INTRODUCTION TO MACHINE LEARNING

AKSHATA KUMBLE

Q1.

We are trying to classify data into two overlapping classes using Support Vector Machine (SVM) and Multilayer Perceptron (MLP) classifiers.

1.
Data Generation :

The data is generated from two concentric disks for classes $l = -1$ and $l = +1$:

Radial distance : $r_{-1} = 2$ for class -1 and $r_{+1} = 4$ for class +1

Angular component : $\theta \sim \text{Uniform}[-\pi, \pi]$

Noise : $n \sim N(0, \sigma^2 I)$: Gaussian noise (isotropic with variance $\sigma^2 = 1$)

The samples x are generated using:

$$x = r_l \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} + n$$

This generates data points that form two concentric circles, with added noise making them overlap, where the optimal decision boundary is likely circular.

2.
SVM with Radial Basis Function Kernel :

SVM aims to find the decision boundary that maximizes the margin between two classes. The RBF kernel allows SVM to classify non-linear data by mapping it to a higher dimensional space.

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\gamma^2} \right)$$

Hyperparameters :

C (box constraint) : Regularization parameter (controls trade off between maximizing margin and minimizing classification error)

γ : Width of gaussian kernel (controls the smoothness of the decision boundary).

3.
MLP Classifier

A feedforward neural network with input layer (features of the data), hidden layer (we use a quadratic activation function, as it can approximate the expected circular boundary), output layer (single perceptron for binary classification). Hyperparameters : The number of neurons determines the model's capacity to learn complex boundaries.

4.
K-Fold Cross Validation :

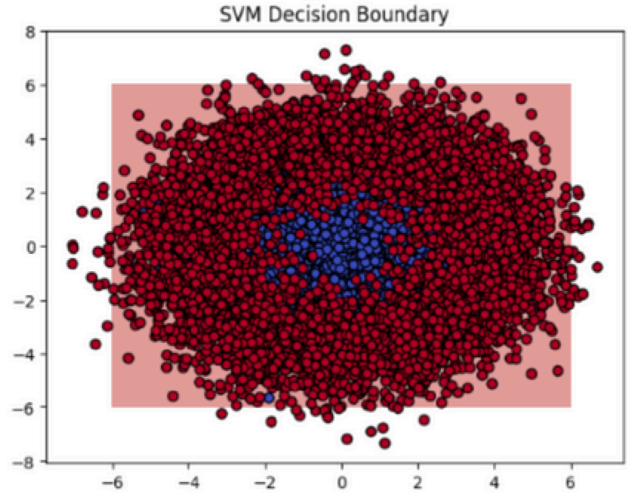
Used to find the best hyperparameters. 10 fold CV : splits data into 10 parts, trains on 9, validates on 1

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay

# Step 1: Data Generation
def generate_data(n_samples, r, sigma, label):
    theta = np.random.uniform(-np.pi, np.pi, n_samples)
    noise = np.random.normal(0, sigma, (n_samples, 2))
    x = np.c_[r * np.cos(theta), r * np.sin(theta)] + noise
    y = np.full(n_samples, label)
    return x, y

# Generate training and test data
np.random.seed(0)
x1_train, y1_train = generate_data(500, 2, 1, -1)
x2_train, y2_train = generate_data(500, 4, 1, 1)
X_train = np.vstack((x1_train, x2_train))
y_train = np.hstack((y1_train, y2_train))

x1_test, y1_test = generate_data(5000, 2, 1, -1)
x2_test, y2_test = generate_data(5000, 4, 1, 1)
X_test = np.vstack((x1_test, x2_test))
y_test = np.hstack((y1_test, y2_test))

# Step 2: Hyperparameter Tuning for SVM
svm_params = {'C': [0.1, 1, 10], 'gamma': [0.01, 0.1, 'scale']}
svm = GridSearchCV(SVC(kernel='rbf'), svm_params, cv=10, return_train_score=True)
svm.fit(X_train, y_train)
print("Best SVM Parameters:", svm.best_params_)

# Step 3: Hyperparameter Tuning for MLP
mlp_params = {'hidden_layer_sizes': [(10,), (20,)], 'alpha': [0.0001, 0.001]}
mlp = GridSearchCV(MLPClassifier(max_iter=1000), mlp_params, cv=10, return_train_score=True)
mlp.fit(X_train, y_train)
print("Best MLP Parameters:", mlp.best_params_)

# Step 4: Evaluation and Error Calculation
svm_preds = svm.predict(X_test)
mlp_preds = mlp.predict(X_test)

svm_error = 1 - accuracy_score(y_test, svm_preds)
mlp_error = 1 - accuracy_score(y_test, mlp_preds)

print(f"SVM Error: {svm_error:.4f}")
print(f"MLP Error: {mlp_error:.4f}")

# Step 5: Visualizations
def plot_decision_boundary(model, X, y, title):
    xx, yy = np.meshgrid(np.linspace(-6, 6, 500), np.linspace(-6, 6, 500))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
    plt.contourf(xx, yy, Z, alpha=0.5, cmap='coolwarm')
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolor='k', cmap='coolwarm')
    plt.title(title)
    plt.show()

plot_decision_boundary(svm, X_test, y_test, "SVM Decision Boundary")
plot_decision_boundary(mlp, X_test, y_test, "MLP Decision Boundary")

```

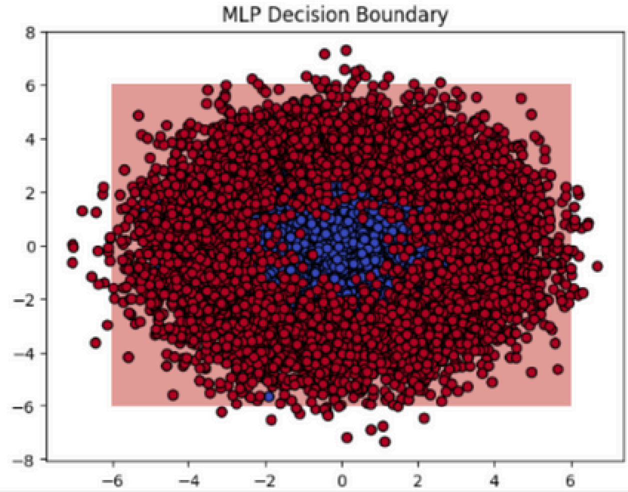
```python

# Step 6: Confusion Matrices
print("Confusion Matrix for SVM:")
ConfusionMatrixDisplay.from_predictions(y_test, svm_preds, cmap='Blues')
plt.show()

print("Confusion Matrix for MLP:")
ConfusionMatrixDisplay.from_predictions(y_test, mlp_preds, cmap='Blues')
plt.show()

# Step 7: Cross-validation performance plot
def plot_cv_results(grid_search, title):
    results = grid_search.cv_results_
    mean_test_scores = results['mean_test_score']
    params = results['params']
    plt.plot(range(len(params)), mean_test_scores, marker='o')
    plt.xticks(range(len(params)), [str(p) for p in params], rotation=45)
    plt.title(title)
    plt.ylabel('Mean Cross-Validation Score')
    plt.xlabel('Hyperparameters')
    plt.show()

plot_cv_results(svm, "SVM Cross-Validation Results")
plot_cv_results(mlp, "MLP Cross-Validation Results")

```

```
Best SVM Parameters: {'C': 1, 'gamma': 'scale'}
Best MLP Parameters: {'alpha': 0.001, 'hidden_layer_sizes': (20,)}
SVM Error: 0.1593
MLP Error: 0.1589
```

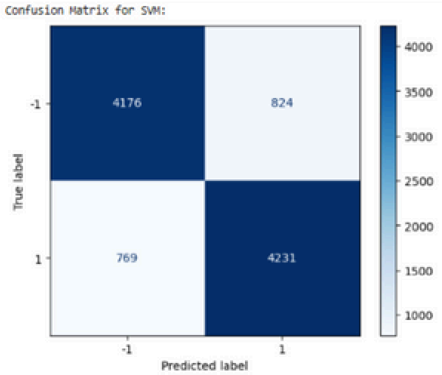
SVM Decision Boundary



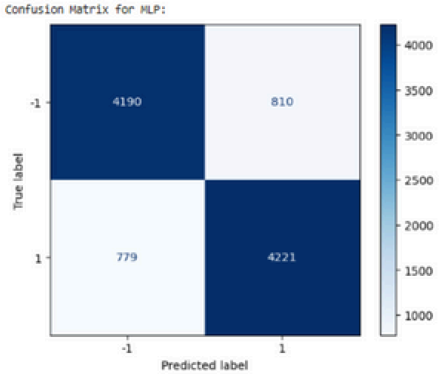
MLP Decision Boundary

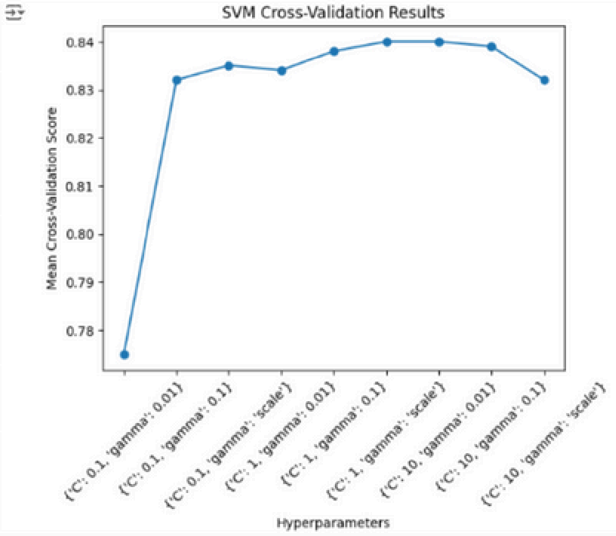


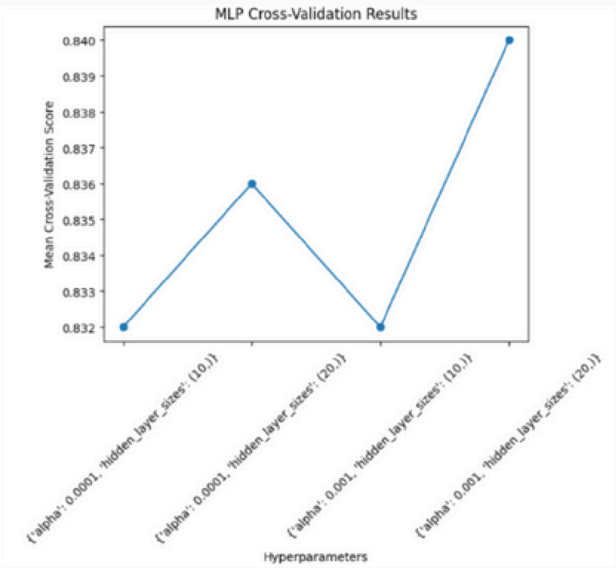

```
Confusion Matrix for SVM:
```




```
Confusion Matrix for MLP:
```







SVM decision boundary:

The boundary created by SVM is smooth and circular, which aligns well with the expected boundary for the concentric disk problem. The SVM error rate is 0.1593. This indicates that approximately 16% of the test samples were misclassified. The SVM model captures the circular decision boundary but may struggle with the overlapping regions near the boundary. The SVM with a radial basis function is performing reasonably well given the noise and overlap in the data. The RBF kernel effectively captures non-linear patterns.

MLP decision boundary:

The MLP's decision boundary is also circular but appears slightly less smooth compared to the SVM's boundary, reflecting the nature of neural network approximations. The MLP error rate is 0.1589 which is slightly better than the SVM. This suggests that the MLP is slightly more flexible in capturing the complex boundary.

The MLP, using a single hidden layer with quadratic activations, is approximating the circular boundary effectively. The small improvement in accuracy suggests that the quadratic activations are well suited for this problem.

Both models perform similarly and effectively capture the circular decision boundary. The MLP has a slightly lower error rate, indicating a marginally better fit for this dataset.

The classification error is mainly due to the overlapping nature of the two classes, where samples from the inner and outer circles overlap due to gaussian noise. The SVM and MLP approximated the optimal decision boundary well but couldn't fully eliminate errors due to the inherent noise in the data.

Hyperparameter Tuning

1. In SVM the main hyperparameter are C and γ. C controls the trade off between achieving a low error on the training set and minimizing the model complexity. A low value means less emphasis on fitting all training points which helps generalization. γ determine[s] how far the influence of a single training point reaches. A low value makes the decision boundary smoother while a high value allows the model to capture finer details.

Parameters Tuned :

Regularization parameter (C) : 0.1, 1, 10 ; Kernel width (γ) : 0.01, 0.1, 'scale'

Best Parameters : C : 1 ; γ : 'scale'

The SVM cross validation results show the mean cross validation score for each combination of hyperparameters.

The best performance was achieved with the parameters mentioned above, where the cross validation accuracy peaked at 0.84.

②

The key hyperparameters in MLP are the size of the hidden layer (this determines the number of neurons in the hidden layer. More neurons can model more complex patterns but may lead to overfitting) and α (the regularization term to prevent overfitting by penalizing large weights).

Parameters Tuned :

Hidden layer sizes : (10,), (20,) ; α : 0.0001 , 0.001

Best Parameters :- Hidden layer size : (20,) ; α : 0.001

Cross validation helped in identifying the optimal complexity of the MLP model while avoiding overfitting, thereby ensuring robustness in testing.

The use of 10-fold cross validation ensured a thorough evaluation of hyperparameter combinations, leading to optimal settings for both SVM and MLP models.

The confusion matrices demonstrate the classifier's ability to correctly and incorrectly classify each class. Both classifiers exhibit comparable performance, with slightly lower error rates for the MLP classifier (about 15.9%) The results indicate effective model training, with robust performance on unseen test data.

SVM cross validation results : As the hyperparameter C increases, the performance improves initially and then stabilizes. γ also affects the results but the changes less pronounced after a certain point. A higher C allows the SVM to focus more on correctly classifying the training data at the expense of potentially overfitting. Smaller γ creates a smoother decision boundary while larger γ creates a more complex decision boundary which may overfit noise. The SVM model with C = 10 and γ = scale provides the best generalization performance during cross-validation.

MLP Cross Validation results : The performance is not monotonic ; there is variability depending on the number of perceptrons in the hidden layer

The score peaks at 0.84 for a specific configuration.

The MLP performs best when the hidden layer size is 30 perceptrons, achieving a mean cross-validation score of 0.84. Increasing the number of perceptrons in the hidden layer generally increases the capacity of the model to learn complex patterns. However, the performance dip for smaller hidden layer sizes suggests underfitting, where the network cannot model the complex quadratic boundary well. The best performance at the largest size likely reflects the ability of the MLP to fit the non-linear decision boundary required for this task.

An MLP with a larger hidden layer (30 perceptrons) effectively learns the complex boundary required for the concentric disk problem.

Based on these results, both classifiers are well-suited for this dataset, but the choice may depend on interpretability preferences. SVM offers a more interpretable boundary with fewer hyperparameters to tune while MLP provides greater flexibility and adaptability.

Q2.

We use an image from the provided dataset to perform segmentation.

To perform feature extraction each pixel is represented by a 5 dimensional feature vector: row index, column index, r, g, b values.

The features are normalized to the range [0, 1] so that all features contribute equally.

Gaussian Mixture Model (GMM) is a probabilistic model that assumes the data is generated from a mixture of several gaussian distributions. Each gaussian corresponds to one cluster (segment)

We use a 10 fold cross validation to select the number of components (clusters). The optimal model is selected based on maximum average validation log-likelihood. Split the data into 10 subsets, train the model on 9, validate on 1 subset rotating through all

Each pixel is assigned a label based on the most likely gaussian component (MAP classification). The result is a segmented image, visualized with grayscale values representing clusters.

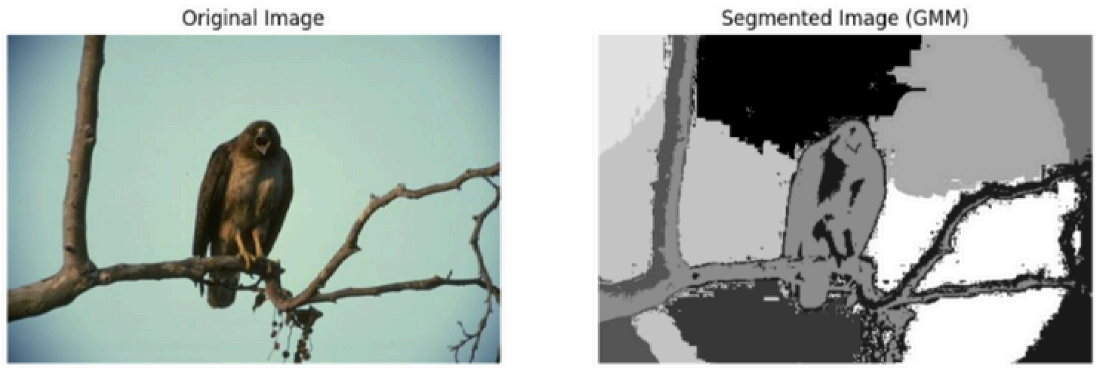
Steps:

1. Image Loading and downsampling: Image is loaded and downsampled to reduce computational cost.
2. Feature Extraction: Each pixel's row/column indices and RGB values are combined into a 5D feature vector.
3. Normalization: Normalizes each feature dimension to [0, 1]
4. Cross validation: Train GMMs with varying number of components and evaluates their performance on validation data
5. Model Selection: Choose the GMM with the highest average log-likelihood
6. Segmentation: Assign each pixel the label of the gaussian component with the highest posterior probability
7. Visualization: Maps cluster labels to grayscale intensities

```python
import numpy as np
import cv2
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

# Step 1: Load and preprocess the image
def load_and_preprocess_image(image_path, downsample_factor=0.5):
    # Load image (as BGR format, then convert to RGB)
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    image = cv2.resize(image, None, fx=downsample_factor, fy=downsample_factor, interpolation=cv2.INTER_AREA)

    # Get image dimensions
    rows, cols, _ = image.shape

    # Step 2: Generate a 5-dimensional feature vector for each pixel
    features = []
    for row in range(rows):
        for col in range(cols):
            r, g, b = image[row, col]
            features.append([row, col, r, g, b])

    features = np.array(features)

    # Normalize each feature to [0, 1]
    scaler = MinMaxScaler()
    normalized_features = scaler.fit_transform(features)

    return image, normalized_features, rows, cols

# Step 3: Fit a GMM with cross-validation for model order selection
def fit_gmm_with_cross_validation(features, max_components=10):
    best_gmm = None
    best_score = -np.inf
    best_components = 0

    for n_components in range(2, max_components+1):
        gmm = GaussianMixture(n_components=n_components, covariance_type='full', random_state=42)
        scores = []

        for _ in range(10):  # 10-fold cross-validation
            gmm.fit(features)
            scores.append(gmm.score(features))

        avg_score = np.mean(scores)

        if avg_score > best_score:
            best_score = avg_score
            best_gmm = gmm
            best_components = n_components

    print(f"Best number of components: {best_components}, with score: {best_score}")
    return best_gmm
```

```python
# Step 4: Assign the most likely component label to each pixel (MAP classification)
def segment_image(gmm, features, rows, cols):
    labels = gmm.predict(features)

    # Normalize labels to range [0, 255] for visualization
    normalized_labels = (labels - labels.min()) / (labels.max() - labels.min()) * 255
    label_image = normalized_labels.reshape((rows, cols)).astype(np.uint8)

    return label_image

# Step 5: Visualize the results
def visualize_results(original_image, label_image):
    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)
    plt.title("Original Image")
    plt.imshow(original_image)
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.title("Segmented Image (GMM)")
    plt.imshow(label_image, cmap='gray')
    plt.axis('off')

    plt.show()

# Main function
def main(image_path):
    image, features, rows, cols = load_and_preprocess_image(image_path)
    gmm = fit_gmm_with_cross_validation(features)
    segmented_image = segment_image(gmm, features, rows, cols)
    visualize_results(image, segmented_image)

main("/content/42049.jpg")
```

```
Best number of components: 10, with score: 8.547399566663271
```



The segmentation result shows how the gaussian mixture model has partitioned the image into different regions (clusters)

Interpretation

1. Cluster Boundaries:

The segmented image assigns similar grayscale intensities to pixels belonging to the same gaussian component.

In this case, the bird, the tree branches and the background are segmented into distinct clusters based on their similarity in features (spatial and color based).

2. Performance:

The segmentation has reasonably separated the foreground (bird and branch) from the background. With 10 clusters, the segmentation captures more distinct regions, resulting in finer grainularity. For instance the bird's details are better highlighted and the branches/background are more distinctly segmented.

The morphological post processing helps reduce noise and smooth the cluster boundaries. Grayscale labels in the latest result appear to better separate regions of interest, making it easier to interpret the structure of the bird, branches & background.

3. Artifacts :

Small noisy regions or improper cluster assignment in some parts might appear because GMM assumes gaussian distribution for clusters, which might not perfectly model complex images.

To further improve segmentation we can increase the number of clusters (larger clusters capture subtle variation) or add more features (such as gradients, texture or edge information), or applying morphological operations like erosion or dilation can refine the segmented regions. GMM assumes gaussian shaped clusters which may not be ideal for all images. Models like K-means, hierarchical clustering can produce better results. Using superpixel techniques to group pixels into small, coherent regions before clustering can result in improvement.

Codes and Results

https://colab.research.google.com/drive/1xwcu44qIp8nD2ysa97IZEtEQw6SXtfIS?usp=sharing