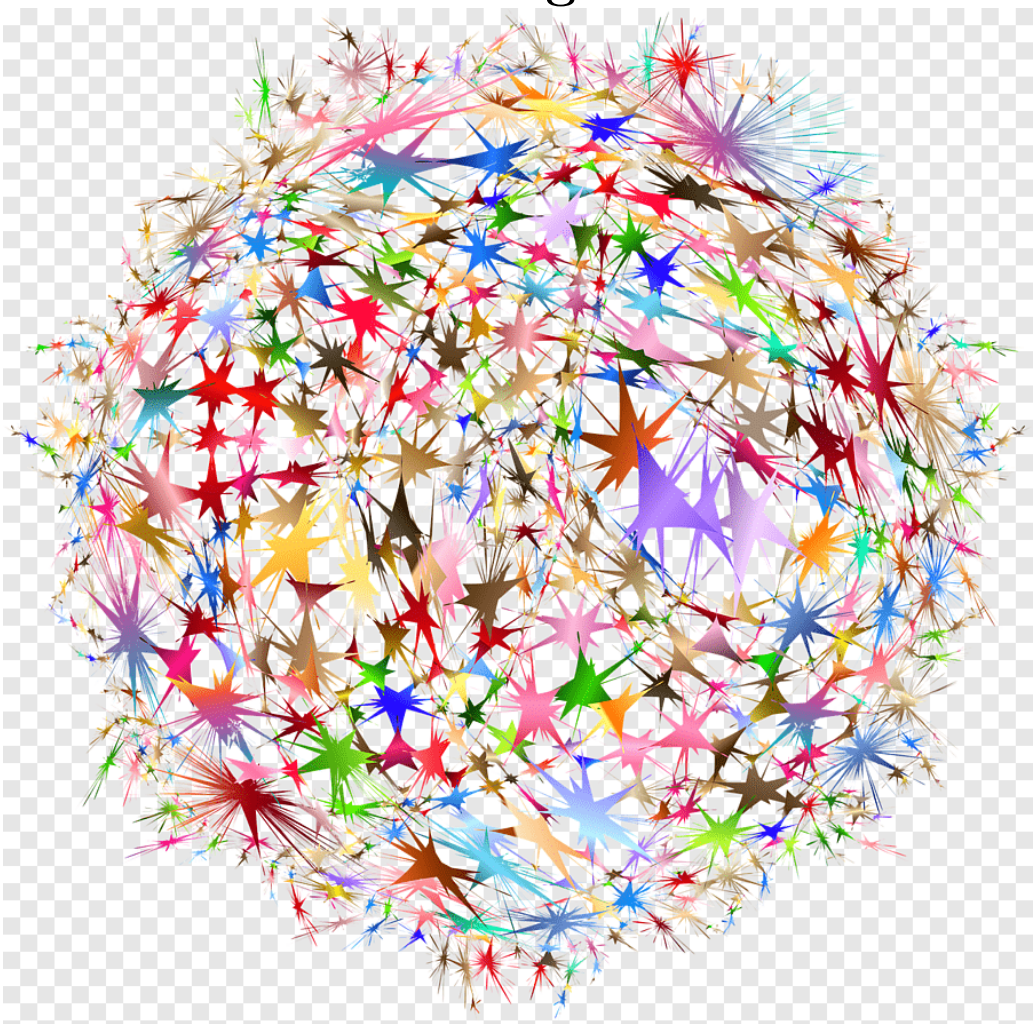
Northeastern University

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: 50%

Train and test **Support Vector Machine (SVM)** and **Multi-layer Perceptron (MLP)** classifiers that aim for minimum probability of classification error (i.e., we are using 0-1 loss; all error instances are equally bad). You may use a trusted implementation of training, validation, and testing in your choice of programming language. The **SVM** should use a *Gaussian* (sometimes called radial-basis) kernel. The **MLP** should be a single-hidden layer model with your choice of activation functions for all perceptrons.

Generate 1000 independent and identically distributed (iid) samples for training and 10000 iid samples for testing. All data for class $l \in \{-1, +1\}$ should be generated as follows:

$$\mathbf{x} = r_l \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} + \mathbf{n} \tag{1}$$

where $\theta \sim Uniform[-\pi, \pi]$ and $\mathbf{n} \sim N(0, \sigma^2\mathbf{I})$. Use $r_{-1} = 2$, $r_{+1} = 4$, $\sigma = 1$.

Note: *The two class sample sets will be highly overlapping two concentric disks, and due to angular symmetry, we anticipate the best classification boundary to be a circle between the two disks. Your SVM and MLP models will try to approximate it. Since the optimal boundary is expected to be a quadratic curve, quadratic polynomial activation functions in the hidden layer of the MLP may be considered as to be an appropriate modeling choice. If you have time (optional, not needed for assignment), experiment with different activation function selections to see the effect of this choice.*

Use the training data with 10-fold cross-validation to determine the best hyperparameters (box constraints parameter and Gaussian kernel width for the SVM, number of perceptrons in the hidden layer for the MLP). Once these hyperparameters are set, train your final SVM and MLP classifier using the entire training data set. Apply your trained SVM and MLP classifiers to the test data set and estimate the probability of error from this data set.

Report the following:

1. Visual and numerical demonstrations of the K-fold cross-validation process indicating how the hyperparameters for SVM and MLP classifiers are set.

2. Visual and numerical demonstrations of the performance of your SVM and MLP classifiers on the test data set.

It is your responsibility to figure out how to present your results in a convincing fashion to indicate the quality of training procedure execution, and the test performance estimate.

Hint: *For hyperparameter selection, you may show the performance estimates for various choices and indicate where the best result is achieved. For test performance, you may show the data and classification boundary superimposed, along with an estimated probability of error from the samples. Modify and supplement these ideas as you see appropriate.*

Answer

Import Function

```
# Enable inline plotting for Google Colab
%matplotlib inline

# Import necessary libraries
import matplotlib.pyplot as plt  # For general plotting
from matplotlib.ticker import MaxNLocator
from math import ceil, floor

import numpy as np

from scipy.stats import multivariate_normal as mvn
from skimage.io import imread
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.mixture import GaussianMixture
from sklearn.model_selection import KFold, GridSearchCV
from sklearn.preprocessing import LabelBinarizer
from sklearn.svm import SVC

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

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

# Set options for numpy to display numbers without scientific notation
np.set_printoptions(suppress=True)

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

# Set global plotting configurations for better visualization
plt.rc('axes', titlesize=20)  # Font size of the Title
plt.rc('font', size=24)  # Font size for text
plt.rc('axes', labelsize=18)  # Font size for the x and y labels
plt.rc('xtick', labelsize=16)  # Font size of the x-axis tick labels
plt.rc('ytick', labelsize=16)  # Font size of the y-axis tick labels
plt.rc('legend', fontsize=14)  # Font size of the legend
plt.rc('figure', titlesize=24)  # Font size of the figure title
```

Utility Functions

```
def plot_binary_classification_results(ax, predictions, labels):
    # Get indices of the four decision scenarios:
    # True Negatives
    tn = np.argwhere((predictions == -1) & (labels == -1))
    # False Positives
    fp = np.argwhere((predictions == 1) & (labels == -1))
    # False Negatives
    fn = np.argwhere((predictions == -1) & (labels == 1))
    # True Positives
    tp = np.argwhere((predictions == 1) & (labels == 1))

    # Plot points based on classification results
    ax.plot(X_test[tn, 0], X_test[tn, 1], 'og', label="Correct Class -1")  # True Negatives
    ax.plot(X_test[fp, 0], X_test[fp, 1], 'or', label="Incorrect Class -1")  # False Positives
    ax.plot(X_test[fn, 0], X_test[fn, 1], '+r', label="Incorrect Class 1")  # False Negatives
    ax.plot(X_test[tp, 0], X_test[tp, 1], '+g', label="Correct Class 1")  # True Positives
```

```python
class TwoLayerMLP(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(TwoLayerMLP, self).__init__()
        # Define the hidden layer
        self.hidden_layer = nn.Linear(input_size, hidden_size)
        # Define the output layer
        self.output_layer = nn.Linear(hidden_size, 1)  # Binary classification: single output node

    def forward(self, x):
        x = torch.relu(self.hidden_layer(x))  # ReLU activation for the hidden layer
        x = self.output_layer(x)  # Output logits (no activation)
        return x
```

- The below-provided function, `generate_multiring_dataset`, generates a two-class dataset for binary classification with a specified number of samples, dimensionality, and parameters governing the distribution.
- The labels are assigned as -1 or $+1$, representing the two classes. The decision on sample allocation to each class is made randomly based on a prior probability. The positive class samples are generated from a uniform distribution with a random angle, while the negative class samples are similarly generated but with an additional random rotation.
- The function utilizes a mixture of uniform and Gaussian distributions, with the parameters such as the mixture ratio, radii, mean, and covariance matrix controlled by the input `pdf_params`. The resulting dataset is returned as a matrix of samples (X) and corresponding labels.

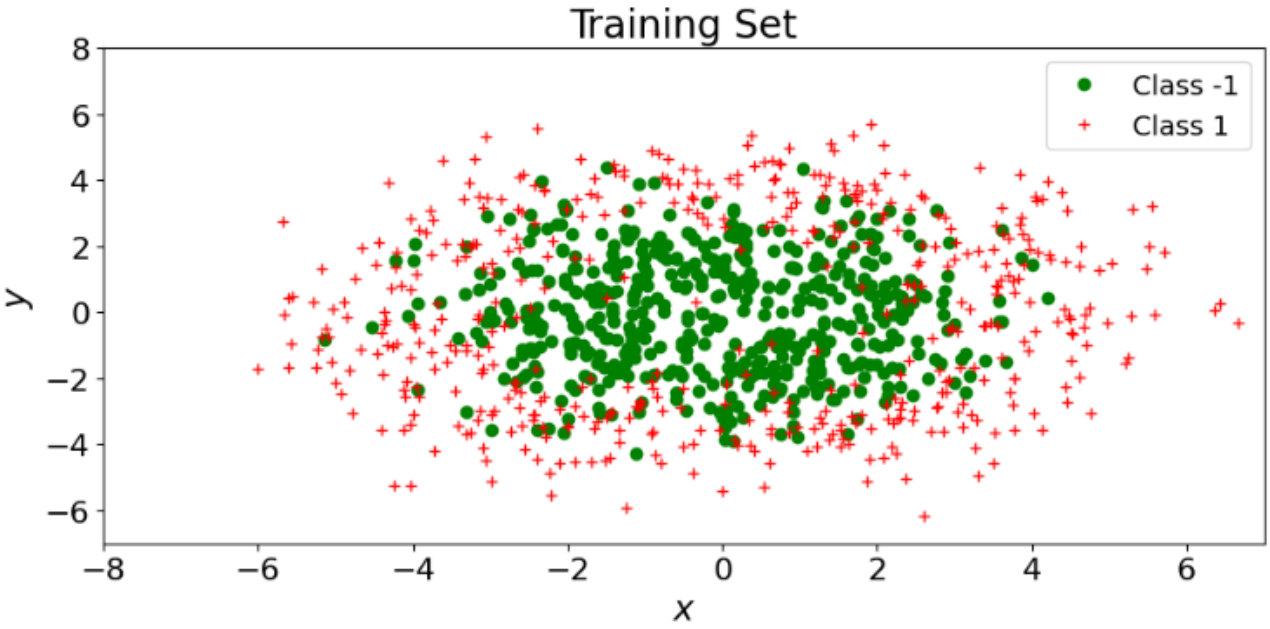
```python
def generate_multiring_dataset(num_samples, n, pdf_params):
    # Output samples and labels
    X = np.zeros((num_samples, n))
    # Note that the labels are either -1 or +1 for binary classification
    labels = np.ones(num_samples)

    # Decide randomly which samples will come from each class
    indices = np.random.rand(num_samples) < pdf_params['prior']
    # Reassign random samples to the negative class values (-1)
    labels[indices] = -1
    num_neg = sum(indices)

    # Create mixture distribution
    theta = np.random.uniform(low=-np.pi, high=np.pi, size=num_samples)
    uniform_component = np.array([np.cos(theta), np.sin(theta)]).T

    # Positive class samples
    X[~indices] = (
        pdf_params['r+'] * uniform_component[~indices] +
        mvn.rvs(pdf_params['mu'], pdf_params['Sigma'], num_samples - num_neg)
    )
    # Negative class samples
    X[indices] = (
        pdf_params['r-'] * uniform_component[indices] +
        mvn.rvs(pdf_params['mu'], pdf_params['Sigma'], num_neg)
    )

    return X, labels
```

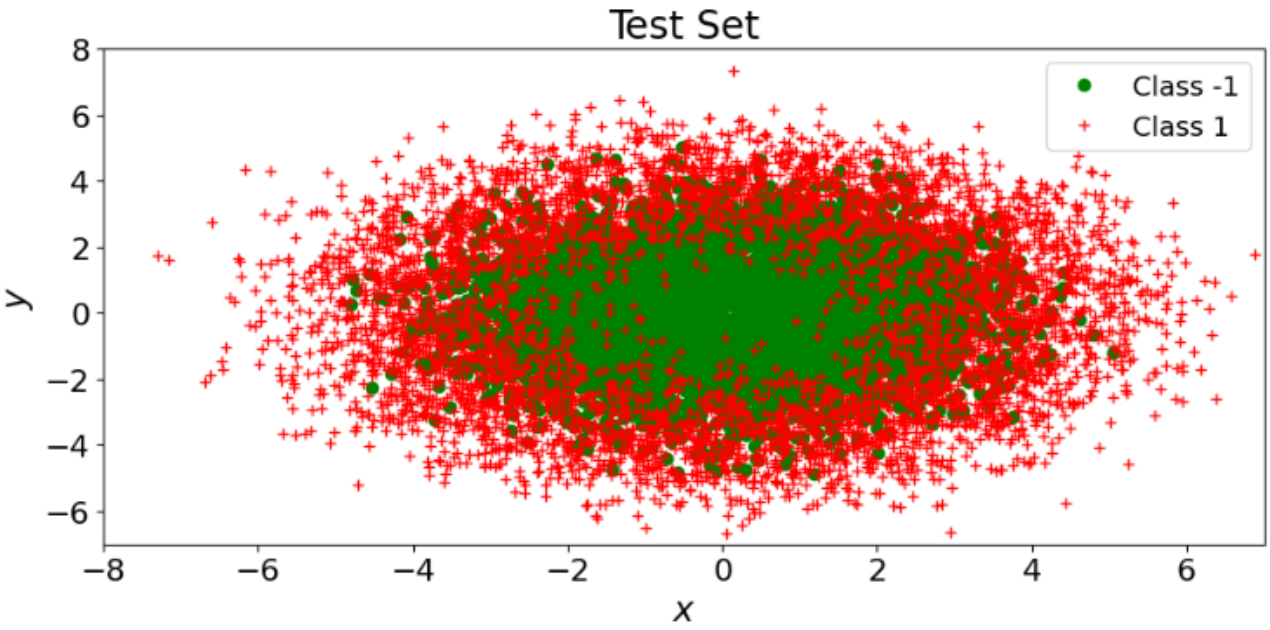
- The provided code defines a function `plot_dataset` for visualizing two-dimensional datasets with labeled classes, where class -1 is represented by green circles (`'go'`) and class 1 by red crosses (`'r+'`).
- The main script generates two datasets, a training set $(X_{\text{train}}, y_{\text{train}})$ and a test set $(X_{\text{test}}, y_{\text{test}})$, using the `generate_multiring_dataset` function with specified parameters. Subsequently, the script creates a 2×1 subplot layout to display the original training and test datasets, utilizing the defined plotting function.
- The resulting plots showcase the distribution of class labels in each set, aiding in visualizing the generated multiring dataset. The axes limits are adjusted based on the test set samples for consistency, providing a clear representation of the data distribution.

```python
def plot_dataset(ax, X, y, title):
    ax.set_title(title)
    ax.plot(X[y == -1, 0], X[y == -1, 1], 'go', label="Class -1")
    ax.plot(X[y == 1, 0], X[y == 1, 1], 'r+', label="Class 1")
    ax.set_xlabel(r"$x$")
    ax.set_ylabel(r"$y$")
    ax.legend()
```

```python
# Two-dimensional data
n = 2
mix_pdf = {
    'r+': 4,  # Radius for positive class
    'r-': 2,  # Radius for negative class
    'prior': 0.5,  # Prior probability for class -1
    'mu': np.zeros(n),  # Mean vector for Gaussian noise
    'Sigma': np.identity(n)  # Covariance matrix for Gaussian noise
}

# Number of training and test set samples
N_train = 1000
N_test = 10000

# Generate training and test sets
X_train, y_train = generate_multiring_dataset(N_train, n, mix_pdf)
X_test, y_test = generate_multiring_dataset(N_test, n, mix_pdf)

# Plot the original data and their true labels
fig, ax = plt.subplots(2, 1, figsize=(10, 10))

plot_dataset(ax[0], X_train, y_train, "Training Set")
plot_dataset(ax[1], X_test, y_test, "Test Set")

# Using test set samples to limit axes
x1_lim = (np.floor(np.min(X_test[:, 0])), np.ceil(np.max(X_test[:, 0])))
x2_lim = (np.floor(np.min(X_test[:, 1])), np.ceil(np.max(X_test[:, 1])))
plt.setp(ax, xlim=x1_lim, ylim=x2_lim)
plt.tight_layout()
plt.show()
```

OUTPUT





SVM

- **Hyperparameter Tuning (Box Constraints and Gaussian Kernel Width):**
  - **Box Constraints (C)**: The box constraints parameter controls the trade-off between achieving a smooth decision boundary and classifying training points correctly. It is optimized using 10-fold cross-validation on the training data. Different values of C are tried, and the one that results in the lowest cross-validation error is selected.
  - **Gaussian Kernel Width (Gamma)**: The Gaussian kernel width determines the influence of a single training example. A small gamma means a larger influence, while a large gamma means a smaller influence. Similar to C, gamma is optimized through 10-fold cross-validation.

- **Training the Final SVM Classifier:**
  - Once the optimal values for C and gamma are determined through cross-validation, the final SVM classifier is trained on the entire training dataset using these hyperparameters.
- **Testing and Probability of Error Estimation:**
  - The trained SVM classifier is then applied to the test dataset, and predictions are made. The probability of error is estimated by comparing the predicted labels with the true labels in the test dataset.
- **Approach:**
  - The SVM optimization problem involves finding the hyperplane that maximizes the margin between classes while minimizing the classification error. The box constraints parameter (C) and the Gaussian kernel width (gamma) influence the regularization term and the shape of the decision boundary, respectively. The optimization problem can be formulated as a convex quadratic programming problem.
  - The optimal hyperparameters are determined by minimizing the cross-validation error, which serves as a proxy for the generalization error.
- **Implementation:**
  - The provided code performs hyperparameter tuning for a Support Vector Machine (SVM) using 10-fold cross-validation. The hyperparameters under consideration are the regularization strength (C) and the Gaussian kernel width (gamma). The dataset is divided into 10 folds, and the SVM model is trained and evaluated multiple times with different combinations of C and gamma using the `GridSearchCV` method. The best hyperparameters are selected based on the combination that minimizes the cross-validation error.
  - The code then visualizes the probability of error against the regularization strength while holding the kernel width constant. This is achieved by plotting the cross-validation probability of error for various values of C, sorted in ascending order, with each line representing a different gamma value. The resulting plot provides insights into the trade-off between regularization strength and model performance, particularly with varying kernel widths. Lower values of regularization strength may result in overfitting, while higher values may lead to underfitting. The visualization aids in selecting an appropriate balance between regularization and flexibility, ultimately contributing to the robustness of the SVM model on unseen data.

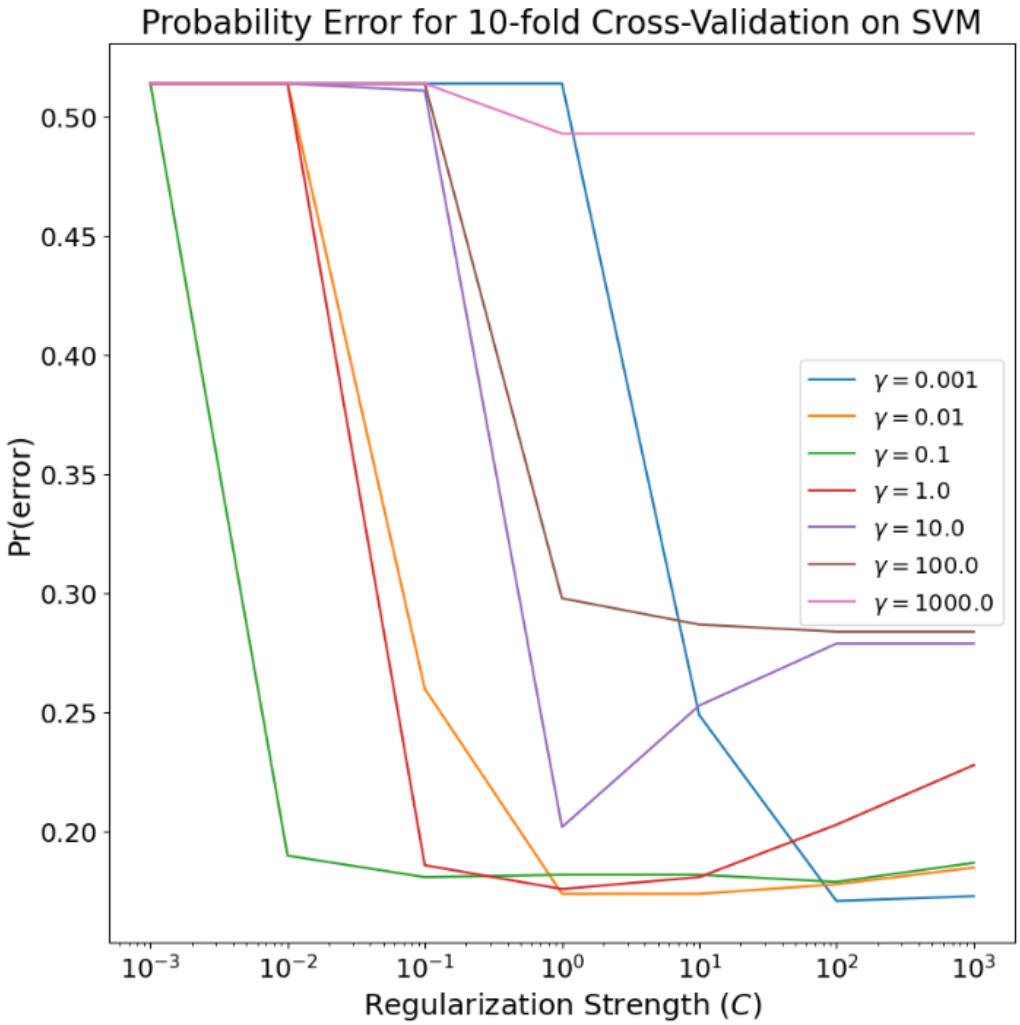
```python
# Set the number of folds for cross-validation
K = 10

# Define parameter ranges for GridSearchCV
regularization_strengths = np.logspace(-3, 3, 7)  # Range for the regularization strength (C)
kernel_widths = np.logspace(-3, 3, 7)  # Range for the kernel width (gamma)
param_grid = {'C': regularization_strengths, 'gamma': kernel_widths}

# Initialize an SVM classifier with Gaussian (RBF) kernel
svm_classifier = SVC(kernel='rbf')

# Define K-fold cross-validator
cross_validator = KFold(n_splits=K, shuffle=True)

# Perform grid search with cross-validation
classifier = GridSearchCV(svm_classifier, param_grid=param_grid, cv=cross_validator)
classifier.fit(X_train, y_train)

# Extract the best parameters
best_regularization_strength = classifier.best_params_['C']
best_kernel_width = classifier.best_params_['gamma']

# Print the best hyperparameters and their performance
print(f"Best Regularization Strength: {best_regularization_strength:.3f}")
print(f"Best Kernel Width: {best_kernel_width:.3f}")
print(f"SVM CV Pr(error): {1 - classifier.best_score_:.3f}")
```

OUTPUT

```
Best Regularization Strength: 100.000
Best Kernel Width: 0.001
SVM CV Pr(error): 0.171
```

```python
# Extract grid search results
regularization_data = classifier.cv_results_['param_C']
gamma_data = classifier.cv_results_['param_gamma']
cv_prob_error = 1 - classifier.cv_results_['mean_test_score']

# Plot probability of error as a function of C and gamma
plt.figure(figsize=(10, 10))

# Loop through each kernel width to plot
for g in kernel_widths:
    # Find indices corresponding to the current gamma value
    gamma_indices = np.where(gamma_data == g)[0]

    # Sort the indices based on regularization strength for plotting
    sort_idx = np.argsort(regularization_data[gamma_indices])
    sorted_regularization = regularization_data[gamma_indices][sort_idx]
    sorted_prob_error = cv_prob_error[gamma_indices][sort_idx]

    # Plot Pr(error) for the current gamma value
    plt.plot(sorted_regularization, sorted_prob_error, label=fr"$\gamma = {g}$")

# Customize the plot
plt.title("Probability Error for 10-fold Cross-Validation on SVM")
plt.xscale('log')  # Logarithmic scale for regularization strength
plt.xlabel(r"Regularization Strength ($C$)")
plt.ylabel("Pr(error)")
plt.legend()
plt.show()
```

OUTPUT



- The presented code trains a Support Vector Machine (SVM) classifier using the previously determined optimal hyperparameters (best regularization strength and kernel width) on the entire training dataset.
- The SVM model is then applied to the test dataset, and predictions are made. The probability of error on the test dataset is calculated by comparing the predicted labels with the true labels, providing an assessment of the model's performance on unseen data.
- Additionally, the code visualizes the binary classification results and the SVM decision boundaries on the test set. The plot showcases the distribution of correct and incorrect predictions, with the decision boundaries delineating regions corresponding to different classes. The shaded areas represent the SVM's classification regions, aiding in the interpretation of the model's decision-making process. This visualization offers valuable insights into the classifier's performance and how it partitions the feature space.

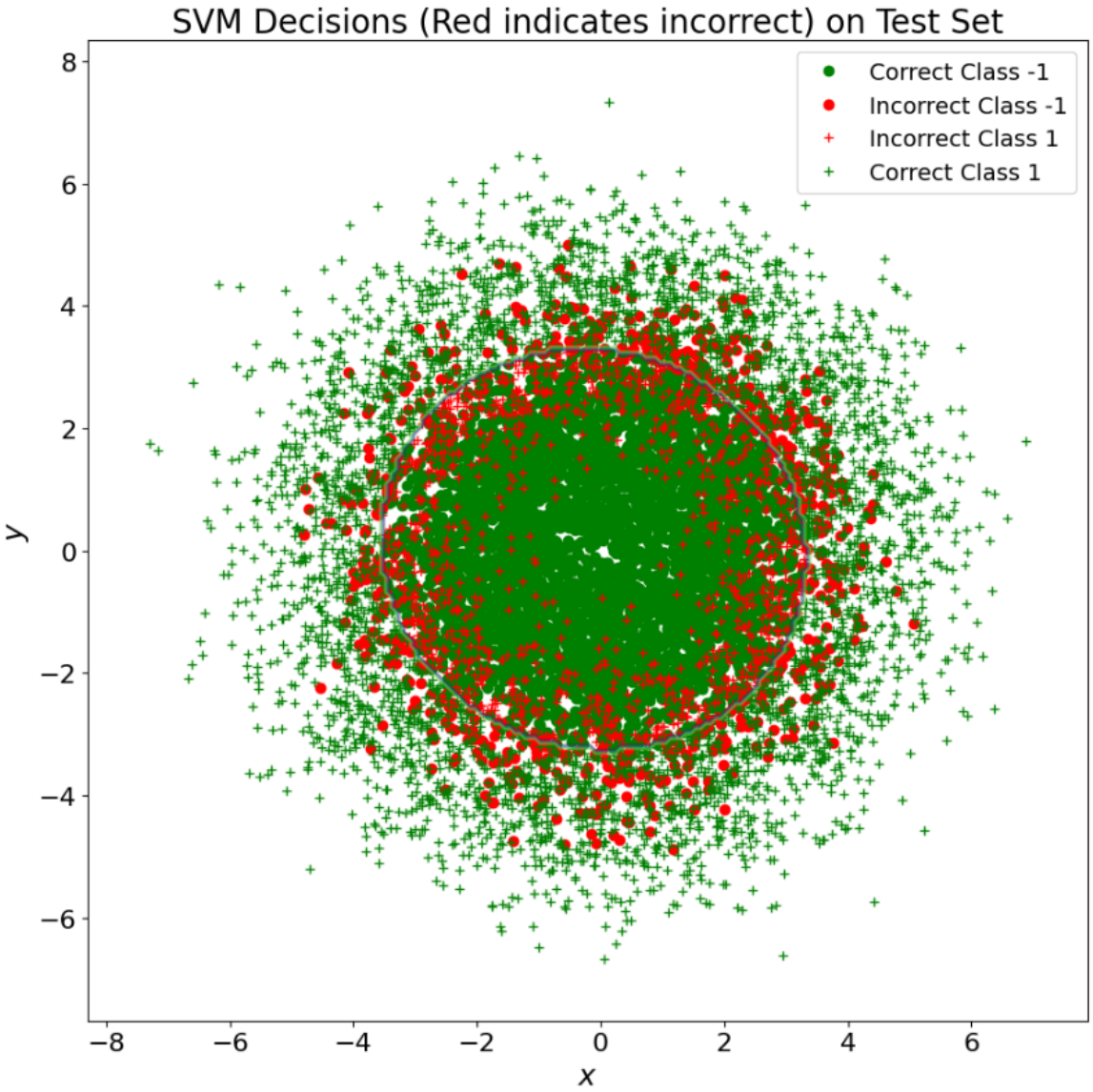
```
# Train SVM using the best parameters on the entire training dataset
svm_classifier = SVC(C=best_regularization_strength, kernel='rbf', gamma=best_kernel_width)
svm_classifier.fit(X_train, y_train)
test_predictions = svm_classifier.predict(X_test)

# Calculate the probability of error on the test dataset
incorrect_indices = np.argwhere(y_test != test_predictions)
test_error_probability = len(incorrect_indices) / N_test
print("Test Set Probability of Error for SVM: %.4f\n" % test_error_probability)
```

OUTPUT

```
Test Set Probability of Error for SVM: 0.1729
```

```
# Plot the binary classification results and SVM decision boundaries on the test set
fig, ax = plt.subplots(figsize=(10, 10))
plot_binary_classification_results(ax, test_predictions, y_test)

# Define the region of interest based on data limits
x_min, x_max = X_test[:, 0].min() - 1, X_test[:, 0].max() + 1
y_min, y_max = X_test[:, 1].min() - 1, X_test[:, 1].max() + 1
x_span = np.linspace(x_min, x_max, num=200)
y_span = np.linspace(y_min, y_max, num=200)
xx, yy = np.meshgrid(x_span, y_span)

# Prepare grid points for decision boundary predictions
grid_points = np.c_[xx.ravel(), yy.ravel()]

# Z matrix represents the SVM classifier predictions for the grid points
decision_boundary = svm_classifier.predict(grid_points).reshape(xx.shape)
ax.contour(xx, yy, decision_boundary, cmap=plt.cm.viridis, alpha=0.25)

# Add labels, title, and legend
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$y$")
ax.set_title("SVM Decisions (Red indicates incorrect) on Test Set")
ax.legend()

# Adjust layout and display plot
plt.tight_layout()
plt.show()
```

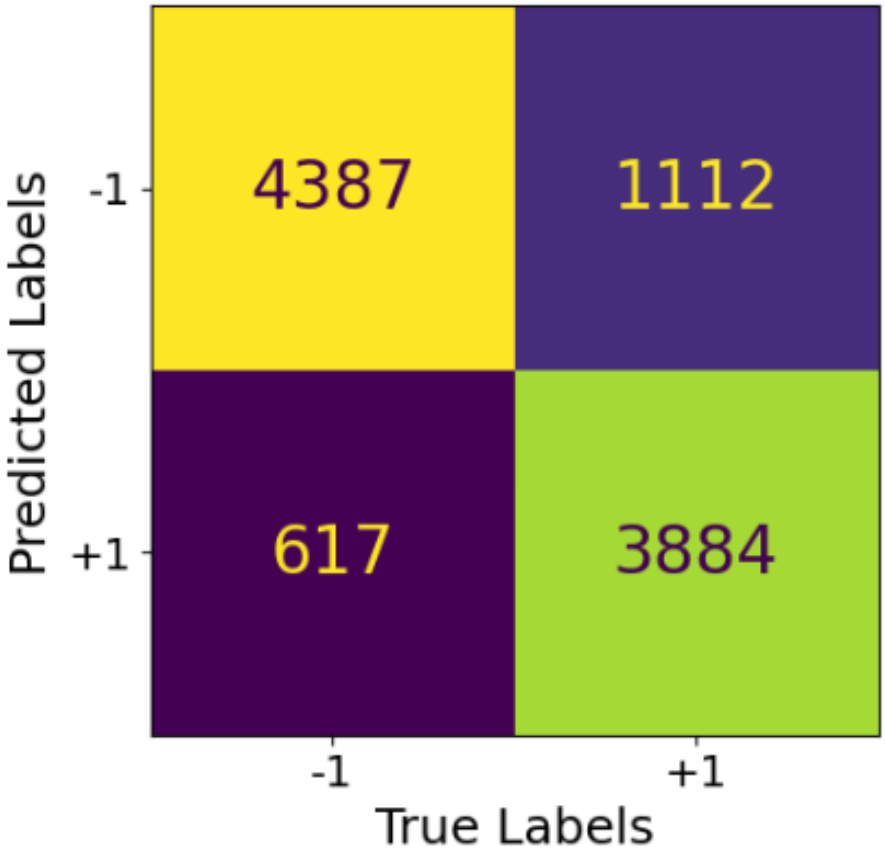
OUTPUT



Confusion Matrix:

```
# Simply using scikit-learn's confusion matrix
print("Confusion Matrix (rows: Predicted class, columns: True class):")
conf_mat = confusion_matrix(test_predictions, y_test)

# Display the confusion matrix
conf_display = ConfusionMatrixDisplay.from_predictions(
    test_predictions, y_test, display_labels=['-1', '+1'], colorbar=False
)

# Add axis labels
plt.ylabel("Predicted Labels")
plt.xlabel("True Labels")
```

OUTPUT



MLP

- **Hyperparameter Tuning (Number of Perceptrons):**
  - The number of perceptrons in the hidden layer is optimized using 10-fold cross-validation on the training data. Different values for the number of perceptrons are considered, and the one resulting in the lowest cross-validation error is selected.
- **Training the Final MLP Classifier:**
  - Once the optimal number of perceptrons is determined, the final MLP classifier is trained on the entire training dataset with this chosen architecture.
- Testing and Probability of Error Estimation:
  - The trained MLP classifier is applied to the test dataset, and predictions are made. The probability of error is estimated by comparing the predicted labels with the true labels in the test dataset.
- **Approach:**
  - The MLP optimization involves finding the weights and biases that minimize the error between the predicted and true labels. The number of perceptrons in the hidden layer influences the capacity of the model to capture complex patterns in the data.
  - The optimal number of perceptrons is determined by minimizing the cross-validation error.

- **Implementation:**
  - The provided code encompasses a k-fold cross-validation framework for training and evaluating Multi-Layer Perceptron (MLP) models with varying numbers of perceptrons in the hidden layer. The function `k_fold_cross_validation` employs k-fold partitioning on the input dataset, iterating over different numbers of perceptrons specified in the `perceptron_options` array. For each configuration, the code trains an MLP model using stochastic gradient descent with momentum as the optimizer.
  - The trained models are then evaluated on their respective validation folds, and the probability of error is calculated by comparing the predicted labels with the true labels. The average probability of error across all folds is computed for each model configuration, and the function returns the number of perceptrons that result in the minimum average cross-validation error.
  - Furthermore, the code generates a plot illustrating the relationship between the number of perceptrons and the average cross-validation probability of error. This visualization aids in understanding how the complexity of the model, controlled by the number of perceptrons, impacts its performance on unseen data. The plot assists in selecting an optimal model architecture by identifying the configuration that minimizes the average cross-validation error, facilitating the training of an MLP model with improved generalization capabilities.

```python
def train_model(model, input_data, target_labels, optimizer, criterion=nn.BCEWithLogitsLoss(), num_epochs=100):
    # Set the model in training mode
    model.train()

    for epoch in range(num_epochs):
        # Forward pass: compute predicted probabilities
        predicted_logits = model(input_data)
        # Calculate the loss between predictions and target labels
        loss = criterion(predicted_logits, target_labels.unsqueeze(1))

        # Clear gradients before backward pass
        optimizer.zero_grad()
        # Backward pass: compute gradients
        loss.backward()
        # Update model parameters using the optimizer
        optimizer.step()

    return model, loss
```

```python
def predict_with_model(model, input_data):
    # Set the model in evaluation mode
    model.eval()

    # Disable gradient calculation for inference
    with torch.no_grad():
        # Forward pass: get predicted logits
        predicted_logits = model(input_data)
        # Apply sigmoid to logits for obtaining output probabilities
        predicted_probs = torch.sigmoid(predicted_logits).detach().numpy()
        # Reshape to remove unnecessary dimensions
        return predicted_probs.reshape(-1)
```

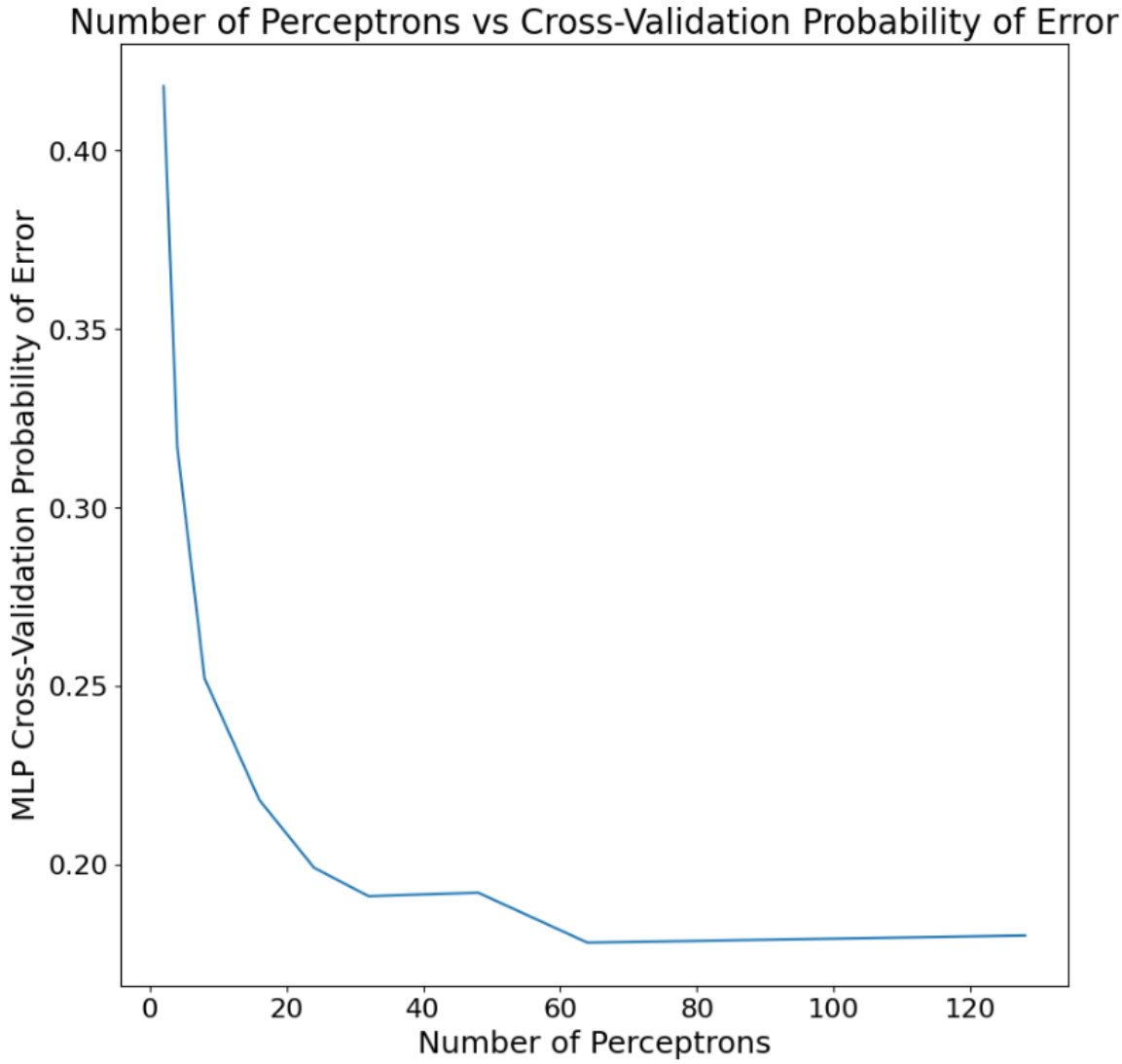
```python
def train_perceptron_model(model, input_data, target_labels, optimizer, num_epochs=100):
    # Set the model in training mode
    model.train()

    # Define the loss function (handles sigmoid internally)
    criterion = nn.BCEWithLogitsLoss()

    for epoch in range(num_epochs):
        # Forward pass: compute logits
        predicted_logits = model(input_data)
        # Compute the loss
        loss = criterion(predicted_logits, target_labels.unsqueeze(1))  # Unsqueeze to match logits shape

        # Clear the gradients before backward pass
        optimizer.zero_grad()
        # Backward pass: compute gradients
        loss.backward()
        # Update model parameters using the optimizer
        optimizer.step()

    return model, loss
```

```python
def k_fold_cross_validation(K, perceptron_options, 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_perceptrons = np.zeros((len(perceptron_options), K))

    # Step 2: Iterate over all perceptron options based on the number of perceptrons
    for m, perceptrons in enumerate(perceptron_options):
        # K-fold cross-validation
        for k, (train_indices, valid_indices) in enumerate(kf.split(data)):
            # Extract the training and validation sets from the K-fold split
            X_train_k = torch.FloatTensor(data[train_indices])
            y_train_k = torch.FloatTensor(labels[train_indices])
            X_valid_k = torch.FloatTensor(data[valid_indices])
            y_valid_k = torch.FloatTensor(labels[valid_indices])

            # Create the perceptron model
            model = TwoLayerMLP(X_train_k.shape[1], perceptrons)
            optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

            # Train the model
            model, _ = train_perceptron_model(model, X_train_k, y_train_k, optimizer)

            # Evaluate the model on the validation fold
            prediction_probs = predict_with_model(model, X_valid_k)
            predictions = np.round(prediction_probs)

            # Retain the probability of error estimates
            error_valid_perceptrons[m, k] = np.sum(predictions != y_valid_k.numpy()) / len(y_valid_k)

    # Step 3: Compute the average probability of error (across K folds) for each model
    average_error_perceptrons = np.mean(error_valid_perceptrons, axis=1)

    # Find the optimal choice of perceptrons
    optimal_perceptrons = perceptron_options[np.argmin(average_error_perceptrons)]

    print(f"Best Number of Perceptrons: {optimal_perceptrons}")
    print(f"Average Cross-Validation Probability of Error: {np.min(average_error_perceptrons):.3f}")

    # Plot the cross-validation results
    plt.figure(figsize=(10, 10))
    plt.plot(perceptron_options, average_error_perceptrons)
    plt.title("Number of Perceptrons vs Cross-Validation Probability of Error")
    plt.xlabel("Number of Perceptrons")
    plt.ylabel("MLP Cross-Validation Probability of Error")
    plt.show()

    return optimal_perceptrons
```

```python
# List of perceptron counts to evaluate
P_list = [2, 4, 8, 16, 24, 32, 48, 64, 128]

# Convert -1/+1 labels into binary format (0/1), suitable for the MLP loss function
lb = LabelBinarizer()
y_train_binary = lb.fit_transform(y_train)[:, 0]

# Perform K-Fold Cross-Validation to find the best number of perceptrons
P_best = k_fold_cross_validation(K, P_list, X_train, y_train_binary)

# Output the best number of perceptrons
print(f"Optimal Number of Perceptrons: {P_best}")
```

OUTPUT

```
Best Number of Perceptrons: 64
Average Cross-Validation Probability of Error: 0.178
```



- The provided code implements a technique known as model restarts to enhance the robustness and mitigate the risk of getting trapped in suboptimal local minima during the training of a Multi-Layer Perceptron (MLP).

- The process involves training the same MLP architecture multiple times (in this case, 10 times) with random weight initializations. For each iteration, a new instance of the `TwoLayerMLP` model is created, and its parameters are randomly initialized. The model is then trained using stochastic gradient descent with momentum as the optimizer. After each training iteration, the resulting model and its corresponding loss on the training data are stored.

- Finally, the best-performing model is selected from the list of trained models based on the model with the lowest training loss. This approach helps increase the likelihood of finding a globally optimal set of parameters for the MLP, improving the overall effectiveness and robustness of the trained model.

- The test set is converted into a PyTorch tensor for compatibility with the MLP model. The model's predictions are obtained by applying a decision boundary at 0.5 to the sigmoid outputs, and the predictions are decoded back to the original class encoding using label binarization.

- The probability of error on the test set is then calculated by comparing the predicted labels with the true labels. Subsequently, the code generates a binary classification plot illustrating the distribution of correct and incorrect predictions on the test set. Additionally, the decision boundaries of the MLP on a grid, previously used for plotting SVM decision surfaces, are visualized.

- Finally, a confusion matrix is computed and displayed to assess the model's classification performance in terms of true positive, true negative, false positive, and false negative instances. The overall analysis provides insights into the MLP's efficacy and its ability to generalize to unseen data.

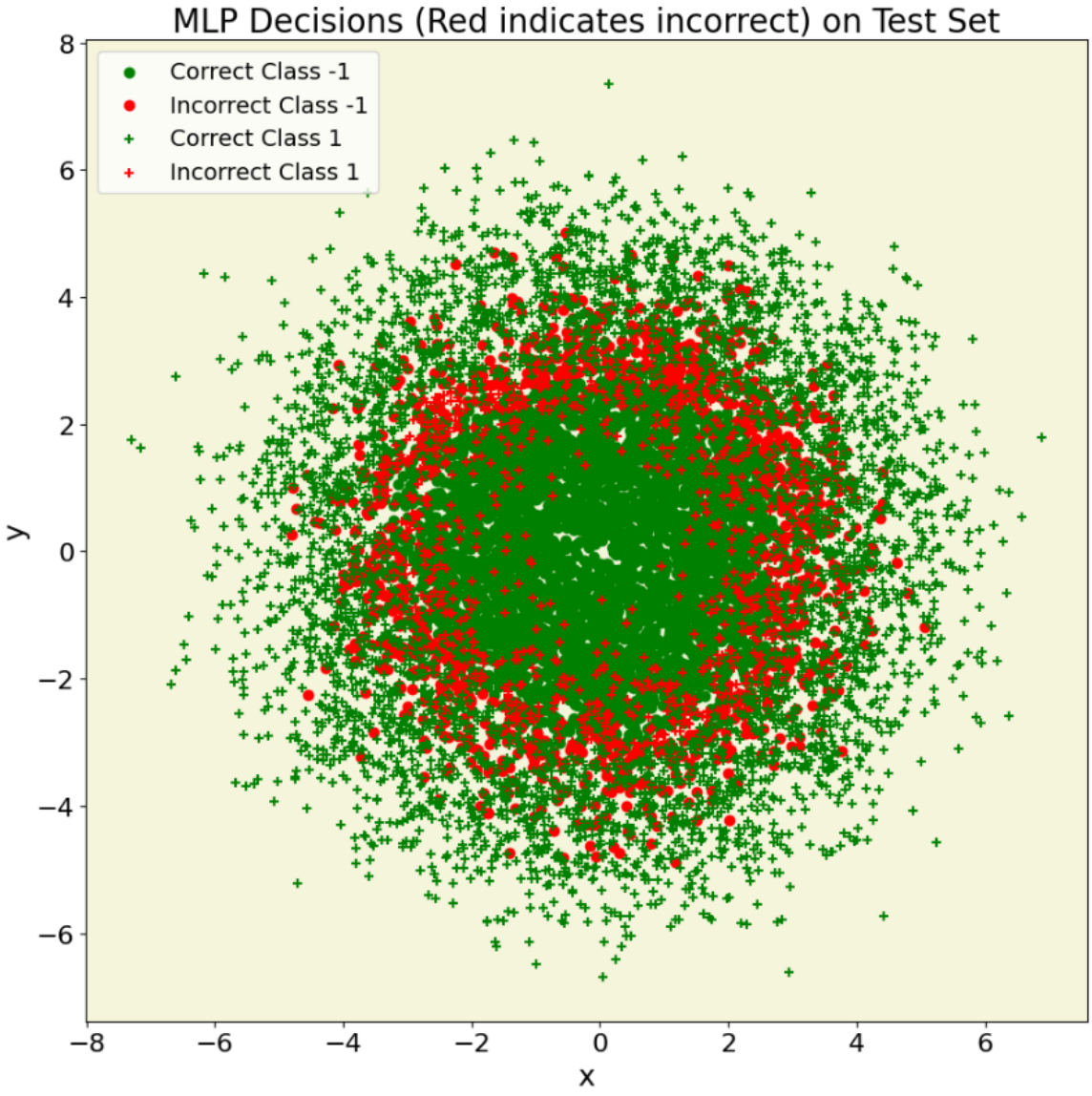
```python
# Function to Train with Model Restarts
def train_with_restarts(model_class, input_size, hidden_size, X_train, y_train, num_restarts=10, num_epochs=100):
    best_loss = float('inf')
    best_model = None
    criterion = torch.nn.BCEWithLogitsLoss()

    for restart in range(num_restarts):
        # Initialize a new model instance with random weights
        model = model_class(input_size, hidden_size)
        optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
        model.train()

        for epoch in range(num_epochs):
            optimizer.zero_grad()
            logits = model(X_train)  # Compute logits
            loss = criterion(logits, y_train.unsqueeze(1))  # Compute loss
            loss.backward()
            optimizer.step()

        # Keep track of the best-performing model
        if loss.item() < best_loss:
            best_loss = loss.item()
            best_model = model

    return best_model
```

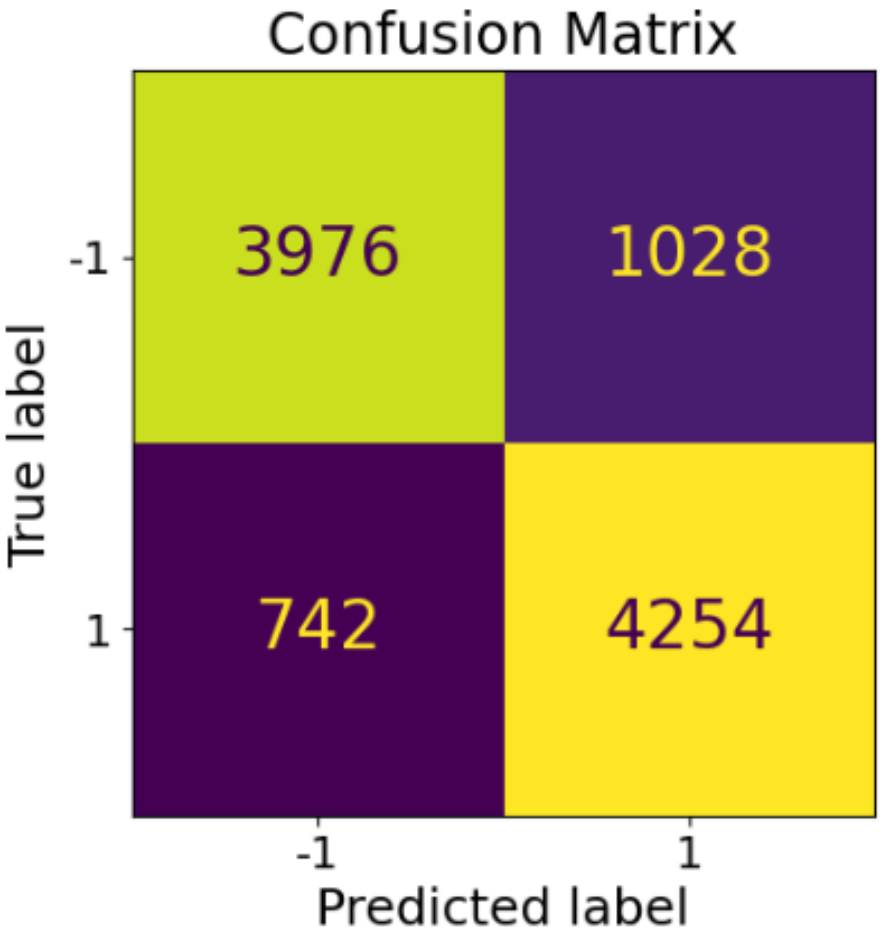
```python
def evaluate_and_visualize(model, X_test, y_test, lb):
    model.eval()
    with torch.no_grad():
        logits = model(X_test)  # Compute logits
        probabilities = torch.sigmoid(logits).numpy().flatten()
        predictions = (probabilities > 0.5).astype(int)
        decoded_predictions = lb.inverse_transform(predictions.reshape(-1, 1))
        decoded_y_test = lb.inverse_transform(y_test.reshape(-1, 1))

    # Calculate the test error probability
    test_error_probability = np.mean(decoded_predictions != decoded_y_test)
    print(f"Test Set Probability of Error: {test_error_probability:.4f}")

    # Plot binary classification results
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.set_title("MLP Decisions (Red indicates incorrect) on Test Set")

    # Identify correct and incorrect predictions
    correct_class_minus_1 = (decoded_y_test == -1) & (decoded_predictions == -1)
    incorrect_class_minus_1 = (decoded_y_test == -1) & (decoded_predictions == 1)
    correct_class_plus_1 = (decoded_y_test == 1) & (decoded_predictions == 1)
    incorrect_class_plus_1 = (decoded_y_test == 1) & (decoded_predictions == -1)

    # Scatter plot for classification results
    ax.scatter(
        X_test[correct_class_minus_1, 0], X_test[correct_class_minus_1, 1],
        c="green", marker="o", label="Correct Class -1"
    )
    ax.scatter(
        X_test[incorrect_class_minus_1, 0], X_test[incorrect_class_minus_1, 1],
        c="red", marker="o", label="Incorrect Class -1"
    )
    ax.scatter(
        X_test[correct_class_plus_1, 0], X_test[correct_class_plus_1, 1],
        c="green", marker="+", label="Correct Class 1"
    )
    ax.scatter(
        X_test[incorrect_class_plus_1, 0], X_test[incorrect_class_plus_1, 1],
        c="red", marker="+", label="Incorrect Class 1"
    )

    # Set plot background color
    ax.set_facecolor("beige")

    # Add labels and legend
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.legend()
    plt.tight_layout()
    plt.show()

    # Confusion matrix
    conf_mat = confusion_matrix(decoded_y_test, decoded_predictions, labels=[-1, 1])
    disp = ConfusionMatrixDisplay(confusion_matrix=conf_mat, display_labels=[-1, 1])
    disp.plot(cmap="viridis", values_format="d", colorbar=False)
    plt.title("Confusion Matrix")
    plt.show()
```

```
# Example Workflow
input_size = X_train.shape[1]
hidden_size = 64  # Best value based on cross-validation
lb = LabelBinarizer()
y_train_binary = torch.FloatTensor(lb.fit_transform(y_train)[:, 0])
y_test_binary = lb.transform(y_test)[:, 0]

# Convert data to PyTorch tensors
X_train_tensor = torch.FloatTensor(X_train)
X_test_tensor = torch.FloatTensor(X_test)

# Train the MLP model with restarts
best_mlp_model = train_with_restarts(TwoLayerMLP, input_size, hidden_size, X_train_tensor, y_train_binary)

# Evaluate and visualize the model's performance
evaluate_and_visualize(best_mlp_model, X_test_tensor, y_test_binary, lb)
```

OUTPUT

```
Test Set Probability of Error: 0.1770
```




Confusion Matrix
3976
1028
742
4254
-1
1
True label
-1
1
Predicted label

Question 2: 50%

In this question, you will use GMM-based clustering to segment a color image. Pick your color image from this dataset: https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/BSDS300/html/dataset/images.html
As preprocessing, for each pixel, generate a 5-dimensional feature vector as follows:

1. Append row index, column index, red value, green value, blue value for each pixel into a raw feature vector.

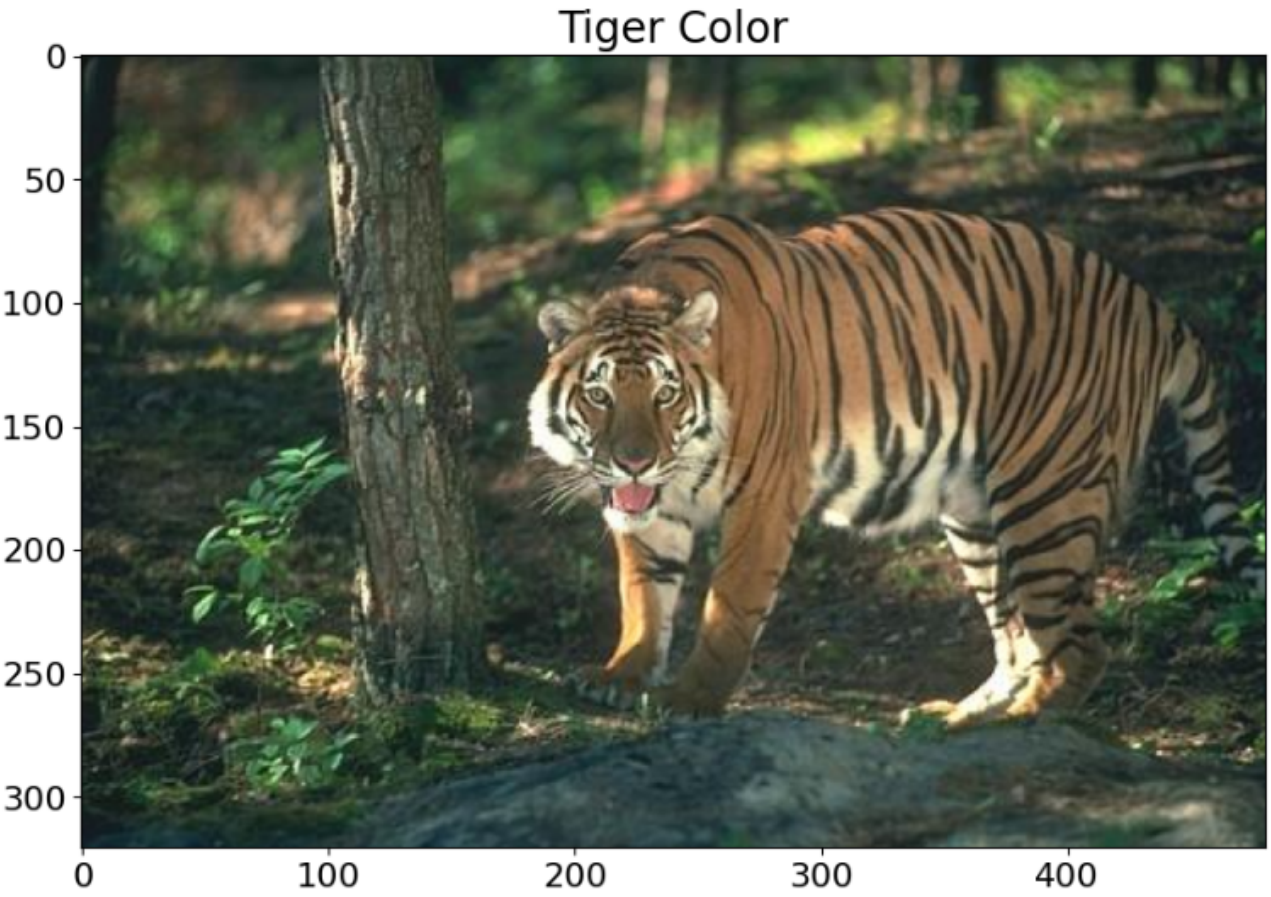
2. Normalize each feature entry individually to the interval $[0, 1]$, so that all of the feature vectors representing every pixel in an image fit into the 5-dimensional unit-hypercube.

Fit a Gaussian Mixture Model to these normalized feature vectors representing the pixels of the image. To fit the GMM, use maximum likelihood parameter estimation and 10-fold cross-validation (with maximum average validation-log-likelihood as the objective) for model order selection.
Once you have identified the *best* GMM for your feature vectors, assign the most likely component label to each pixel by evaluating component label posterior probabilities for each feature vector according to your GMM, similar to MAP classification. Present the original image and your GMM-based segmentation labels assigned to each pixel side by side for easy visual assessment of your segmentation outcome. If using grayscale values as segment/component labels, please uniformly distribute them between min/max grayscale values to have good contrast in the label image.
Hint: *If the image has too many pixels for your available computational power, you may downsample the image to reduce overall computational needs.*

Answer

```
import numpy as np
from sklearn.mixture import GaussianMixture
from sklearn.model_selection import KFold
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
from skimage.io import imread

# Set options for numpy to display numbers without scientific notation
np.set_printoptions(suppress=True)

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

# Set global plotting configurations for better visualization
plt.rc('axes', titlesize=20)  # Font size of the Title
plt.rc('font', size=24)  # Font size for text
plt.rc('axes', labelsize=18)  # Font size for the x and y labels
plt.rc('xtick', labelsize=16)  # Font size of the x-axis tick labels
plt.rc('ytick', labelsize=16)  # Font size of the y-axis tick labels
plt.rc('legend', fontsize=14)  # Font size of the legend
plt.rc('figure', titlesize=24)  # Font size of the figure title
```

```
# Load the image from the dataset URL
image = imread("https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/BSDS300/html/images/plain/normal/color/108005.jpg")

# Display the image
fig = plt.figure(figsize=(10, 10))
plt.imshow(image)
plt.title("Tiger Color")
plt.axis("on")
plt.show()
```



- The function `generate_feature_vector` takes an image as input and converts it into a NumPy array for further processing. It then obtains the row and column indices of the image to create a feature matrix that includes these indices along with pixel values.
- The number of channels in the feature matrix depends on whether the image is grayscale (1 channel) or a color image with RGB values (3 channels).
- For grayscale images, it calculates feature ranges as the difference between the maximum and minimum values, normalizing each feature to the unit interval $[0, 1]$ using max-min normalization.
- The process is similar for color images, where it additionally includes separate channels for each RGB color component.
- The function handles different image dimensions and returns the original image array along with the feature vector containing normalized pixel values. This feature vector has a shape of $(\text{height} \times \text{width}, 3)$ for grayscale images and $(\text{height} \times \text{width}, 5)$ for color images, reflecting the row and column indices, and the normalized pixel values for each channel.
- If the image has an incorrect number of dimensions, it prints a message indicating the issue. Overall, the function encapsulates the extraction and normalization of essential features from images, preparing them for further analysis or machine learning tasks.

```python
def generate_feature_vector(image):
    # Convert the image to a NumPy array
    image_array = np.array(image)

    # Obtain the row and column indices of the image (height and width)
    image_indices = np.indices((image_array.shape[0], image_array.shape[1]))

    # Initialize an array to store features: row index, column index, RGB
    # Handle grayscale images
    if image_array.ndim == 2:  # Grayscale image
        features = np.array([
            image_indices[0].flatten(),
            image_indices[1].flatten(),
            image_array.flatten()
        ]).T
    elif image_array.ndim == 3:  # RGB image
        features = np.array([
            image_indices[0].flatten(),
            image_indices[1].flatten(),
            image_array[..., 0].flatten(),
            image_array[..., 1].flatten(),
            image_array[..., 2].flatten()
        ]).T
    else:
        print("Incorrect image dimensions for feature extraction")
        return None, None

    # Normalize each feature to [0, 1]
    min_values = np.min(features, axis=0)
    max_values = np.max(features, axis=0)
    feature_ranges = max_values - min_values
    normalized_features = (features - min_values) / feature_ranges

    return image_array, normalized_features
```

- In the below code snippet, a Gaussian Mixture Model (GMM) is employed for image segmentation. The image, represented by the variable `image`, is first converted into a NumPy array (`img_np`), and a feature vector (`feature_vector`) is generated using the `generate_feature_vector` function.
- This feature vector incorporates information about the row and column indices, as well as the color channels of the image, appropriately normalized.
- Subsequently, the Expectation-Maximization (EM) algorithm is utilized to estimate the parameters of the GMM.
- The `GaussianMixture` class from `scikit-learn` is instantiated with four components (`n_components=4`) and other default parameters. The `fit_predict` method is then applied to perform hard clustering, assigning each pixel to the most probable component label using the `argmax` operation.
- The resulting segmentation labels are reshaped into an image format (`labels_img`), allowing for visualization of the segmentation outcome. The `matplotlib` library is used to display the image with color-coded segments, showcasing the segmentation result with four distinct components.

```python
# Generate feature vectors for the image
img_np, feature_vector = generate_feature_vector(image)

# Fit GMM with predefined parameters
gmm = GaussianMixture(n_components=4, max_iter=400, tol=1e-3, random_state=42)
gmm.fit(feature_vector)

# Predict the most likely component labels for each pixel
gmm_predictions = gmm.predict(feature_vector)

# Reshape the labels to match the original image shape
labels_img = gmm_predictions.reshape(img_np.shape[0], img_np.shape[1])

# Generate a color map for the segmentation
# Assign a unique random color to each GMM component
n_components = gmm.n_components
colors = np.random.randint(0, 255, size=(n_components, 3))  # Random RGB values for each component

# Map labels to colors
segmentation_colored = np.zeros((img_np.shape[0], img_np.shape[1], 3), dtype=np.uint8)
for i in range(n_components):
    segmentation_colored[labels_img == i] = colors[i]

# Display the original and segmented images
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# Original image
axes[0].imshow(img_np)
axes[0].set_title("Original Image")
axes[0].axis("off")

# Segmentation in color
axes[1].imshow(segmentation_colored)
axes[1].set_title(f"GMM Image Segmentation Result with $K = {n_components}$")
axes[1].axis("off")

plt.tight_layout()
plt.show()
```

OUTPUT

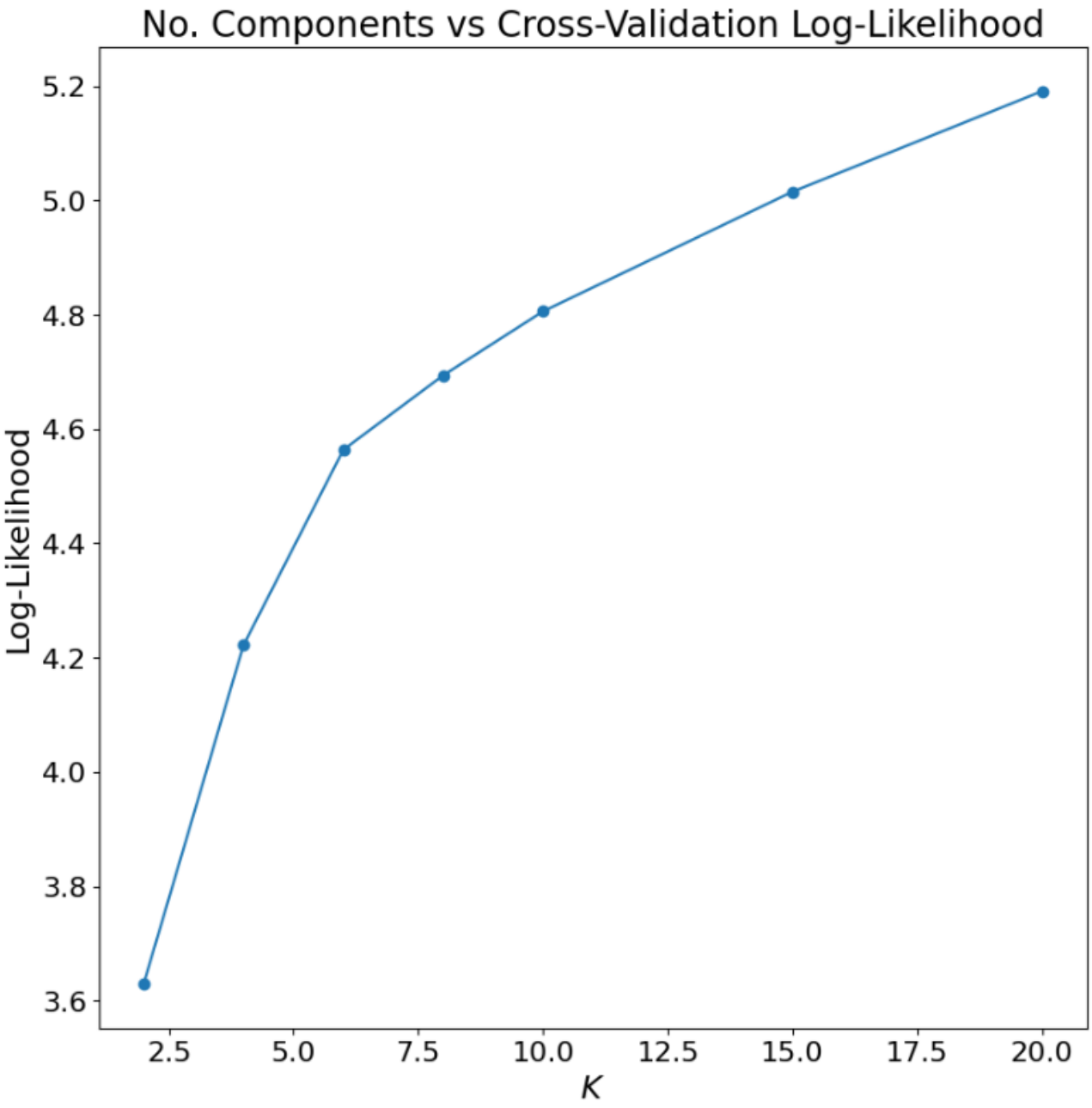
- The below code defines a function `k_fold_gmm_components` that performs k-fold cross-validation for Gaussian Mixture Models (GMM) with varying numbers of components.
- The function takes three parameters: `K` (the number of folds for cross-validation), `n_components_list` (a list of candidate numbers of components for GMM), and `data` (the dataset on which the cross-validation is performed). The function uses scikit-learn's `KFold` for data partitioning and `GaussianMixture` for training GMMs.
- The function iterates over the provided list of candidate components, fitting a GMM for each component and evaluating its performance using k-fold cross-validation.
- It calculates the average log-likelihood across folds for each component. After processing all components, it identifies and prints the best number of components based on the highest average log-likelihood.
- Additionally, it generates a plot illustrating the relationship between the number of components and the cross-validation log-likelihood. Finally, the function returns a list of the top three components based on their log-likelihood scores.

```python
def k_fold_gmm_components(K, n_components_list, data):
    # Initialize space for storing validation log-likelihoods
    log_likelihood_valid = np.zeros((len(n_components_list), K))

    # Initialize K-Fold
    kf = KFold(n_splits=K, shuffle=True)

    # Iterate over model orders
    for m, comp in enumerate(n_components_list):
        for fold, (train_indices, val_indices) in enumerate(kf.split(data)):
            # Fit GMM on training data
            gmm = GaussianMixture(n_components=comp, max_iter=400, tol=1e-3)
            gmm.fit(data[train_indices])

            # Compute log-likelihood on validation data
            log_likelihood_valid[m, fold] = gmm.score(data[val_indices])

    # Compute average log-likelihood for each model order
    avg_log_likelihood = np.mean(log_likelihood_valid, axis=1)

    # Find the optimal number of components with the highest average log-likelihood
    best_component = n_components_list[np.argmax(avg_log_likelihood)]
    max_log_likelihood = np.max(avg_log_likelihood)

    # Display results
    print(f"Best No. Cluster Components: {best_component}")
    print(f"Log-Likelihood Score: {max_log_likelihood:.3f}")

    # Plotting the log-likelihood vs. number of components
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.plot(n_components_list, avg_log_likelihood, marker="o")
    ax.set_title("No. Components vs Cross-Validation Log-Likelihood")
    ax.set_xlabel("$K$")
    ax.set_ylabel("Log-Likelihood")
    plt.show()

    return best_component
```

```
# Define parameters for cross-validation
k_folds = 10
n_components_list = [2, 4, 6, 8, 10, 15, 20]

# Perform cross-validation to find the best number of components
best_components = k_fold_gmm_components(k_folds, n_components_list, feature_vector)
```

OUTPUT

```
Best No. Cluster Components: 20
Log-Likelihood Score: 5.191
```

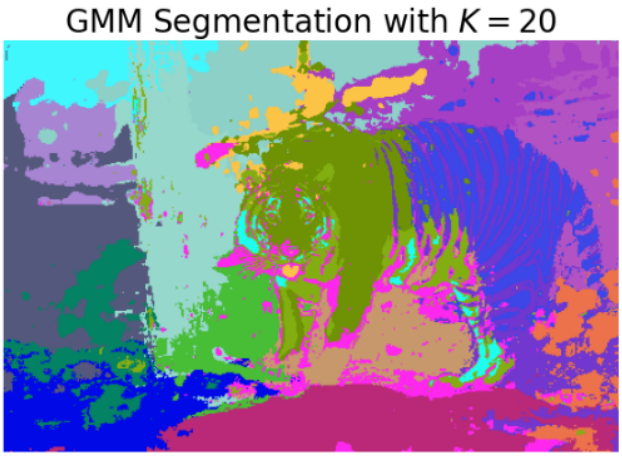


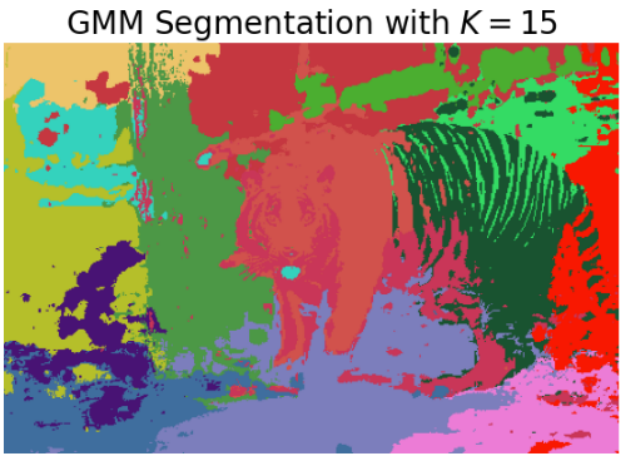
- This code creates a 2×2 subplot figure using `matplotlib` to visualize the results of Gaussian Mixture Model (GMM) segmentation on an example image.
- The first subplot displays the original image labeled as "Boat Color." Subsequent subplots present segmented images based on the top three numbers of components determined in the earlier GMM cross-validation.
- For each specified number of components, a GMM is trained and used to predict pixel-wise labels on the input image. The resulting segmented images are displayed in separate subplots with titles indicating the segmentation ranking and the corresponding number of components.
- The final plot showcases the original image, along with the three segmented images, providing a visual representation of the GMM clustering results. The code employs the best three components previously identified in the cross-validation process.

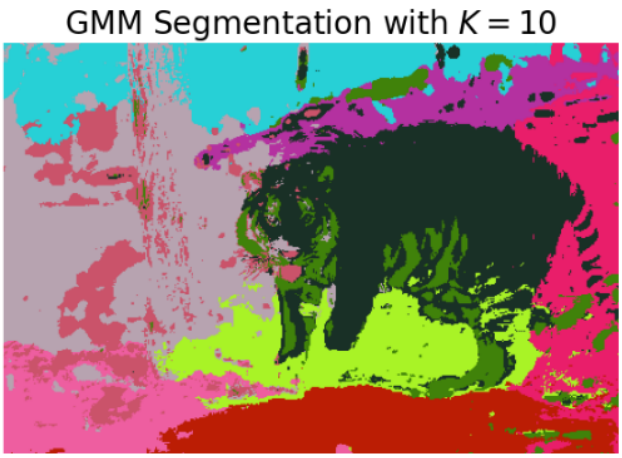
```python
def visualize_gmm_segmentation(image, feature_vector, best_components):
    # Create a 2x2 subplot to visualize the original image and three segmentations
    fig, axes = plt.subplots(2, 2, figsize=(12, 12))

    # Display the original image
    axes[0, 0].imshow(image)
    axes[0, 0].set_title("Original Image")
    axes[0, 0].axis("off")

    # Iterate over the best components and perform GMM segmentation
    for idx, n_components in enumerate(best_components):
        # Fit GMM with the specified number of components
        gmm = GaussianMixture(n_components=n_components, max_iter=400, tol=1e-3, random_state=42)
        gmm.fit(feature_vector)

        # Predict pixel labels
        gmm_predictions = gmm.predict(feature_vector)

        # Reshape the labels to match the original image dimensions
        labels_img = gmm_predictions.reshape(image.shape[0], image.shape[1])

        # Generate a random color mapping for the segmentation
        colors = np.random.randint(0, 255, size=(n_components, 3))
        segmentation_colored = np.zeros((image.shape[0], image.shape[1], 3), dtype=np.uint8)
        for i in range(n_components):
            segmentation_colored[labels_img == i] = colors[i]

        # Determine subplot position
        row, col = divmod(idx + 1, 2)
        axes[row, col].imshow(segmentation_colored)
        axes[row, col].set_title(f"GMM Segmentation with $K = {n_components}$")
        axes[row, col].axis("off")

    # Adjust layout for better spacing
    plt.tight_layout()
    plt.show()
```

```python
# Example usage with image, feature_vector, and best_components
best_components = [20, 15, 10]  # Replace with your top three GMM component counts
visualize_gmm_segmentation(image, feature_vector, best_components)
```

OUTPUT







- The code's visual results, combined with the observation on the trend of improving cross-validation log-likelihood with an increasing number of components (K) in Gaussian Mixture Model (GMM) order selection, highlight the complexity and challenges in unsupervised learning tasks.
- Specifically, the segmentation results show that selecting a lower value for K, such as $K = 4$, might yield visually more appropriate image segmentation than higher values like $K = 20$.
- However, determining what constitutes "good" performance in clustering tasks is inherently challenging due to the lack of ground truth labels. The ambiguity in selecting the optimal number of components reflects the broader issue of choosing evaluation metrics for unsupervised learning, emphasizing the inherent difficulty of these problems compared to their supervised counterparts.