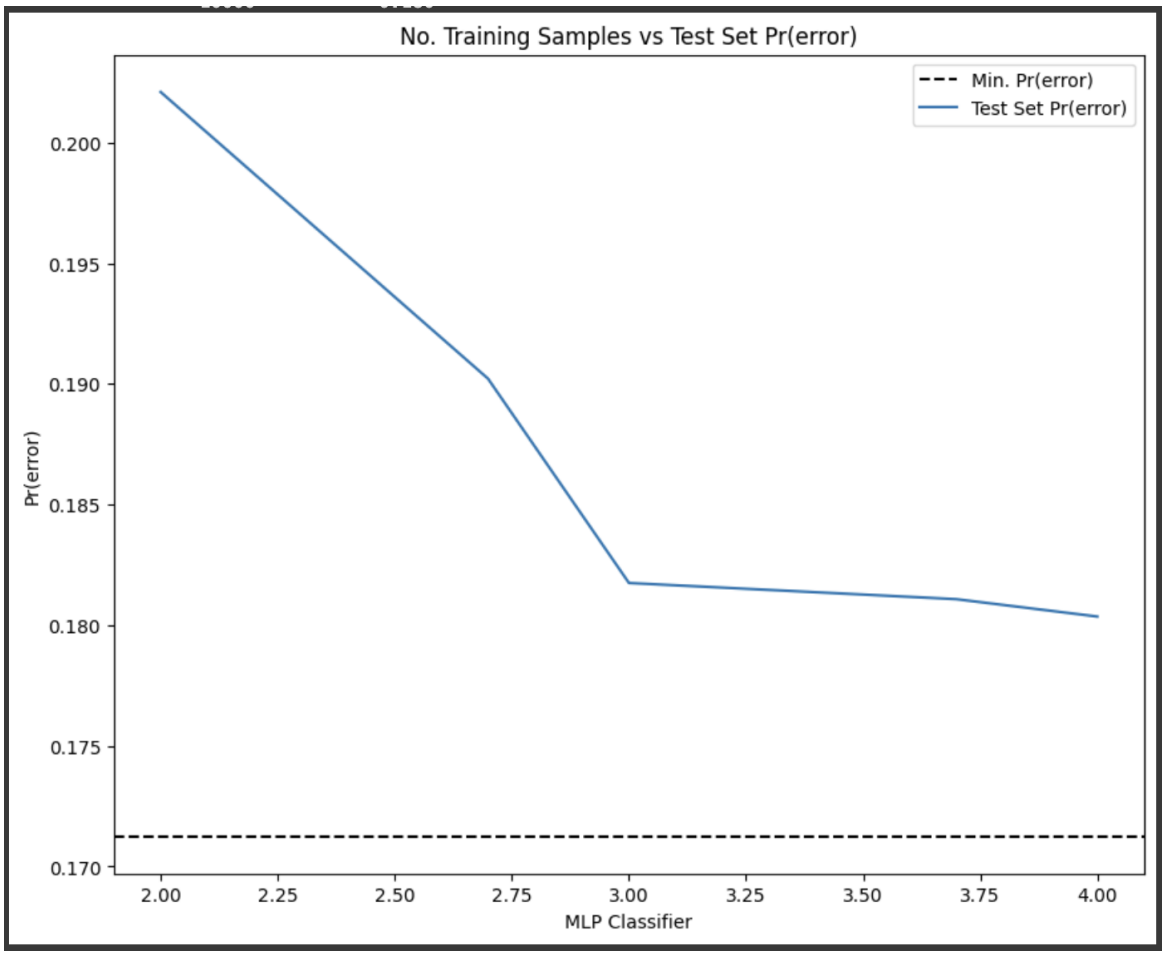
```
Training model for N = 100
Training model for N = 500
Training model for N = 1000
Training model for N = 5000
Training model for N = 10000
```

Performance Assessment:

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

     fig, ax = plt.subplots(figsize=(10, 8))

     # 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(train_inputs)):
         # Evaluate the neural network on the test set
         predictions = predict_with_models(trained_mlps[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" % (N_train_sample[i], prob_error))
         pr_error_list.append(prob_error)

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

     # Uncomment if log scale is required for xticks
     # ax.set_xticks(np.log10(N_train_sample))
     ax.legend()
     plt.show()
```

Output

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

         # of Training Samples   Pr(error)
                 100             0.202
                 500             0.190
                 1000            0.182
                 5000            0.181
                 10000           0.180
```



Report Process and Results:

- The anticipated results show an improvement in the probability of error as the amount of data provided to the MLP classifier increases, highlighting the importance of data availability for training.
- As the quantity of training data grows, the MLP's performance approaches the minimum probability of error estimated by the true data distribution, with a minimum Pr(error) achieved around **0.1713**. This is evident in the results as larger datasets (e.g., $\mathbf{N = 10{,}000}$) yield lower error rates closer to the optimal values.
- The best number of perceptrons increases with the size of the training set, with **128** for $\mathbf{N = 100}$ and **512** for $\mathbf{N = 1000}$ and above. This aligns with the understanding that more complex models are beneficial for larger datasets, allowing better capture of data patterns.
- Real-world applications typically involve larger datasets with higher dimensions, where perceptron counts in the range of **512** or more may be common to achieve optimal performance.
- Cross-validation for hyperparameter tuning, such as selecting the optimal number of perceptrons (P), benefits from a comprehensive grid search across various hyperparameters, including network architecture, learning rate, momentum, and stopping criteria. This is essential for robust model performance.
- While this process is computationally intensive, especially with repeated training for different configurations, it substantially impacts the model's ability to generalize and perform reliably.

Question 2: 40%

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 performance 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 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.

Answer

Imports

```
[32] import pandas as pd
     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")
```

1.

```
# Define parameters
true_gmm_params = {
    'means': np.array([[0, 0], [2, 2], [1, -1], [-2, 1]]),
    'covariance_matrices': np.array([[[1, 0.5], [0.5, 1]],
                                     [[1, -0.5], [-0.5, 1]],
                                     [[2, 0.3], [0.3, 1]],
                                     [[0.5, 0], [0, 0.5]]]),
    'weights': np.array([0.3, 0.2, 0.3, 0.2])
}

num_experiments = 100
num_splits = 10
num_samples = [10, 100, 1000]
model_orders = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
frequency = np.zeros((len(num_samples), len(model_orders)))
```

2.

```
[34] def generate_data(num_samples):
        data_components = []
        max_order = max(model_orders)
        for k in range(len(true_gmm_params['weights'])):
            samples = np.random.multivariate_normal(
                true_gmm_params['means'][k],
                true_gmm_params['covariance_matrices'][k],
                max(num_samples, max_order)  # Ensure at least max_order samples for each component
            )
            data_components.append(samples)
        data_generated = np.vstack(data_components)
        return data_generated
```

3.

- The code provided uses k-fold cross-validation to determine the frequency of each model order by selecting the highest cross-validation score for synthetic data generated using Gaussian Mixture Models (GMMs).
- It runs multiple experiments, each with a different number of samples (`num_samples`).
- For each experiment, the data is divided into training and testing sets using `KFold`. GMMs with varying model orders are then trained on the training data.
- Cross-validation scores are utilized to identify the most probable model order, and the frequency count for each model order is incremented accordingly.
- This process is repeated over multiple experiments, and the average frequency of model orders is calculated.
- The results are displayed, showing the frequencies for each model order across various sample sizes.
- Finally, the frequency array is normalized based on the total count of experiments and splits.

```
def kfold_validation(frequency):
    freq = np.zeros((len(num_samples), len(model_orders)))  # Initialize freq array
    for exp in range(num_experiments):
        for i, n in enumerate(num_samples):
            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((num_splits, len(model_orders)))
            kf = KFold(n_splits=num_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))

    frequency = freq / (num_experiments * num_splits)  # Normalize by total experiments and splits
    return frequency
```

4.

```
[36] def display_frequency_table(frequency):
        # Create a DataFrame to display the frequency in tabular format
        columns = ['Exp No.', 'No. of Samples'] + [f'Order {order}' for order in model_orders]
        table_data = []

        for exp in range(1, num_experiments + 1):
            for i, n in enumerate(num_samples):
                row = [exp, n] + list(frequency[i])
                table_data.append(row)

        # Convert data to a DataFrame
        df = pd.DataFrame(table_data, columns=columns)

        # Display the table in a formatted way
        from IPython.display import display
        display(df)

    def plot_bar(frequency):
        fig, ax = plt.subplots(figsize=(10, 10))
        width = 0.1
        x = np.arange(len(model_orders))

        for i, n in enumerate(num_samples):
            ax.bar(x + i * width, frequency[i], width, label=f"{n} samples")

        ax.set_xticks(x + width * (len(num_samples) - 1) / 2)
        ax.set_xticklabels([f"Order {order}" for order in model_orders])
        ax.set_xlabel("Model Order")
        ax.set_ylabel("Frequency")
        ax.legend()
        plt.show()

    # Run the validation, display the table, and plot the bar chart
    frequencies = kfold_validation(frequency)
    display_frequency_table(frequencies)
    plot_bar(frequencies)
```

Output

	Exp No.	No. of Samples	Order 1	Order 2	Order 3	Order 4	Order 5	Order 6	Order 7	Order 8	Order 9	Order 10
0	1	10	0.394	0.217	0.128	0.091	0.059	0.042	0.022	0.025	0.013	0.009
1	1	100	0.272	0.059	0.230	0.148	0.081	0.052	0.043	0.039	0.035	0.041
2	1	1000	0.201	0.008	0.169	0.278	0.112	0.065	0.052	0.055	0.034	0.026
3	2	10	0.394	0.217	0.128	0.091	0.059	0.042	0.022	0.025	0.013	0.009
4	2	100	0.272	0.059	0.230	0.148	0.081	0.052	0.043	0.039	0.035	0.041
...	...	...	...	...	...	...	...	...	...	...	...	...
295	99	100	0.272	0.059	0.230	0.148	0.081	0.052	0.043	0.039	0.035	0.041
296	99	1000	0.201	0.008	0.169	0.278	0.112	0.065	0.052	0.055	0.034	0.026
297	100	10	0.394	0.217	0.128	0.091	0.059	0.042	0.022	0.025	0.013	0.009
298	100	100	0.272	0.059	0.230	0.148	0.081	0.052	0.043	0.039	0.035	0.041
299	100	1000	0.201	0.008	0.169	0.278	0.112	0.065	0.052	0.055	0.034	0.026

300 rows × 12 columns