INTRODUCTION TO MACHINE LEARNING

AKSHATA KUMBLE

Q1.

We will create a multi-class classification problem with 4 classes and gaussian distributions. Using a theoretically optimal classifier as a benchmark we will train multiple MLPs with a variety of configurations, using cross-entropy loss and maximum likelihood estimation. This allows us to approximate the MAP (Maximum a Posteriori) classification rule.

Process of developing the solution.

1. Define data distribution and generate samples

   1. Define Class Conditional Distribution:

   We have C = 4 classes and for each class we define a 3 dimensional gaussian distribution with a unique mean vector and covariance matrix. Since we want the theoretically optimal classifier to have a probability of error between 10-20%. we set mean vectors and covariance matrices such that the classes overlap slightly, that should lead to some classification ambiguity.

   2. Generating Training and Test sets

   We generate several datasets for training (with various sample sizes) and a large test dataset for performance evaluation. For each class, we generate samples from the gaussian distribution defined for that class. Combine the samples from all classes and assign class labels. Generate training sets with 100, 500, 5000, 10000 samples and test set with 100,000 samples.

   To test if this setup gives us desired error rates for a MAP classifier, we can generate a large dataset and apply a MAP classifier with knowledge of true parameters. The MAP classifier will compute the likelihood of each class & assign the sample to the class with the highest posterior probability.

   With the specified parameters, we should observe an error rate b/w 10-20% depending on the degree of overlap in our gaussian distribution. If the error rate is outside this range we can iteratively adjust the means or covariances by bringing the mean vectors closer or increasing the off diagonal elements in the covariance matrices for added correlation.

   Adjustments:- ① Means: chose positions relatively close to each other to create class overlap. This overlap increases misclassification achieving the desired error rate. ② Covariances: Used varying covariance sizes to control the spread, introducing slight asymmetry with off diag. elements makes certain regions in feature space ambiguous. Expanding the variances will increase the spread of each class making them overlap more. We can increase the diagonal values making the covariance matrices more asymmetrical.

```python
import numpy as np

# Adjusted mean vectors, not too close or too far
class_means = [
    np.array([1.5, 1.5, 1.5]),
    np.array([-1.5, -1.5, -1.5]),
    np.array([1.5, -1.5, 1.5]),
    np.array([-1.5, 1.5, -1.5])
]

# Adjusted covariance matrices with moderate variances
class_covariances = [
    np.array([[1.8, 0.4, 0.2], [0.4, 1.8, 0.3], [0.2, 0.3, 1.8]]),
    np.array([[1.9, 0.3, 0.1], [0.3, 1.9, 0.2], [0.1, 0.2, 1.9]]),
    np.array([[1.8, -0.3, 0.5], [-0.3, 1.7, 0.2], [0.5, 0.2, 1.6]]),
    np.array([[1.7, 0.5, -0.3], [0.5, 1.7, 0.6], [-0.3, 0.6, 1.8]])
]

from scipy.stats import multivariate_normal

# Generate test dataset to evaluate error probability
def generate_data(n_samples, class_means, class_covariances):
    X = []
    y = []
    for i, (mean, cov) in enumerate(zip(class_means, class_covariances)):
        X_class = np.random.multivariate_normal(mean, cov, n_samples // len(class_means))
        X.append(X_class)
        y.append(np.full(X_class.shape[0], i))
    return np.vstack(X), np.hstack(y)

# Create a test dataset of 100,000 samples
test_data, test_labels = generate_data(100000, class_means, class_covariances)


# MAP classifier function using known class pdfs
def theoretical_MAP_classifier(X, class_means, class_covariances):
    posteriors = []
    for mean, cov in zip(class_means, class_covariances):
        # Calculate posterior probability assuming equal priors
        posterior = multivariate_normal(mean=mean, cov=cov).pdf(X)
        posteriors.append(posterior)
    # Assign each sample to the class with the highest posterior probability
    return np.argmax(posteriors, axis=0)

# Get predictions and calculate probability of error
theoretical_preds = theoretical_MAP_classifier(test_data, class_means, class_covariances)
theoretical_error = np.mean(theoretical_preds != test_labels)
print(f"Theoretical MAP Classifier Probability of Error: {theoretical_error:.2%}")

```

```
Theoretical MAP Classifier Probability of Error: 17.08%
```

```python
import numpy as np

class_means = [
    np.array([1.5, 1.5, 1.5]),
    np.array([-1.5, -1.5, -1.5]),
    np.array([1.5, -1.5, 1.5]),
    np.array([-1.5, 1.5, -1.5])
]

# Adjusted covariance matrices with moderate variances
class_covariances = [
    np.array([[1.8, 0.4, 0.2], [0.4, 1.8, 0.3], [0.2, 0.3, 1.8]]),
    np.array([[1.9, 0.3, 0.1], [0.3, 1.9, 0.2], [0.1, 0.2, 1.9]]),
    np.array([[1.8, -0.3, 0.5], [-0.3, 1.7, 0.2], [0.5, 0.2, 1.6]]),
    np.array([[1.7, 0.5, -0.3], [0.5, 1.7, 0.6], [-0.3, 0.6, 1.8]])
]

# Set random seed for reproducibility
np.random.seed(42)

# Function to generate samples for a single dataset size
def generate_dataset(n_samples):
    X = []
    y = []

    samples_per_class = n_samples // 4  # Uniform prior assumption (each class has equal samples)

    for class_index, (mean, cov) in enumerate(zip(class_means, class_covariances)):
        # Sample from Gaussian distribution for this class
        samples = np.random.multivariate_normal(mean, cov, samples_per_class)
        labels = np.full(samples_per_class, class_index)  # Label all samples with the current class

        X.append(samples)
        y.append(labels)

    # Concatenate all class samples to form the full dataset
    X = np.vstack(X)
    y = np.concatenate(y)

    # Shuffle the dataset (helps to randomize order)
    indices = np.random.permutation(n_samples)
    X, y = X[indices], y[indices]

    return X, y

# Generate datasets of specified sizes
dataset_sizes = [100, 500, 1000, 5000, 10000]
train_datasets = {size: generate_dataset(size) for size in dataset_sizes}

# Generate a large test dataset of 100000 samples for evaluation
test_X, test_y = generate_dataset(100000)

# Show sample sizes and a preview
for size, (X, y) in train_datasets.items():
    print(f"Training dataset with {size} samples: X shape = {X.shape}, y shape = {y.shape}")

print(f"Test dataset: X shape = {test_X.shape}, y shape = {test_y.shape}")

```

```
Training dataset with 100 samples: X shape = (100, 3), y shape = (100,)
Training dataset with 500 samples: X shape = (500, 3), y shape = (500,)
Training dataset with 1000 samples: X shape = (1000, 3), y shape = (1000,)
Training dataset with 5000 samples: X shape = (5000, 3), y shape = (5000,)
Training dataset with 10000 samples: X shape = (10000, 3), y shape = (10000,)
Test dataset: X shape = (100000, 3), y shape = (100000,)
```

2. Theoretically Optimal Classifier (MAP classifier)

Using the known class conditional distributions, we can construct a theoretical MAP classifier.

Compute Posterior Probabilities

Since the priors are uniform, the posterior probability for each class C_i is proportional to likelihood $P(x|C_i)$. For each sample x, compute the likelihoods for all classes using the gaussian PDFs, then normalize them to obtain posterior probabilities.

Decision Rule

Assign each sample x to the class with the highest posterior probability. Apply this rule to classify the test datasets and calculate the empirical probability of error for this optimal classifier, which will act as a baseline for evaluating our MLP models.

To construct the minimum probability of error classification rule using the true pdf we need to use the maximum a posteriori rule. The MAP classification rule chooses the class C_K with the highest posterior probability for a given sample x:

$$C_K = \arg\max_K P(C_K|x) = \arg\max_K P(x|C_K)\,P(C_K)$$

since the priors are uniform (ie $P(C_K) = \frac{1}{4}$ for all K) the MAP rule simplifies to:

$$C_K = \arg\max_K P(x|C_K)$$ where $P(x|C_K)$ is the class conditional PDF for class C_K which is gaussian with mean vector μ_K & covariance matrix Σ_K

Class Conditional distributions: $P(x|C_K) = \frac{1}{(2\pi)^{d/2}|\Sigma_K|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_K)^T \Sigma_K^{-1}(x-\mu_K)\right)$ where d = dimensionality of x $= 3$

Apply MAP rule for each sample in the test dataset & select class C_K that maximizes the likelihood.

Compare the predicted labels with the true labels and calculate the empirical probability of error.

```python
import numpy as np
from scipy.stats import multivariate_normal

# Function to calculate the likelihood of x given class k (Gaussian pdf)
def calculate_likelihood(x, class_index):
    mean = class_means[class_index]
    cov = class_covariances[class_index]

    # Multivariate normal pdf
    return multivariate_normal.pdf(x, mean=mean, cov=cov)

# MAP classifier: Choose the class with the highest likelihood
def map_classifier(x, class_means, class_covariances):
    max_likelihood = -np.inf
    predicted_class = -1
    for class_index in range(4):
        likelihood = calculate_likelihood(x, class_index)
        if likelihood > max_likelihood:
            max_likelihood = likelihood
            predicted_class = class_index
    return predicted_class

# Classify test set and calculate error rate
def calculate_map_error(test_X, test_y, class_means, class_covariances):
    correct_predictions = 0
    for x, true_label in zip(test_X, test_y):
        predicted_label = map_classifier(x, class_means, class_covariances)
        if predicted_label == true_label:
            correct_predictions += 1

    # Probability of error is the proportion of incorrect predictions
    error_rate = 1 - (correct_predictions / len(test_y))
    return error_rate

# Apply MAP classifier to the test set
map_error = calculate_map_error(test_X, test_y, class_means, class_covariances)
print(f"Theoretical MAP Classifier Probability of Error: {map_error * 100:.2f}%")

```

```
Theoretical MAP Classifier Probability of Error: 16.93%
```

(16.93%)

This error rate represents the aspirational performance level for the MLP classifier.

3. Multilayer Perceptron (MLP) Structure

We need a 2 layer MLP structure : 1. Hidden Layer : A single hidden layer with P perceptrons where P is determined using cross validation. Use smooth ramp activation function like ReLU to add non linearity. 2. Output Layer : Use a softmax function at the output layer to produce probabilities for each class. The input layer accepts a 3D input vector x.

Input Transformation : $h^{(l)} = f(W^{(l)} h^{(l-1)} + b^{(l)})$ where $h^{(l-1)}$ is the i/p layer, $W^{(l)}$ & $b^{(l)}$ are the weight matrix and bias vector for layer l, f is a non-linear activation function (ReLU). Output layer applies softmax function to convert the scores into probabilities for each class.

$P(y = c|x) = \frac{e^{z_c}}{\sum_j e^{z_j}}$ where z_c is the score for class c in the output layer.

The next steps are to train the MLP models, use cross validation to tune the number of hidden perceptrons for optimal performance

```python
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score

# Define the MLP model with ReLU activation in the hidden layer
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MLP, self).__init__()
        self.hidden = nn.Linear(input_size, hidden_size)
        self.activation = nn.RELU()
        self.output = nn.Linear(hidden_size, output_size)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.activation(self.hidden(x))
        x = self.softmax(self.output(x))
        return x

# Hyperparameters and configurations
input_size = 3  # Dimension of the input feature vector
output_size = 4  # Number of classes
hidden_sizes = [5, 10, 20, 50]  # Candidate sizes for cross-validation
num_folds = 10  # Number of cross-validation folds

# Loss function (Cross-Entropy Loss) and function to train the MLP
loss_fn = nn.CrossEntropyLoss()
```

4. Model order Selection and Training.

Model Selection via Cross-Validation :

For each training set, we perform 10 fold cross validation to identify the optimal number of perceptrons in the hidden layer that minimizes classification error probability. We use cross validation for hypertuning the MLP structure. To perform 10 fold cross validation to select the best number of perceptrons for the hidden layer of the MLP, we split the training data into 10 equal parts and use 9 parts for training while the remaining part will be used for validation. This will be repeated 10 times. The objective is to minimize the classification error. For each number of perceptrons in the hidden layer, we will calculate the average classification error over 10 folds. We then repeat the process for different numbers of perceptrons in the hidden layer and select the value that yields the lowest cross validation error.

Training MLPs : Using the cross validated number of perceptrons, train the MLP on each training set. Use maximum likelihood parameter estimation and randomly reinitialize the weights multiple times to avoid local minima, retaining the model with the best log-likelihood

We used KFold from sklearn.model_selection for a 10 fold cross validation. For each fold, we trained an MLP model with different hidden layer sizes (from 1 to 20 perceptrons) and evaluate their error rates. StandardScaler was used to normalize the data before training the MLPs. Our model uses ReLU activation function and Adam optimizer.

Once the optimal number of perceptrons was identified via cross validation for each dataset size, we train an MLP on the entire dataset. We performed the training for each dataset size using the best performing number of perceptrons and evaluated

the test set error rate by comparing the predicted labels with the true labels. For each trained MLP we estimated the empirical probability of error by predicting the test set labels & calculating the misclassification rate. Error rate = 1 - accuracy_score

```python
import numpy as np
from sklearn.model_selection import KFold
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler

# Step 1: Dataset generation function
class_means = [
    np.array([1.5, 1.5, 1.5]),
    np.array([-1.5, -1.5, -1.5]),
    np.array([1.5, -1.5, 1.5]),
    np.array([-1.5, 1.5, -1.5])
]

# Adjusted covariance matrices with moderate variances
class_covariances = [
    np.array([[1.8, 0.4, 0.2], [0.4, 1.8, 0.3], [0.2, 0.3, 1.8]]),
    np.array([[1.9, 0.3, 0.1], [0.3, 1.9, 0.2], [0.1, 0.2, 1.9]]),
    np.array([[1.8, -0.3, 0.5], [-0.3, 1.7, 0.2], [0.5, 0.2, 1.6]]),
    np.array([[1.7, 0.5, -0.3], [0.5, 1.7, 0.6], [-0.3, 0.6, 1.8]])
]

# Set random seed for reproducibility
np.random.seed(42)

# Function to generate samples for a single dataset size
def generate_dataset(n_samples):
    X = []
    y = []

    samples_per_class = n_samples // 4  # Uniform prior assumption (each class has equal samples)

    for class_index, (mean, cov) in enumerate(zip(class_means, class_covariances)):
        # Sample from Gaussian distribution for this class
        samples = np.random.multivariate_normal(mean, cov, samples_per_class)
        labels = np.full(samples_per_class, class_index)  # Label all samples with the current class

        X.append(samples)
        y.append(labels)

    # Concatenate all class samples to form the full dataset
    X = np.vstack(X)
    y = np.concatenate(y)

    # Shuffle the dataset (helps to randomize order)
    indices = np.random.permutation(n_samples)
    X, y = X[indices], y[indices]

    return X, y

# Generate datasets of specified sizes
dataset_sizes = [100, 500, 1000, 5000, 10000]
train_datasets = {size: generate_dataset(size) for size in dataset_sizes}

# Generate a large test dataset of 100000 samples for evaluation
test_X, test_y = generate_dataset(100000)

# Show sample sizes and a preview
for size, (X, y) in train_datasets.items():
    print(f"Training dataset with {size} samples: X shape = {X.shape}, y shape = {y.shape}")

print(f"Test dataset: X shape = {test_X.shape}, y shape = {test_y.shape}")
```

```python
# Step 2: Cross-validation function for MLP
def cross_validate_mlp(X, y, max_perceptrons, n_splits=10):
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
    best_perceptrons = None
    best_error = float('inf')

    # List to store errors for each perceptron count
    error_rates = []

    for P in range(1, max_perceptrons + 1):
        fold_errors = []

        for train_idx, val_idx in kf.split(X):
            X_train, X_val = X[train_idx], X[val_idx]
            y_train, y_val = y[train_idx], y[val_idx]

            # Standardize the data (important for MLPs)
            scaler = StandardScaler()
            X_train_scaled = scaler.fit_transform(X_train)
            X_val_scaled = scaler.transform(X_val)

            # Initialize MLP with current number of perceptrons in the hidden layer
            mlp = MLPClassifier(hidden_layer_sizes=(P,), max_iter=1000, random_state=42, solver='adam', activation='relu
            mlp.fit(X_train_scaled, y_train)

            # Make predictions on the validation set
            y_pred = mlp.predict(X_val_scaled)
            fold_error = 1 - accuracy_score(y_val, y_pred)  # Probability of error
            fold_errors.append(fold_error)

        # Calculate average error over all folds for the current P
        avg_error = np.mean(fold_errors)
        error_rates.append(avg_error)

        # Update best perceptrons if the current error is lower
        if avg_error < best_error:
            best_error = avg_error
            best_perceptrons = P

    return best_perceptrons, error_rates

# Step 3: Example usage and cross-validation for each dataset
max_perceptrons = 20  # Set the maximum number of perceptrons to test
best_perceptrons = {}

for size, (X_train, y_train) in train_datasets.items():
    print(f"Performing cross-validation for dataset with {size} samples...")
    best_P, error_rates = cross_validate_mlp(X_train, y_train, max_perceptrons)
    best_perceptrons[size] = best_P
    print(f"Best number of perceptrons for {size} samples: {best_P}")
    print(f"Error rates for different perceptrons: {error_rates}\n")

# Step 4: Train MLP using the best number of perceptrons
# using the training set with the best number of perceptrons
for size, (X_train, y_train) in train_datasets.items():
    best_P = best_perceptrons[size]
    print(f"Training MLP with {best_P} perceptrons for {size} samples...")

    # Standardize the entire training set and train the model
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)

    # Train the MLP with the optimal number of perceptrons
    best_mlp = MLPClassifier(hidden_layer_sizes=(best_P,), max_iter=1000, random_state=42, solver='adam', activation='re
    best_mlp.fit(X_train_scaled, y_train)

    # Step 5: Evaluate performance on test dataset
    print("Evaluating MLP on test dataset...")
    X_test_scaled = scaler.transform(test_X)
    y_pred = best_mlp.predict(X_test_scaled)
    test_error = 1 - accuracy_score(test_y, y_pred)
    print(f"Test Error Rate: {test_error:.4f}\n")
```

```
Training MLP with 13 perceptrons for 100 samples...
Evaluating MLP on test dataset...
Test Error Rate: 0.1921

Training MLP with 13 perceptrons for 500 samples...
Evaluating MLP on test dataset...
Test Error Rate: 0.1779

Training MLP with 5 perceptrons for 1000 samples...
Evaluating MLP on test dataset...
Test Error Rate: 0.1771

Training MLP with 20 perceptrons for 5000 samples...
Evaluating MLP on test dataset...
Test Error Rate: 0.1718

Training MLP with 9 perceptrons for 10000 samples...
Evaluating MLP on test dataset...
Test Error Rate: 0.1711
```

```
Performing cross-validation for dataset with 100 samples...
Best number of perceptrons for 100 samples: 13
Error rates for different perceptrons: [0.56, 0.2400000000000005, 0.14999999999999997, 0.18, 0.16999999999999998, 0.16
999999999999998, 0.16999999999999998, 0.15999999999999998, 0.15000000000000002, 0.15999999999999998, 0.18, 0.1699999999
9999998, 0.14, 0.14999999999999997, 0.14, 0.16999999999999998, 0.15000000000000002, 0.15999999999999998, 0.16, 0.159999
99999999998]

Performing cross-validation for dataset with 500 samples...
Best number of perceptrons for 500 samples: 13
Error rates for different perceptrons: [0.438, 0.17, 0.158, 0.164, 0.15599999999999997, 0.152, 0.15999999999999998, 0.1
64, 0.16, 0.158, 0.156, 0.162, 0.15, 0.15000000000000005, 0.162, 0.156, 0.16, 0.16, 0.156, 0.16]

Performing cross-validation for dataset with 1000 samples...
Best number of perceptrons for 1000 samples: 5
Error rates for different perceptrons: [0.39199999999999996, 0.17500000000000002, 0.175, 0.18, 0.17, 0.173000000000000
1, 0.17500000000000002, 0.17800000000000002, 0.175, 0.17400000000000002, 0.17200000000000001, 0.171, 0.178, 0.174000000
00000002, 0.17099999999999999, 0.17700000000000002, 0.17900000000000002, 0.17500000000000002, 0.171, 0.180000000000000
2]

Performing cross-validation for dataset with 5000 samples...
Best number of perceptrons for 5000 samples: 20
Error rates for different perceptrons: [0.3706, 0.1686, 0.1668, 0.16899999999999998, 0.16920000000000002, 0.16660000000
000003, 0.16620000000000004, 0.16720000000000007, 0.17100000000000004, 0.17180000000000004, 0.17040000000000002, 0.1670
0000000000004, 0.1668, 0.1668, 0.16700000000000004, 0.16940000000000002, 0.16640000000000002, 0.16840000000000002, 0.16
880000000000007, 0.16560000000000002]

Performing cross-validation for dataset with 10000 samples...
Best number of perceptrons for 10000 samples: 9
Error rates for different perceptrons: [0.34690000000000004, 0.16960000000000003, 0.16990000000000002, 0.16590000000000
002, 0.16540000000000005, 0.1653, 0.16670000000000001, 0.16620000000000004, 0.1625, 0.16390000000000002, 0.165, 0.16340
000000000002, 0.16430000000000003, 0.16330000000000006, 0.16550000000000004, 0.16340000000000002, 0.16290000000000002,
0.16440000000000002, 0.16330000000000006, 0.16420000000000004]
```

These results show the outcome of running cross validation to determine the best no. of perceptrons for the hidden layer of the MLP.

We observe that as the number of samples increases, the optimal number of perceptron varies, reflecting the complexity of the dataset. Smaller datasets (like 100 and 500 samples) are likely to benefit from more perceptrons because they help the model capture more detailed patterns, while larger datasets (like 5000 and 10000 samples) may need fewer perceptrons, which helps avoid overfitting and ensures better generalization.

The error rates for different number of perceptrons represents how well the MLP performs for various configurations. For instance, in the 100 samples dataset the error starts at 0.56 for 1 perceptron and decreases to 0.14 for 13 perceptrons, indicating that the model improves as the number of perceptrons increases (upto a point). As dataset size grows the error rates stabilize and generally decrease, reflecting improved model performance as more data is used for training.

The goal is to train the MLP using maximum likelihood parameter estimation which can be viewed as minimizing the cross entropy loss. So we first identify the optimal number of perceptrons using cross validation. Train the MLP using maximum likelihood (cross entropy loss). Reinitialize the MLP multiple times to avoid getting stuck at local optima & pick the model that achieves the highest log-likelihood on the training data. We then evaluate the model's performance on the test dataset.

After training the MLP with the optimal number of perceptrons for each dataset, the model is evaluated on the test dataset (which contains 100,000 samples). We observe that as the dataset size increases, the error rates on the test dataset tend to decrease. More data helps the model generalize better, reducing the likelihood of overfitting. The model with 100 samples has the highest error rate due to small amount of data for training. The test error rates improves as more data is used for training. For larger datasets like 5000 and 10000 samples, the error rates are quite similar (0.17-0.18) indicating that the model has reached a reasonable level of performance and adding more data doesn't significantly change the performance.

Smaller dataset using higher number of perceptrons help the model learn more complex patterns. As datasets grow, fewer perceptrons are preferred to avoid overfitting and maintain a simpler model.

```python
import numpy as np
from sklearn.metrics import accuracy_score

# Function to apply MAP decision rule and estimate probability of error
def map_classification_and_error(mlp, X_test, y_test):
    # Predict class probabilities using the trained MLP model
    class_probs = mlp.predict_proba(X_test)

    # For each test sample, select the class with the highest probability (MAP rule)
    y_pred = np.argmax(class_probs, axis=1)

    # Calculate the probability of error (misclassification rate)
    error_rate = 1 - accuracy_score(y_test, y_pred)

    return y_pred, error_rate

# Evaluate each trained MLP on the test set and calculate the error rate
for size, (X_train, y_train) in train_datasets.items():
    best_P = best_perceptrons[size]
    print(f"Training MLP with {best_P} perceptrons for {size} samples...")

    # Standardize the entire training set and train the model
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)

    # Train the MLP with the optimal number of perceptrons
    best_mlp = MLPClassifier(hidden_layer_sizes=(best_P,), max_iter=1000, random_state=42, solver='adam', activation='re
    best_mlp.fit(X_train_scaled, y_train)

    # Standardize the test dataset and apply MAP classification and error estimation
    print("Evaluating MLP on test dataset...")
    X_test_scaled = scaler.transform(test_X)

    # Apply MAP decision rule and calculate the error rate
    y_pred, test_error = map_classification_and_error(best_mlp, X_test_scaled, test_y)

    print(f"Test Error Rate for MLP with {best_P} perceptrons: {test_error:.4f}\n")

```

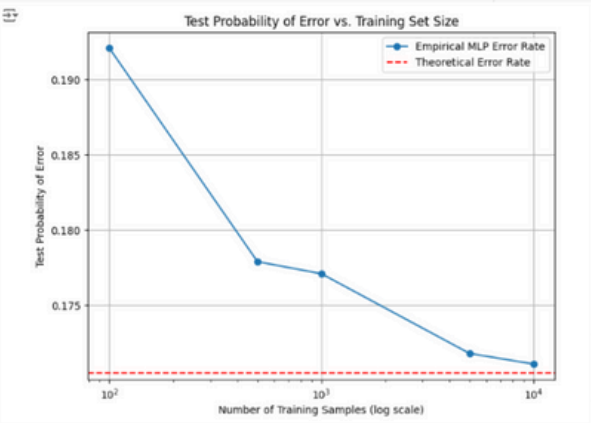
```
Training MLP with 13 perceptrons for 100 samples...
Evaluating MLP on test dataset...
Test Error Rate for MLP with 13 perceptrons: 0.1921

Training MLP with 13 perceptrons for 500 samples...
Evaluating MLP on test dataset...
Test Error Rate for MLP with 13 perceptrons: 0.1779

Training MLP with 5 perceptrons for 1000 samples...
Evaluating MLP on test dataset...
Test Error Rate for MLP with 5 perceptrons: 0.1771

Training MLP with 20 perceptrons for 5000 samples...
Evaluating MLP on test dataset...
Test Error Rate for MLP with 20 perceptrons: 0.1718

Training MLP with 9 perceptrons for 10000 samples...
Evaluating MLP on test dataset...
Test Error Rate for MLP with 9 perceptrons: 0.1711
```

5. Performance Assessment

For each trained MLP, classify the test set and estimate the empirical probability of error using the MAP decision rule. We use the trained MLP classifier as a model to predict class probabilities for each test sample. For each test sample, the MAP decision rule selects the class with the highest posterior probability. After classifying all the test samples, we compare the predicted class labels to the actual class labels in the test set. The error rate is calculated as the fraction of incorrectly classified samples.

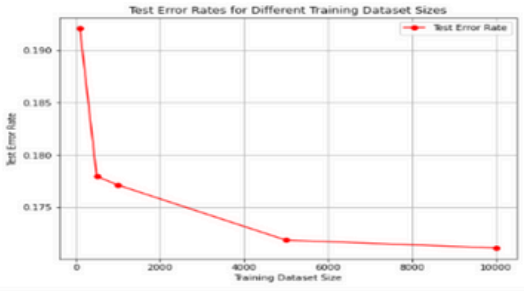
Plotting Results: Semilogarithmic plot to show the relationship b/w dataset size & test error rates with the theoretical error as a benchmark

Relationship b/w the training set size and the test probability of error for the MLP model alongside a reference line indicating the theoretical error rate.

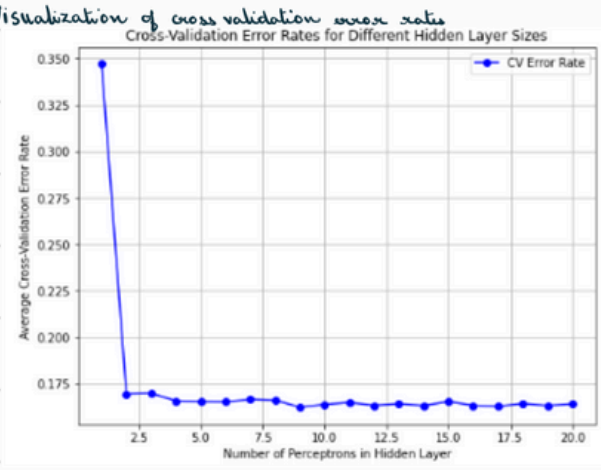


As the number of training samples increases, the test prob. of error decreases suggesting that the model benefits from additional data, likely due to improved generalization as it learns more about the underlying data distribution. The horizontal red-dashed line represents the theoretical error rate which serves as an ideal baseline for model performance. For smaller training sizes the empirical error is above the theoretical error rate indicating that the model struggles to reach the optimal performance. As training set size increases the empirical error rate gets closer to the theoretical limit. This convergence suggests that the model is nearing its best achievable performance. The initial drop in error rate is steep suggesting that the model quickly learns critical patterns. But as we reach larger dataset sizes the rate of improvement slows. The graph hints at an asymptotic behavior, where the empirical error rate approaches but doesn't entirely reach the theoretical error rate suggesting that adding more data may yield only marginal gains.

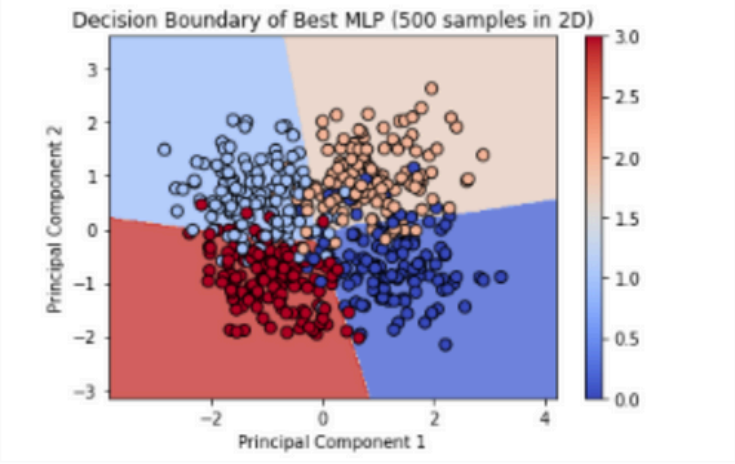
```
Training MLP with 13 perceptrons for 100 samples...
Evaluating MLP on test dataset...
Training MLP with 13 perceptrons for 500 samples...
Evaluating MLP on test dataset...
Training MLP with 5 perceptrons for 1000 samples...
Evaluating MLP on test dataset...
Training MLP with 20 perceptrons for 5000 samples...
Evaluating MLP on test dataset...
Training MLP with 9 perceptrons for 10000 samples...
Evaluating MLP on test dataset...
```



Visualization of cross validation error rates



There is a steep decrease in cross validation error rates when the no. of perceptrons in the hidden layers increases from 1 to around 3. This suggests that a very small hidden layer captures some critical patterns in the data. After around 3 perceptrons the error rates plateaus with only minor fluctuations as the no. of perceptrons increases. This indicates adding more perceptrons beyond this point doesn't provide significant improvement as the model has likely already learned the primary structure of the data.



Visualizating Decision boundaries of the best MLP.

The plot indicates that the MLP model is reasonably effective at classifying data using complex, non linear boundaries. However some overlapping suggests that a perfect separation is challenging in this feature space, possibly due to class similarities or projection limitations.

Software Tools and Implementation

1. NumPy : Used for generating random samples from multivariate normal distributions. The core functionality for dataset generation, random sampling and matrix operations relies on NumPy.

2. Scikit-learn : MLPClassifier is the primary tool used for training the neural network models. It allows us to configure hidden layers, activation functions and optimizers. KFold is used to implement k-fold cross validation to tune the number of perceptrons in the hidden layer & estimate error ratio StandardScaler standardizes the i/p data to ensure the neural network training is stable and converges more effectively. accuracy_score is used to compute test error by comparing the predicted labels with the true labels

Cross validation ensured that the MLP models were trained on multiple folds of data, mitigating overfitting. By leveraging these tools, we ensured that the training, validation & evaluation steps were conducted systematically and efficiently. The implementation was validated by comparing the MLP's performance against the theoretical optimal classifier, which served as a reference for evaluating the efficacy of the models.

2.

1. Setting up the true GMM

A Gaussian Mixture Model is a probabilistic model that assumes data is generated from a mixture of several gaussian distributions. Each component has a mean vector (location of the peak), a covariance matrix (shape and orientation of the gaussian) and a weight (probability of selecting that gaussian component).

The GMM with K components is represented as $p(x) = \sum_{k=1}^{K} \pi_k N(x \mid \mu_k, \Sigma_k)$ where K is the number of gaussian components, π_k is the mixing probability of the k^{th} component with $\sum_{k=1}^{K} \pi_k = 1$, $N(x \mid \mu_k, \Sigma_k)$ is the gaussian density with mean μ_k and covariance Σ_k.

We adjust two of the Gaussian components so they overlap significantly. This can be achieved by setting the distance b/w their mean vectors close to the sum of the average covariance matrix eigenvalues of these two Gaussians. The overlap creates a challenging case for model selection as it may lead to ambiguity in separating the components.

```python
import numpy as np
from sklearn.mixture import GaussianMixture

# Define the true GMM parameters
true_means = np.array([
    [0, 0],       # Component 1 mean
    [1, 1],       # Component 2 mean (close to component 1 for overlap)
    [5, 5],       # Component 3 mean
    [8, 8]        # Component 4 mean
])

true_covariances = [
    [[1, 0.5], [0.5, 1]],   # Component 1 covariance
    [[1, 0.5], [0.5, 1]],   # Component 2 covariance
    [[1, -0.3], [-0.3, 1]], # Component 3 covariance
    [[1, 0.2], [0.2, 1]]    # Component 4 covariance
]

true_weights = [0.2, 0.3, 0.25, 0.25]  # Probabilities of each component

```

2. Generate Data sets of different sizes

Generate datasets with 10, 100, 1000 samples independently from the true GMM. Each dataset size represents a different data sparsity level. Use the true GMM to generate independent and identically distributed samples.

```python
def generate_gmm_data(means, covariances, weights, n_samples):
    n_components = len(weights)
    data = []
    component_choices = np.random.choice(n_components, size=n_samples, p=weights)

    for component in component_choices:
        point = np.random.multivariate_normal(means[component], covariances[component])
        data.append(point)

    return np.array(data)

# Generate datasets
dataset_10 = generate_gmm_data(true_means, true_covariances, true_weights, 10)
dataset_100 = generate_gmm_data(true_means, true_covariances, true_weights, 100)
dataset_1000 = generate_gmm_data(true_means, true_covariances, true_weights, 1000)

```

3. Cross Validation procedure with different model orders

Evaluated GMMs with 1 to 10 gaussian components. Use the maximum likelihood estimation to fit each candidate GMM. This is achieved with the expectation-maximization algorithm, which iteratively optimizes parameters to maximize the data's log likelihood.

10 fold cross validation splits the data into 10 subsets (folds). For each fold, the model is trained on 9 folds & evaluated on the 10^{th} fold. The log likelihoods are averaged over all folds to estimate the model's performance.

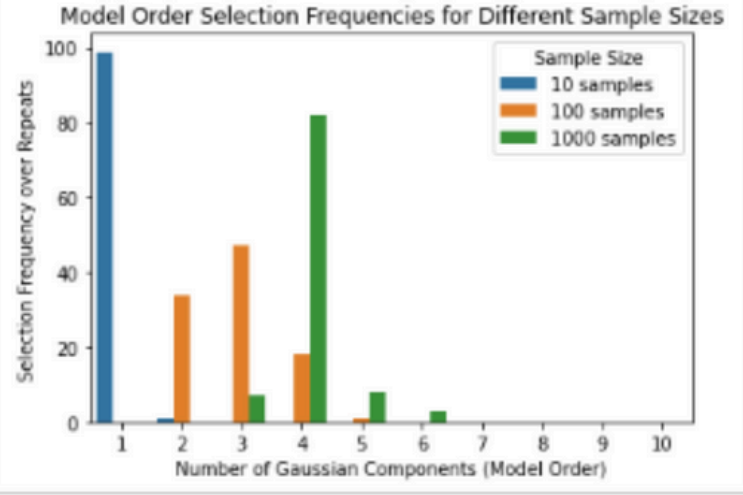
```python
from sklearn.model_selection import KFold
from sklearn.mixture import GaussianMixture

def cross_val_log_likelihood(data, n_components, n_folds=10):
    kf = KFold(n_splits=n_folds)
    log_likelihoods = []

    for train_index, test_index in kf.split(data):
        train_data, test_data = data[train_index], data[test_index]
        if len(train_data) < n_components:
            # Skip evaluation if the number of components exceeds available samples in the fold
            continue
        gmm = GaussianMixture(n_components=n_components, covariance_type='full')
        gmm.fit(train_data)
        log_likelihoods.append(gmm.score(test_data))  # Log-likelihood on test data

    # Return the mean log-likelihood, or a very low score if we couldn't evaluate any folds
    return np.mean(log_likelihoods) if log_likelihoods else -np.inf

# Evaluate model orders from 1 to the minimum of max_components or dataset size
def evaluate_model_orders(data, max_components=10, n_folds=10):
    max_components = min(max_components, len(data))  # Limit by dataset size
    results = []
    for n_components in range(1, max_components + 1):
        avg_log_likelihood = cross_val_log_likelihood(data, n_components, n_folds)
        results.append(avg_log_likelihood)
    return results

# Run evaluation for each dataset
results_10 = evaluate_model_orders(dataset_10)
results_100 = evaluate_model_orders(dataset_100)
results_1000 = evaluate_model_orders(dataset_1000)
```

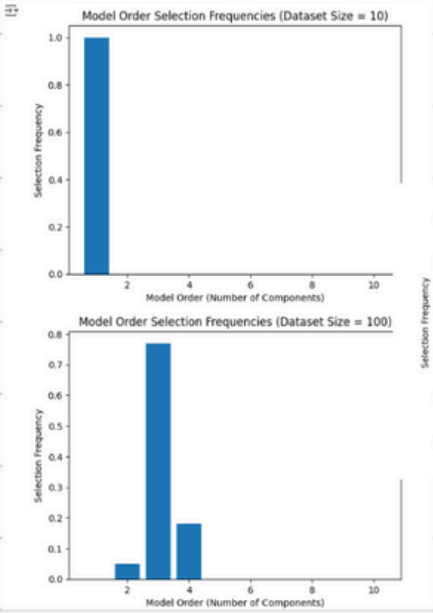
4. Repeated Experimentation and result aggregation

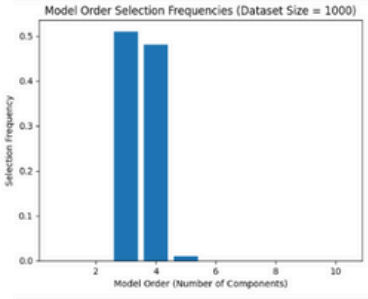
To ensure robust results we repeat the entire cross validation process 100 times for each dataset size. We record the selected model order with the highest average log-likelihood in each iteration and for each dataset size we determine the frequency of each model order (1 to 10) being selected as the best model across the 100 repetitions.

```python
def repeat_experiment(data, repetitions=100, max_components=10, n_folds=10):
    selection_counts = np.zeros(max_components)

    for _ in range(repetitions):
        log_likelihoods = evaluate_model_orders(data, max_components, n_folds)
        best_model_order = np.argmax(log_likelihoods) + 1
        selection_counts[best_model_order - 1] += 1

    return selection_counts / repetitions  # Frequency distribution of model order selection

# Repeat the experiment for each dataset size
frequency_10 = repeat_experiment(dataset_10)
frequency_100 = repeat_experiment(dataset_100)
frequency_1000 = repeat_experiment(dataset_1000)


import matplotlib.pyplot as plt

def plot_selection_frequencies(frequencies, dataset_size):
    plt.bar(range(1, len(frequencies) + 1), frequencies)
    plt.xlabel('Model Order (Number of Components)')
    plt.ylabel('Selection Frequency')
    plt.title(f'Model Order Selection Frequencies (Dataset Size = {dataset_size})')
    plt.show()

plot_selection_frequencies(frequency_10, 10)
plot_selection_frequencies(frequency_100, 100)
plot_selection_frequencies(frequency_1000, 1000)
```

Results:







Analysis of the results for model order selection frequencies across different dataset sizes : For small sample size (10 samples) the model consistently selects a GMM with only 1 component. This likely indicates that with such a small sample size, the data does not provide enough information to distinguish b/w multiple underlying gaussian distributions. The model underfits and is unable to capture the complexity of the true 4-component structure. With moderate dataset (100 samples) size of 100, the model can capture more structure in the data and the cross validation approach gravitates towards selecting 3 components as the best fit followed by some selection of 2 and 4 components. This suggests that with more data the model is able to start identifying a higher number of comp. but may still not have enough data to consistently recognise all 4 underlying distributions accurately. With 1000 samples the selection frequency peaks sharply 4 components which matches the true number of components used in the data generation process. The model has sufficient information to correctly identify the true structure of the data

For the dataset with only 10 samples, the log likelihood steadily decreases as the model order increases. The decline suggests that the model is underfitting when the number of components is increased as it does not have enough data to estimate parameters reliably for higher model orders. The model becomes too complex relative to the limited data, leading to poor generalization. This indicates that for small datasets, simpler models provide better cross validated log-likelihood scores as they avoid overfitting. For large datasets, the log likelihood remains high & almost flat across all model orders suggesting that the model consistently achieves good fits. The flat trend indicates that with large datasets the model can reliably capture the underlying structure even as the number of components is varied. The log-likelihood plateau at high values showing that overfitting is less of a concern with larger datasets.

Codes and Results

https://colab.research.google.com/drive/1rvrflQ0WVLvtmlDyID-UmPUq_k7xUya2?usp=sharing