EECE5644 Assignment 3

Mariona Jaramillo Civill

November 12, 2024

NUID: 002413201 - `jaramillocivill.m@northeastern.edu`

Problem 1

The objective of this problem is to train multilayer perceptrons (MLPs) to approximate class label posteriors for a four-class classification problem, applying a maximum a posteriori (MAP) classification rule to minimize the probability of error under a 0-1 loss. The data distribution consists of four classes with uniform priors, $P(C_i) = 0.25$ for each class C_i, where each class-conditional pdf is modeled as a Gaussian distribution over a 3-dimensional random vector $\mathbf{x} \in \mathbb{R}^3$. Mean vectors and covariance matrices for each class were selected to ensure overlapping distributions, yielding a theoretical MAP classifier error rate between around 10%.

The means and covariance matrices used for the Gaussian distributions in each class are as follows:

$$\mathbf{m}_0 = \begin{bmatrix} 0.5 \\ 0.0 \\ 0.0 \end{bmatrix}, \qquad \mathbf{C}_0 = \begin{bmatrix} 1.0 & 0.2 & 0.1 \\ 0.2 & 0.2 & 0.2 \\ 0.1 & 0.2 & 1.0 \end{bmatrix}$$

$$\mathbf{m}_1 = \begin{bmatrix} 0.0 \\ 1.0 \\ 0.0 \end{bmatrix}, \qquad \mathbf{C}_1 = \begin{bmatrix} 1.2 & 0.1 & 0.2 \\ 0.1 & 0.3 & 0.1 \\ 0.2 & 0.1 & 1.2 \end{bmatrix}$$

$$\mathbf{m}_2 = \begin{bmatrix} 2.0 \\ 2.0 \\ 2.0 \end{bmatrix}, \qquad \mathbf{C}_2 = \begin{bmatrix} 0.8 & 0.3 & 0.1 \\ 0.3 & 0.3 & 0.3 \\ 0.1 & 0.3 & 1.1 \end{bmatrix}$$

$$\mathbf{m}_3 = \begin{bmatrix} 3.0 \\ 4.0 \\ 0.0 \end{bmatrix}, \qquad \mathbf{C}_3 = \begin{bmatrix} 0.2 & 0.1 & 0.3 \\ 0.1 & 0.9 & 0.2 \\ 0.3 & 0.2 & 1.0 \end{bmatrix}$$

Each mean vector corresponds to one of the Gaussian classes, and each covariance matrix defines the spread and orientation of the distribution for each class, allowing moderate overlap between classes. To ensure that there exists this overlap, we have chosen means and covariances that empirically achieve around 10% of probability of error when a MAP classifier that uses the true data pdf is implemented.

The MLP model is structured with an input layer of 3 units (one per dimension of $\mathbf{x}$), a single hidden layer, and an output layer with 4 units corresponding to each class. In the hidden layer, we use a smooth activation function, specifically the Exponential Linear Unit (ELU), and apply a softmax function at the output layer to normalize the outputs as class probabilities. To train the model, we minimize the cross-entropy loss function, $\mathcal{L} = -\sum_{i=1}^{N}\sum_{j=1}^{4} y_{ij}\log(\hat{y}_{ij})$, where y_{ij} is the true label and $\hat{y}_{ij}$ the MLP's predicted probability for each sample-class pair.

Training datasets with 100, 500, 1000, 5000, 10000 samples and a test dataset with 100000 samples have been generated. In Figure 1, the data distribution among classes for the training datasets can be visualized, effectively seeing that there is overlapping. Also, we found the theoretically optimal classifier using the knowledge of our true data pdf and then estimated the empirically minimum probability of error for this classifier on the test dataset, which has turned out to be 0.1058

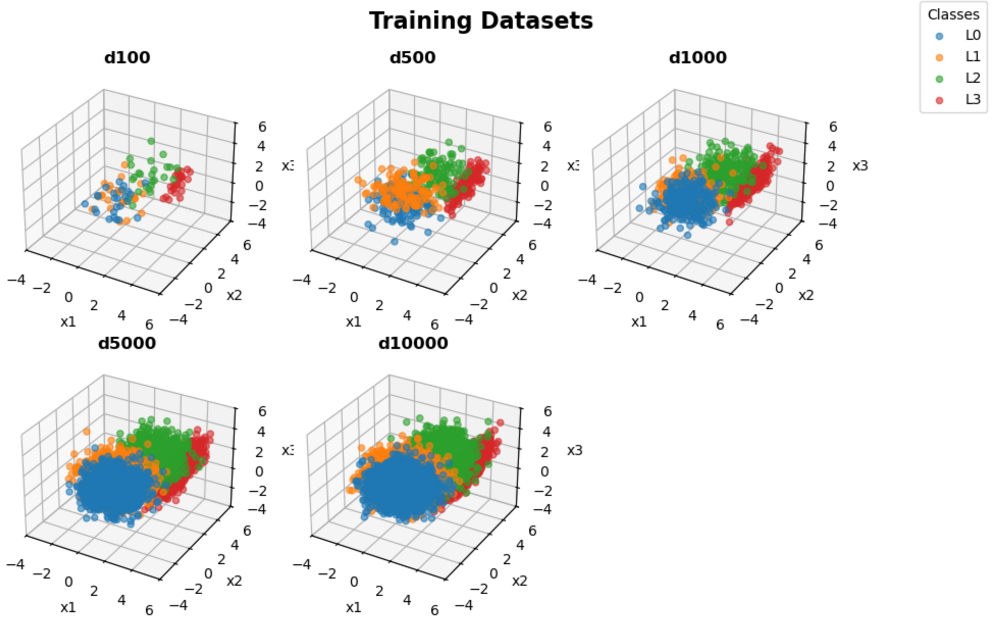

Figure 1: Training datasets distribution

To determine the optimal number of perceptrons in the hidden layer, we used 10-fold cross-validation, evaluating models with different hidden layer sizes for each training dataset size. For each hidden size P, the dataset was split into 10 folds, with the model trained on 9 folds and validated on the remaining fold. This process was repeated across all folds, and the hidden size P that minimized average classification error was selected (Figure 2).

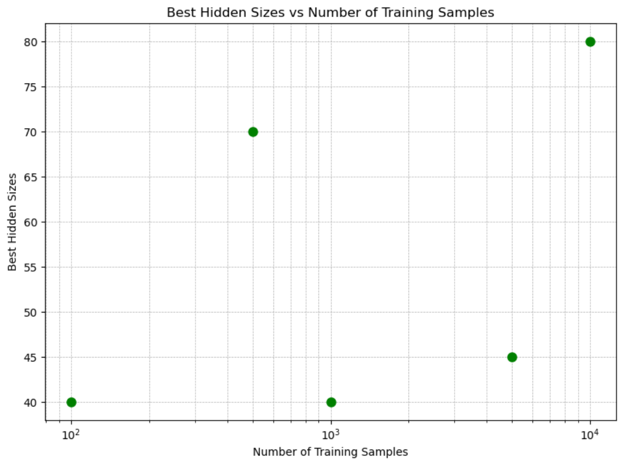

Figure 2: Best hidden size for MLP models trained on datasets of varying sizes after performing crossvalidation

Using the true parameters of each class, we constructed a theoretically optimal MAP classifier by assigning a sample $\mathbf{x}$ to the class C_k that maximizes $P(C_k|\mathbf{x}) = \frac{p(\mathbf{x}|C_k)P(C_k)}{\sum_{j=1}^{4} p(\mathbf{x}|C_j)P(C_j)}$. This MAP classifier served as a benchmark for minimum achievable error.

Empirical probability of error was computed for each trained MLP model on a test set of 100,000 samples. These values were plotted against the number of training samples used for each model, alongside the probability of error for the theoretically optimal classifier as a reference Figure 3. As shown, MLP classifiers approach the optimal probability of error as the training set size increases. Cross-validation was effective in selecting an appropriate model complexity for each dataset, reducing overfitting on smaller datasets and enabling more complex models on larger datasets. An exception to this trend is the model trained on 500 samples, which is an outlier with a more complex structure, using 70 hidden perceptrons.

In conclusion, MLP classifiers, with appropriate training and model selection, can approximate MAP classification with low error rates, approaching the theoretical limit as the amount of training data increases. The use of cross-validation to select the number of hidden units allowed for robust model selection across varying data sizes, confirming that MLPs are effective in achieving low-error classification for this problem.

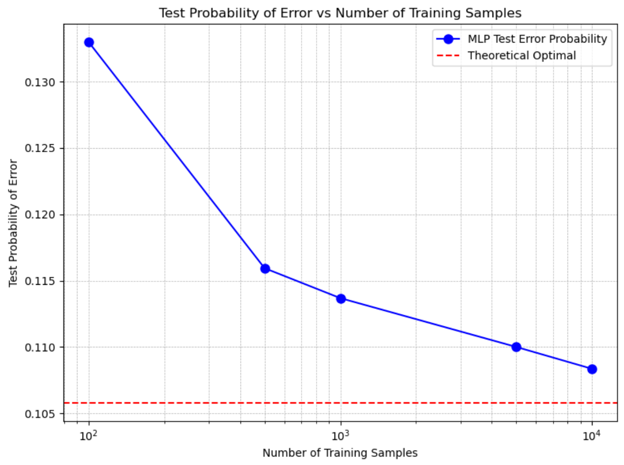

Figure 3: Probability of error for MLP models trained on datasets of varying sizes. The horizontal line represents the theoretically optimal probability of error.

Problem 2

In this experiment, we aimed to determine the optimal model order for a Gaussian Mixture Model (GMM) used to approximate a true probability density function. The true data-generating distribution was specified as a 2-dimensional GMM with 4 components. Each component had a distinct mean vector and covariance matrix, and the probabilities for each component were set to ensure variation in data distribution. Specifically, two of the Gaussian components were made to overlap significantly by setting the distance between their mean vectors comparable to the sum of their average covariance matrix eigenvalues. This overlapping structure introduces complexity in identifying the true model order, adding robustness to our model selection task.

The covariance matrices for the four Gaussian components are defined as:

$$\Sigma_0 = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}, \quad \Sigma_1 = \begin{bmatrix} 1 & -0.6 \\ -0.6 & 1 \end{bmatrix}, \quad \Sigma_2 = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}, \quad \Sigma_3 = \begin{bmatrix} 1 & -0.7 \\ -0.7 & 1 \end{bmatrix}$$

The eigenvalues for each covariance matrix Σ_i are denoted as λ_{i1} and λ_{i2}. For example, the eigenvalues for Σ_0 are λ_{01} and λ_{02}.

The average eigenvalue for Σ_0 is:

$$\bar{\lambda}_0 = \frac{\lambda_{01} + \lambda_{02}}{2}$$

To create an overlap between the first two Gaussian components, we set the distance between their mean vectors to be equal to the sum of the average eigenvalues of Σ_0 and Σ_1:

$$d_{\text{overlap}} = \bar{\lambda}_0 + \bar{\lambda}_1 = 2$$

The mean vectors for the four components are defined as follows:

$$\mu_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mu_1 = \begin{bmatrix} d_{\text{overlap}} \\ d_{\text{overlap}} \end{bmatrix}, \quad \mu_2 = \begin{bmatrix} 6 \\ 0 \end{bmatrix}, \quad \mu_3 = \begin{bmatrix} 3 \\ -3 \end{bmatrix}$$

The weights (probabilities) for each component being selected are specified as:

$$w_0 = 0.2, \quad w_1 = 0.3, \quad w_2 = 0.4, \quad w_3 = 0.1$$

We generated multiple datasets of 10, 100, and 1000 samples each, following the specified true GMM distribution. For each dataset, we evaluated GMMs with model orders from 1 to 9 using maximum likelihood parameter estimation with the Expectation-Maximization (EM) algorithm (A maximum of 9 components was used instead of 10 because, for the case with 10 samples, the maximum cross-validation folds allowed is 10. In this case, each training fold contains only 9 samples, which limits the number of components to 9 or fewer, as a GMM cannot have more components than there are data points in the training set). To determine the most appropriate model order, we performed 10-fold cross-validation on each candidate model order, using the average log-likelihood on the validation folds as the performance metric. The cross-validation process was repeated 100 times for each dataset to obtain reliable results.

For each dataset size, the model order that maximized the average log-likelihood across folds was selected as the best order for that dataset. This selection process was repeated across 100 independent runs, allowing us to compute the selection frequency of each model order. The results of the experiment indicate the rate at which each model order was selected as optimal for each dataset size, as shown in Table 1 and Figure 4

With larger datasets, the GMM models with more components are more frequently selected as the optimal model order. For the smallest dataset (10 samples), the models with fewer components (1-2) were more commonly selected due to limited data, leading to an underestimation of the true model complexity. For the datasets with 100 and 1000 samples, models with 4-6 components were frequently selected, approaching the true complexity of the underlying distribution. This trend demonstrates that larger datasets enable more accurate estimation of the underlying model complexity. Actually, the model with 4 component was the one that obtained highest selection rate for these two datasets, being this the true value for the generated GMM pdf.

Dataset Size	**1 comp**	**2 comp**	**3 comp**	**4 comp**	**5 comp**	**6 comp**	**7 comp**	**8 comp**	**9 comp**
10	0.99	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00
100	0.00	0.00	0.13	0.70	0.16	0.01	0.00	0.00	0.00
1000	0.00	0.00	0.00	0.53	0.35	0.11	0.01	0.00	0.00

Table 1: Selection rates for each GMM model order across different dataset sizes (100 repetitions per dataset size).

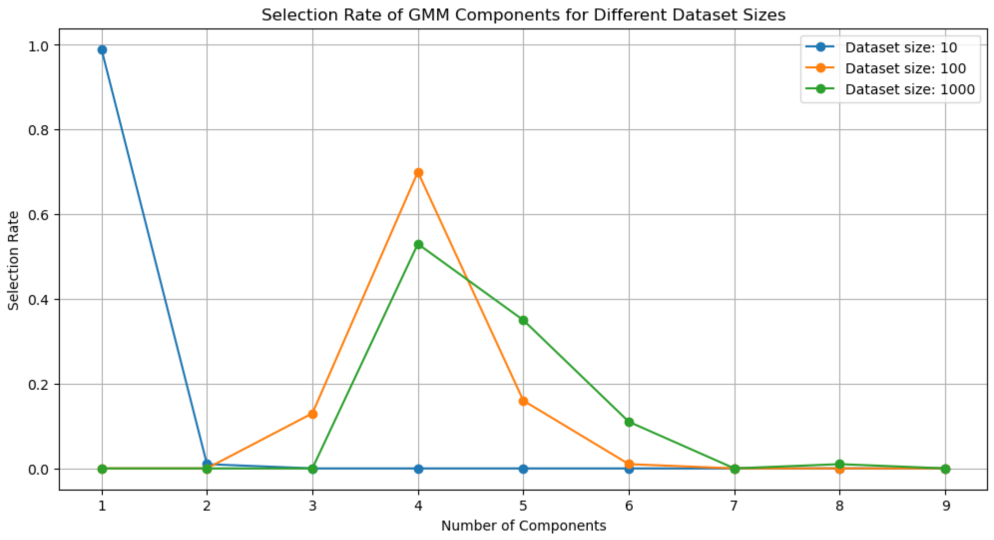

Figure 4: Selection rates of different GMM model orders for each dataset size. Larger datasets favor higher model orders closer to the true complexity of the underlying distribution.

Appendix: Code

Listing 1: Question 1

```
## Define the data distribution
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(0)
torch.manual_seed(0)

# Parameters
num_classes = 4
dim = 3  # 3D feature space


# Parameters for four classes (fixed mean vectors and covariance matrices)
means = [
    np.array([0.5, 0.0, 0.0]),
    np.array([0.0, 1.0, 0.0]),
    np.array([2.0, 2.0, 2.0]),
    np.array([3.0, 4.0, 0.0])
]

# Covariance matrices with moderate diagonal elements and small off-diagonals for overlap
covariances = [
    np.array([[1.0, 0.2, 0.1], [0.2, 0.2, 0.2], [0.1, 0.2, 1.0]]),
    np.array([[1.2, 0.1, 0.2], [0.1, 0.3, 0.1], [0.2, 0.1, 1.2]]),
    np.array([[0.8, 0.3, 0.1], [0.3, 0.3, 0.3], [0.1, 0.3, 1.1]]),
    np.array([[0.2, 0.1, 0.3], [0.1, 0.9, 0.2], [0.3, 0.2, 1.0]])
]

# Uniform class priors
priors = [0.25, 0.25, 0.25, 0.25]

# Generate train/test data based on the specified distribution
def generate_data(n_samples):
    X, y = [], []
    for i in range(len(means)):
        samples = multivariate_normal.rvs(mean=means[i], cov=covariances[i],
    size=int(n_samples * priors[i]))
        labels = np.full(samples.shape[0], i)
        X.append(samples)
        y.append(labels)
    return np.vstack(X), np.hstack(y)

# Generate a test dataset of 100,000 samples
X_test, y_test = generate_data(100000)


def theoretical_map_classifier(X_test):
    predictions = []
    for x in X_test:
        # Calculate posterior probabilities for each class
        posteriors = []
        for i in range(len(means)):
            likelihood = multivariate_normal.pdf(x, mean=means[i], cov=covariances[i])
            posterior = likelihood * priors[i]  # P(x|C_i) * P(C_i)
            posteriors.append(posterior)

        ## Normalize to get proper posterior probabilities
        posteriors = np.array(posteriors)
        posteriors /= posteriors.sum()  # Normalize to sum to 1

        # Classify as the class with the highest posterior probability
        predictions.append(np.argmax(posteriors))
    return np.array(predictions)

```

```
# Evaluate the theoretical classifier on the test set
def evaluate_theoretical_classifier(X_test, y_test):
    predictions = theoretical_map_classifier(X_test)
    error_count = np.sum(predictions != y_test)
    error_prob = error_count / len(y_test)
    return error_prob

# Evaluate the theoretically optimal classifier
optimal_error_prob = evaluate_theoretical_classifier(X_test, y_test)
print(f"Theoretical classifier error probability: {optimal_error_prob:.4f}")

# Define a two-layer MLP model
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, activation_func):
        super(MLP, self).__init__()
        self.hidden = nn.Linear(input_dim, hidden_dim)
        self.output = nn.Linear(hidden_dim, output_dim)
        self.activation = activation_func
        self.softmax = nn.Softmax(dim=1)

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

# Set activation function
activation_func = nn.ELU()  # Changeable activation function

# Cross-validation to select the optimal number of hidden units
def cross_validate(train_data, train_labels, hidden_sizes, k=10):
    best_hidden_size = None
    best_accuracy = 0

    for hidden_dim in hidden_sizes:
        kf = KFold(n_splits=k)
        accuracies = []

        for train_index, val_index in kf.split(train_data):
            X_train, X_val = train_data[train_index], train_data[val_index]
            y_train, y_val = train_labels[train_index], train_labels[val_index]

            model = MLP(input_dim=3, hidden_dim=hidden_dim, output_dim=len(priors),
    activation_func=activation_func)
            criterion = nn.CrossEntropyLoss()
            optimizer = optim.Adam(model.parameters(), lr=0.01)

            for epoch in range(150):  # Limited epochs for cross-validation
                optimizer.zero_grad()
                outputs = model(torch.tensor(X_train, dtype=torch.float32))
                loss = criterion(outputs, torch.tensor(y_train, dtype=torch.long))
                loss.backward()
                optimizer.step()

            with torch.no_grad():
                val_outputs = model(torch.tensor(X_val, dtype=torch.float32))
                _, predicted = torch.max(val_outputs, 1)
                accuracies.append(accuracy_score(y_val, predicted.numpy()))

        avg_accuracy = np.mean(accuracies)
        if avg_accuracy > best_accuracy:
            best_accuracy = avg_accuracy
            best_hidden_size = hidden_dim

    return best_hidden_size

# Training and evaluation of MLP models for different training dataset sizes
train_sizes = [100, 500, 1000, 5000, 10000]
hidden_sizes = [5, 8, 10, 12, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80]  # Expanded
    hidden layer sizes
error_probabilities = []
best_hidden_sizes = []
figSamples = plt.figure(figsize=(10, 6))

for i, n_samples in enumerate(train_sizes):
    X, y = generate_data(n_samples)

    ax = figSamples.add_subplot(2, 3, i + 1, projection='3d')  # 2 rows and 3 columns layout
    for class_id in range(num_classes):
        class_data = X[y == class_id]
        ax.scatter(class_data[:, 0], class_data[:, 1], class_data[:, 2],
    label=f'L{class_id}', s=20, alpha=0.6)

    ax.set_title(f'd{n_samples}', fontsize=12, fontweight='bold')
    ax.set_xlabel('x1', fontsize=10)
    ax.set_ylabel('x2', fontsize=10)
    ax.set_zlabel('x3', fontsize=10)
    ax.set_xlim(-4, 6)
    ax.set_ylim(-4, 6)
    ax.set_zlim(-4, 6)
    ax.grid(True, linestyle='--', linewidth=0.5)

    # Find the optimal hidden layer size through cross-validation
    best_hidden_size = cross_validate(X, y, hidden_sizes)
    best_hidden_sizes.append(best_hidden_size)

    # Train the final MLP model with the optimal hidden size
    model = MLP(input_dim=3, hidden_dim=best_hidden_size, output_dim=len(priors),
    activation_func=activation_func)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.01)

    for epoch in range(150):  # Full training with more epochs
        optimizer.zero_grad()
        outputs = model(torch.tensor(X, dtype=torch.float32))
        loss = criterion(outputs, torch.tensor(y, dtype=torch.long))
        loss.backward()
        optimizer.step()

    # Evaluate the trained MLP model on the test set
    with torch.no_grad():
        test_outputs = model(torch.tensor(X_test, dtype=torch.float32))
        _, predicted = torch.max(test_outputs, 1)
        accuracy = accuracy_score(y_test, predicted.numpy())
        error_probabilities.append(1 - accuracy)

# Add a legend and show the plot
# Add a single legend outside the subplots
handles, labels = ax.get_legend_handles_labels()
figSamples.legend(handles[:num_classes], labels[:num_classes], loc='upper right',
    title="Classes")
plt.suptitle("Training Datasets", fontsize=16, fontweight='bold')
plt.tight_layout()
plt.subplots_adjust(top=0.88, right=0.85, hspace=0.2)  # Reduced hspace for tighter rows
plt.show()

# Plot 1: Test Error Probability with Theoretical Optimal
plt.figure(figsize=(8, 6))
plt.semilogx(train_sizes, error_probabilities, marker='o', markersize=8, label='MLP Test
    Error Probability', color='blue')
plt.axhline(y=optimal_error_prob, color='red', linestyle='--', label='Theoretical Optimal')
plt.xlabel('Number of Training Samples')
plt.ylabel('Test Probability of Error')
plt.title('Test Probability of Error vs Number of Training Samples')
plt.grid(True, which="both", linestyle='--', linewidth=0.5)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()

# Plot 2: Best Hidden Sizes for Each Training Size
plt.figure(figsize=(8, 6))
plt.semilogx(train_sizes, best_hidden_sizes, linestyle='', marker='o', markersize=8,
    color='green')  # Only markers
plt.xlabel('Number of Training Samples')
plt.ylabel('Best Hidden Sizes')
plt.title('Best Hidden Sizes vs Number of Training Samples')
plt.grid(True, which="both", linestyle='--', linewidth=0.5)
plt.tight_layout()
plt.show()
```

Listing 2: Question 2

```
import numpy as np
```

```
from sklearn.mixture import GaussianMixture
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt
import pandas as pd

# Set random seed for reproducibility
# np.random.seed(0)


# Define the true GMM parameters with overlapping components
# Step 1: Specify initial covariance matrices
covariances = [
    np.array([[1, 0.8], [0.8, 1]]),
    np.array([[1, -0.6], [-0.6, 1]]),
    np.array([[1, 0.5], [0.5, 1]]),
    np.array([[1, -0.7], [-0.7, 1]])
]

# Step 2: Calculate eigenvalues of each covariance matrix
eigenvalues = [np.linalg.eigvalsh(cov) for cov in covariances]
average_eigenvalues = [np.mean(eig) for eig in eigenvalues]

# Step 3: Define means with overlap
# For overlap, we set the distance between the means of two components (e.g., 0 and 1) to
    the sum of their average eigenvalues
overlap_distance = average_eigenvalues[0] + average_eigenvalues[1]
print(overlap_distance)
means = [
    np.array([0, 0]),  # Mean for component 0
    np.array([overlap_distance, overlap_distance]),  # Mean for component 1, placed at a
    distance for overlap
    np.array([6, 0]),  # Mean for component 2, further away
    np.array([3, -3])  # Mean for component 3, further away
]

# Step 4: Define weights for each component to ensure variation in data distribution
weights = [0.2, 0.3, 0.4, 0.1]

# Data generation function
def generate_data(n_samples):
    data = []
    labels = []
    for _ in range(n_samples):
        component = np.random.choice(len(true_means), p=true_weights)
        sample = np.random.multivariate_normal(true_means[component],
    true_covariances[component])
        data.append(sample)
        labels.append(component)
    return np.array(data), np.array(labels)

# Experiment parameters
dataset_sizes = [10, 100, 1000]
num_repeats = 100
num_folds = 10
max_components = 9

# Store selection rates for each dataset size
selection_rates = {size: np.zeros(max_components) for size in dataset_sizes}

# Main experiment loop
for size in dataset_sizes:
    print(f"Processing dataset size: {size}")
    for _ in range(num_repeats):
        data, _ = generate_data(size)

        best_order_counts = np.zeros(max_components)
        kf = KFold(n_splits=num_folds)

        for n_components in range(1, max_components + 1):
            fold_log_likelihoods = []

            for train_index, val_index in kf.split(data):
                X_train, X_val = data[train_index], data[val_index]

                # Fit GMM on the training set
                # gmm = GaussianMixture(n_components=n_components, covariance_type='full',
    random_state=0)
```

```
                gmm = GaussianMixture(n_components=n_components, covariance_type='full')

                gmm.fit(X_train)

                # Compute log-likelihood on the validation set
                log_likelihood = gmm.score(X_val) * len(X_val)
                fold_log_likelihoods.append(log_likelihood)

            # Average log-likelihood over folds
            avg_log_likelihood = np.mean(fold_log_likelihoods)

            # Track the best model order for this repeat
            if avg_log_likelihood > best_order_counts[n_components - 1] or
    best_order_counts[n_components - 1] == 0:
                best_order_counts[n_components - 1] = avg_log_likelihood

        # Select the model order with the highest log-likelihood
        selected_order = np.argmax(best_order_counts) + 1
        selection_rates[size][selected_order - 1] += 1

# Normalize selection rates
selection_rates_normalized = {size: rates / num_repeats for size, rates in
    selection_rates.items()}

# Convert results to a DataFrame for better readability
df_results = pd.DataFrame(selection_rates_normalized).T
df_results.columns = [f"{n_components} components" for n_components in range(1,
    max_components + 1)]

# Display results as a table
print("Selection rates for each dataset size:")
print(df_results)

# Plot selection rates
plt.figure(figsize=(12, 6))
for size, rates in selection_rates_normalized.items():
    plt.plot(range(1, max_components + 1), rates, marker='o', label=f"Dataset size: {size}")
plt.xlabel("Number of Components")
plt.ylabel("Selection Rate")
plt.title("Selection Rate of GMM Components for Different Dataset Sizes")
plt.legend()
plt.grid()
plt.show()
```