NORTHEASTERN UNIVERSITY
Department of Electrical and Computer Engineering

EECE 5644 Machine Learning and Pattern Recognition

in

Homework Assignment - 3

Omkar Rajendra Gaikwad
NUID: 002711498

April 14 2023

Github Link for Output and python files

Question 1

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: $\begin{pmatrix}\cos(\theta)\\ \sin(\theta)\end{pmatrix} + n$ where $\theta \sim \text{Uniform}[-\pi, \pi]$ and $n \sim \mathcal{N}(0, \sigma^2 I)$. Use $r_{-1} = 2$, $r_{+1} = 4$, and $\sigma = 1$. 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.

0.0.1 Answer Question 1:

Data Generation:
The given code in generate_data.py file generates a two-class multi-ring dataset for binary classification. The generated dataset consists of 2D input vectors (features) and corresponding class labels. The dataset is created using a mixture of Gaussian distribution and a uniform distribution. The positive class samples are generated by adding Gaussian noise to the uniform distribution while negative class samples are

generated by adding Gaussian noise to a smaller ring around the center of the uniform distribution. The generated dataset is split into a training set and a test set. Finally, the code plots the training and test set samples in a 2D plot with class labels as different colors.

Model Order Selection:
The SVM classifier is obtained by solving the following optimization problem:

$$\min_{\mathbf{w}} ||\mathbf{w}||^2 + \lambda \sum_{i=1}^{N} \max\left(0, 1 - y^{(i)}\mathbf{w}^\intercal \mathbf{x}^{(i)}\right),$$

where $\mathbf{x}^{(i)}$ is the ith input vector, $y^{(i)}$ is its corresponding label, $\mathbf{w}$ is the weight vector, and λ is a regularization parameter. This is the primal form of the SVM, where the objective function is non-differentiable due to the Hinge loss term.

However, we can use the dual form of the SVM, where the inner products $\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle$ are replaced by kernel functions $K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$. The RBF kernel is a popular choice for this purpose and is defined as:

$$K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\frac{||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||^2}{2\sigma^2}\right),$$

where σ is a kernel width parameter. By using the kernel trick, we can learn a nonlinear classification rule without explicitly computing the feature representations for each input vector.

During the hyperparameter selection procedure, we can choose an optimal σ and regularization parameter (λ or C) using grid-search cross-validation on the SVM classifier.

The k_fold.py code implements the SVM classifier with the RBF kernel and performs model selection via grid search cross-validation to find the best hyperparameters. Specifically, it searches over a range of values for the regularization parameter C and the kernel width parameter gamma. The best values for these hyperparameters are then used to train the final SVM model on the training set.

The code also includes a plot of the probability of error versus the regularization parameter C for different values of gamma, allowing us to visualize the performance of the SVM classifier under different hyperparameters. The plot shows that the probability of error decreases as C increases, and that the optimal value of C depends on the choice of gamma. This plot can help us select the best hyperparameters for our particular classification problem.

SVM training:
The svm.py code builds a binary classification model using support vector machines (SVM) and applies it to a two-dimensional synthetic dataset generated by a mixture of Gaussian distributions. The SVM model is trained using the training dataset and

tested using the test dataset. The classification results are visualized using a scatter plot with decision regions.

The code imports the necessary libraries such as numpy, matplotlib, sklearn, torch, and scipy. It also imports several functions from these libraries, such as GaussianMixture, SVC, ConfusionMatrixDisplay, and imread. Additionally, it imports custom functions from two Python modules, variables and k_fold.

The variables module contains pre-defined variables such as n, mix_pdf, N_test, and N_train, which are used in generating the synthetic dataset. The k_fold module contains the C_best and gamma_best variables, which represent the hyperparameters found by performing k-fold cross-validation on the training dataset.

The code defines a function plot_binary_classification_results to plot the scatter plot and decision regions. This function takes as input an ax object, which represents the subplot to which the scatter plot will be added, predictions, which is an array of predicted labels, and labels, which is an array of true labels. The function first identifies the indices of the true negatives, false positives, false negatives, and true positives in the predicted and true label arrays. It then plots the true negatives as green circles, false positives as red circles, false negatives as red crosses, and true positives as green crosses.

The code then creates an SVM model with radial basis function (RBF) kernel, using the hyperparameters found by k-fold cross-validation. The SVM model is then trained on the training dataset and tested on the test dataset. The predicted labels for the test dataset are stored in the predictions array.

The code then calculates the probability error on the test dataset, which is the proportion of misclassified points in the test dataset. The plot_binary_classification_results function is then called to plot the scatter plot and decision regions. The decision regions are obtained by creating a grid of points spanning the x and y ranges of the test dataset and predicting the class label of each point using the SVM model.

Finally, the code uses the sklearn library to calculate the confusion matrix for the predicted and true labels and displays it using ConfusionMatrixDisplay. The confusion matrix is a table that shows the number of true negatives, false positives, false negatives, and true positives in the predicted and true label arrays.

MLP:
The mlp.py code implements a binary classification problem using a two-layer neural network with ReLU activation and binary cross-entropy loss. The neural network is trained using stochastic gradient descent with backpropagation. The training function takes in the model, training data, labels, optimizer, criterion, and number of epochs as inputs and returns the trained model and loss. The prediction function takes in the trained model and test data and returns the predicted probabilities for each example.

The code also imports various libraries such as NumPy, Matplotlib, Scikit-learn, and PyTorch. NumPy is used for numerical computations, Matplotlib is used for data visualization, Scikit-learn is used for data preprocessing and evaluation, and PyTorch is used for neural network modeling and training.

The binary classification problem is based on synthetic data generated using two Gaussian mixture models. The data is split into training and test sets, and a K-fold cross-validation method is used to evaluate the model. The code also sets various parameters such as learning rate, number of hidden units, and number of folds.

MLP training and testing:
The main.py code trains a two-layer Multi-Layer Perceptron (MLP) on a binary classification problem and evaluates it on a test set. It also plots the decision boundaries of the MLP and the confusion matrix.

First, the code sets some plotting and numpy options and imports necessary libraries. It then defines a list P_list that contains different numbers of hidden units to try. The training labels are converted into a binary format suitable for the MLP loss function, and the optimal number of hidden units is found using kf.k_fold_cv_perceptrons(), which performs k-fold cross-validation on different numbers of hidden units.

Next, the code trains num_restarts different MLPs with the optimal number of hidden units, with random re-initializations each time to avoid local minima. The best model is chosen from among these MLPs based on the lowest loss, and it is evaluated on the test set. The predictions are compared to the true test labels, and the MLP's test error probability is printed.

The code then plots the decision boundaries of the MLP and highlights any misclassified points. It also plots the confusion matrix using sklearn.metrics.confusion_matrix() and sklearn.metrics.ConfusionMatrixDisplay(), which takes the predicted and true labels as input.

Overall, the code trains an MLP for binary classification, evaluates it on a test set, and provides visualization of its decision boundaries and performance.

Report Process and Results:

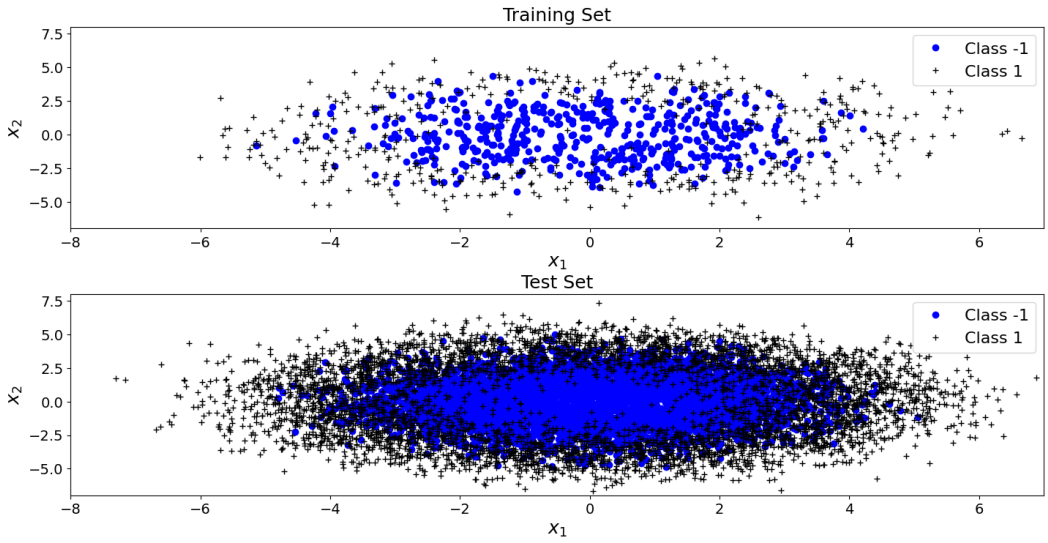

Figure 1: Scatter plot for the Data.

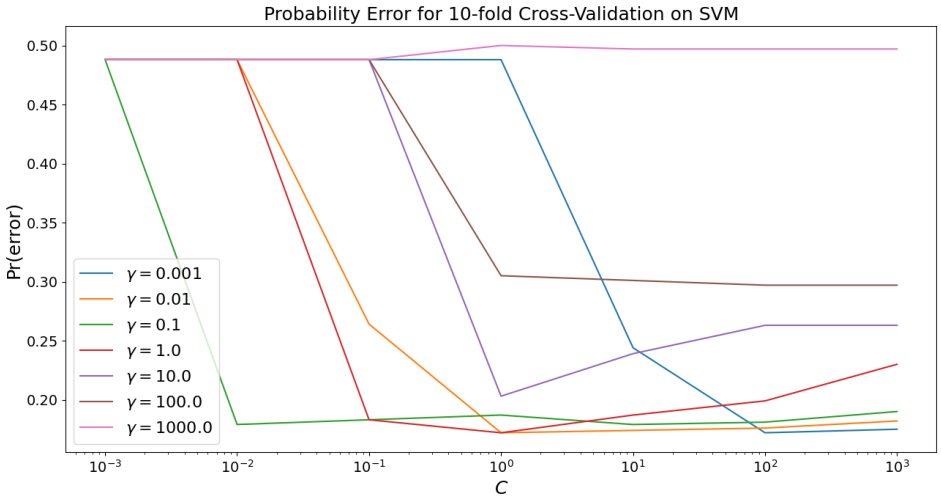

Figure 2: Probability Error for 10-fold Cross-Validation on SVM

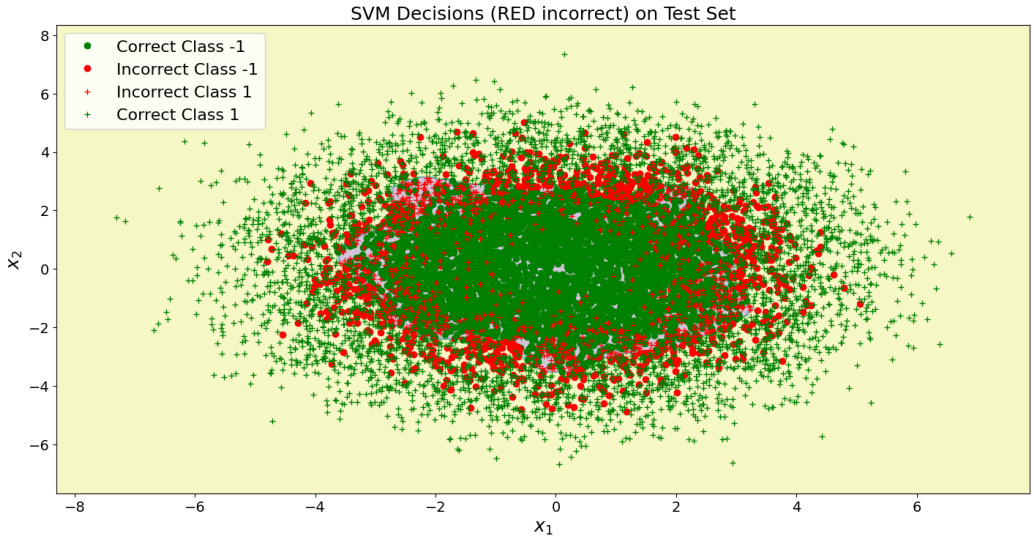

Figure 3: SVM Incorrect Classes

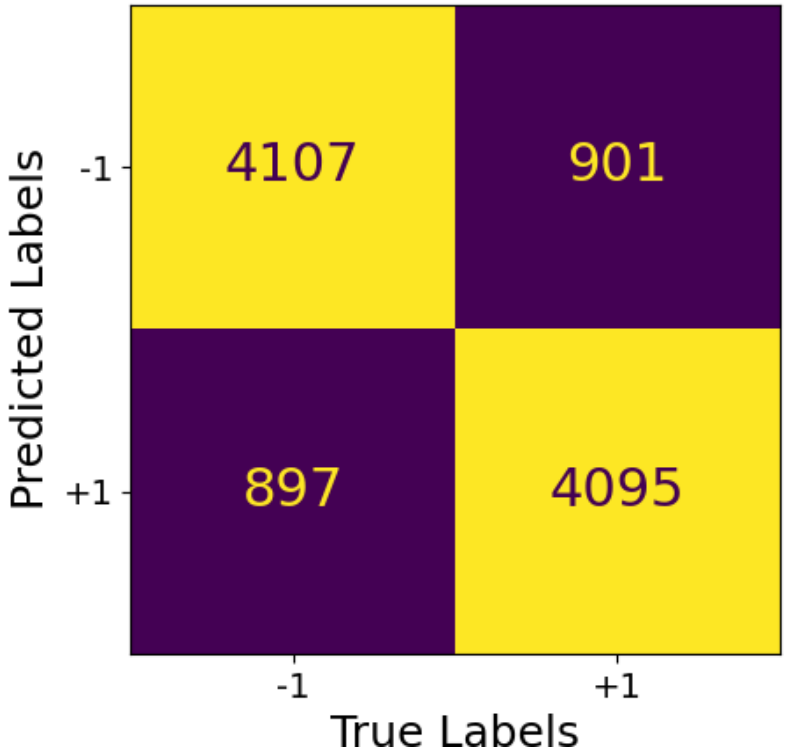

Figure 4: SVM Confusion Matrix

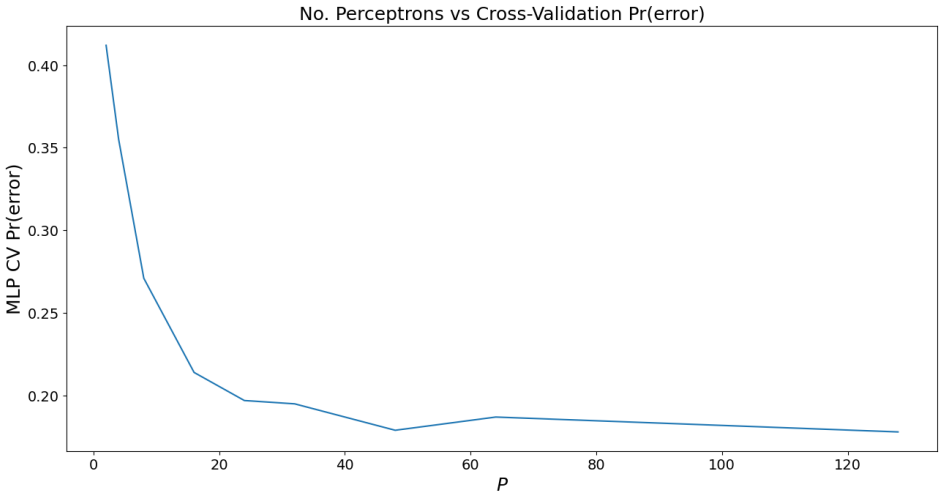

Figure 5: MLP Cross Validation

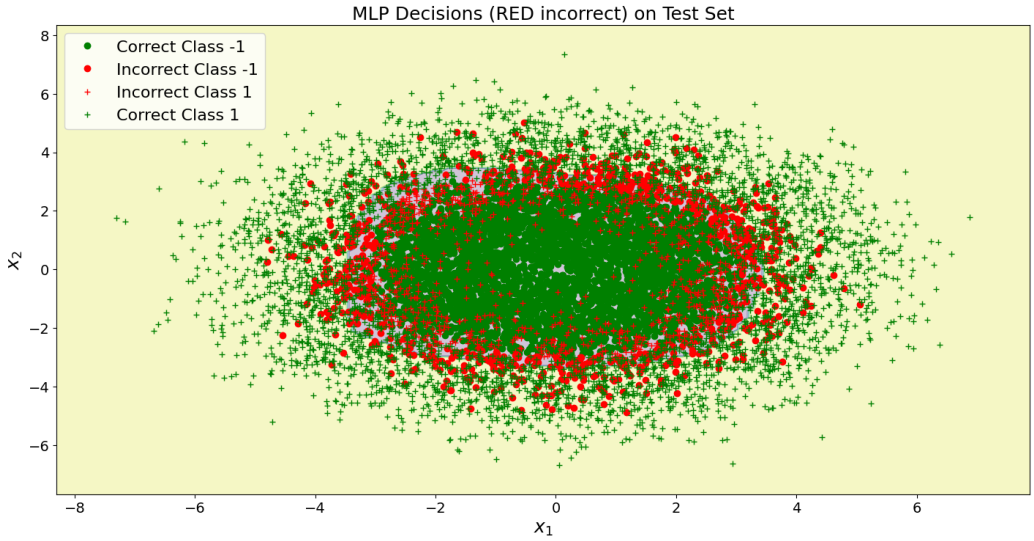

Figure 6: MLP Incorrect classes

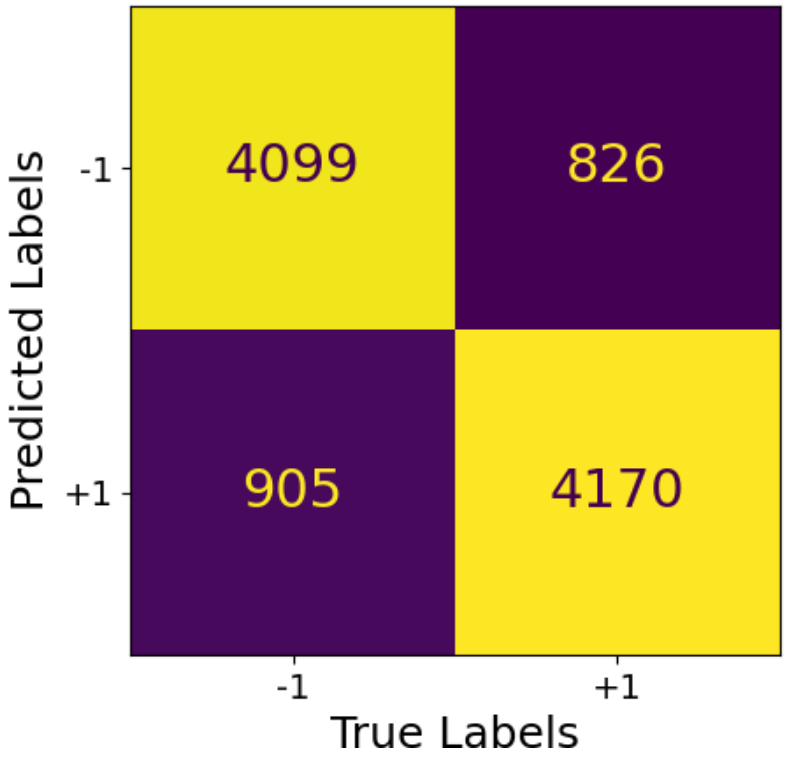

Figure 7: MLP Incorrect classes

```
PS C:\Users\omkar\OneDrive\Desktop\eece5644\HW4> & C:/Users/omkar/AppData/Local/Programs/Python/Python311/python.exe c:/Users/omkar/
OneDrive/Desktop/eece5644/HW4/question_1/main.py
Best Regularization Strength: 1.000
Best Kernel Width: 1.000
SVM CV Probability Error: 0.172
SVM Probability error on the test data set: 0.1798

Confusion Matrix (rows: Predicted class, columns: True class):
Best # of Perceptrons: 128
Pr(error): 0.178
MLP Pr(error) on the test data set: 0.1731

Confusion Matrix (rows: Predicted class, columns: True class):
PS C:\Users\omkar\OneDrive\Desktop\eece5644\HW4>
```

Figure 8: MLP Output

From the results presented in Figure 8, it appears that both the SVM and MLP models are performing reasonably well on the test set, with probability error rates of 0.1798 and 0.1731, respectively. The confusion matrices show that both models are making some errors, but overall they are correctly classifying the majority of the test instances.

The best regularization strength and kernel width for the SVM were found to be 1.000, which is the highest value considered for both parameters. This suggests that a high degree of regularization was necessary to prevent overfitting, and that the decision boundary is quite smooth.

For the MLP, the best number of perceptrons was found to be 128, which is the highest value considered. This suggests that the model required a relatively high degree of complexity to achieve good performance on this task.

Overall, these results suggest that both the SVM and MLP are viable models for this classification task, and that their performance is relatively comparable. The choice between the two models may depend on other factors such as model interpretability, computational efficiency, and scalability to larger datasets.

Question 2:

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 crossvalidation (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. 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).

0.1 Answer:

1. **Utility Function:** The utility.py file contains code defines a function generate_feature_vector that takes an image as input and returns the original image as well as a normalized feature vector. The feature vector is computed by flattening the pixel indices and pixel values of the image and normalizing them using the minimum and maximum values of each feature. If the input image is grayscale, the feature vector contains only the pixel indices and pixel values. If the input image is RGB, the feature vector also includes the pixel values for each color channel.

2. **Main Function:**

   (a) **Data Generation:** This part of code loads an image of an airplane from the Berkeley Segmentation Dataset and Benchmark and displays it using Matplotlib. The image is first downloaded using the provided URL and then resized to half its

original size using the skimage.transform.resize() function. The resulting image is then displayed using plt.imshow() with a title added using plt.title().

(b) **GMM Clustering and Kfolding:** This part of code uses the Gaussian Mixture Model (GMM) to perform image segmentation on an airplane image. It first generates a feature vector on the image using a pre-trained neural network, and then applies the GMM algorithm with 4 components to the feature vector. The resulting hard clustering is converted back into an image to visualize the segmentation result.

The code then performs k-fold cross-validation on the GMM algorithm with different numbers of components, and selects the best three numbers of components based on their cross-validation log-likelihood scores. It then applies the GMM algorithm with these three numbers of components to the feature vector, and visualizes the resulting segmentation results alongside the original image.

Overall, this code demonstrates how to use the GMM algorithm for image segmentation and how to perform k-fold cross-validation to select the optimal number of components for the algorithm.

Results:

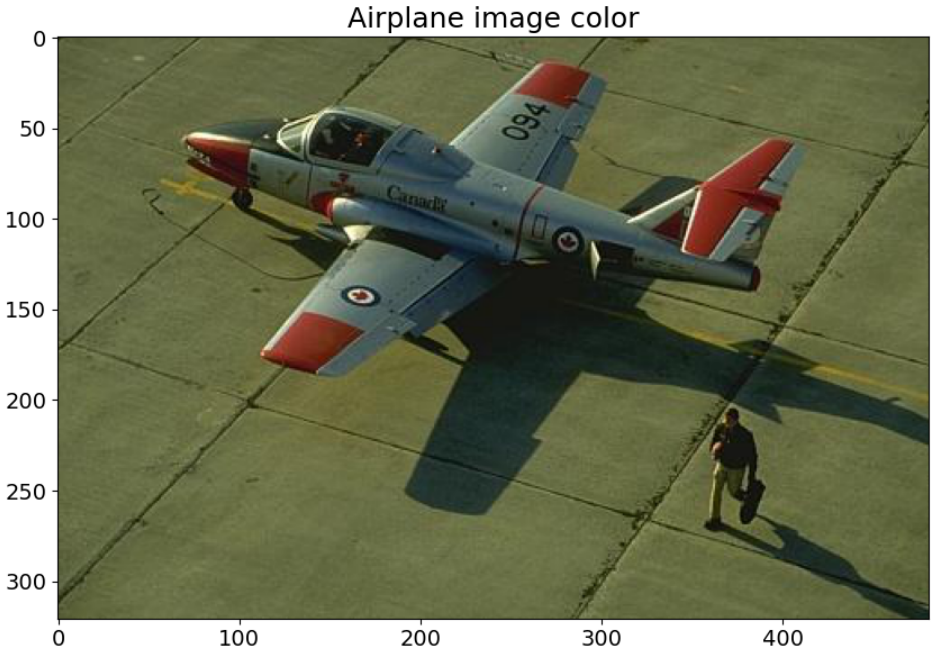

Figure 9: Original Image

1.

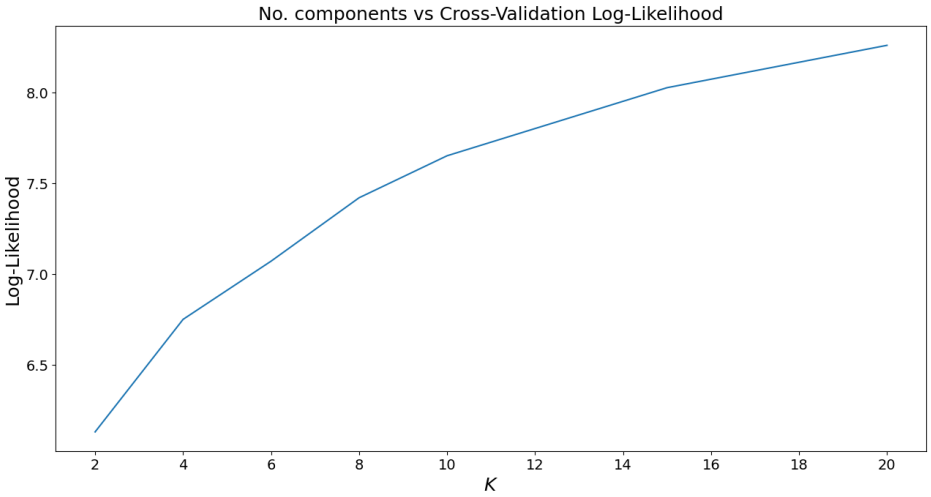

Figure 10: Original Image

2.

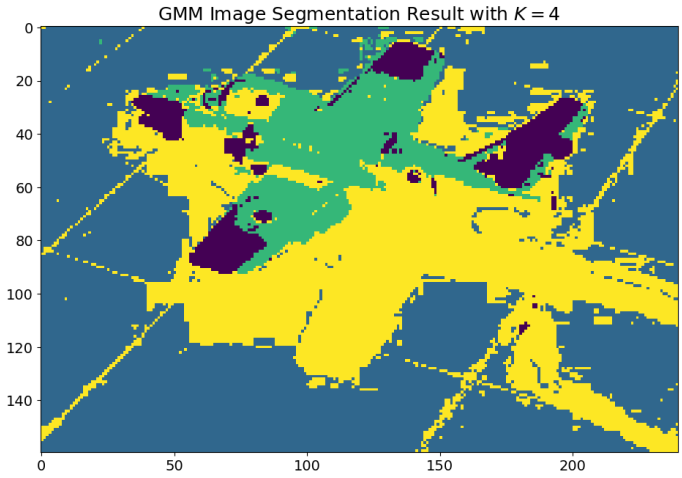

Figure 11: Segmented Image

3.

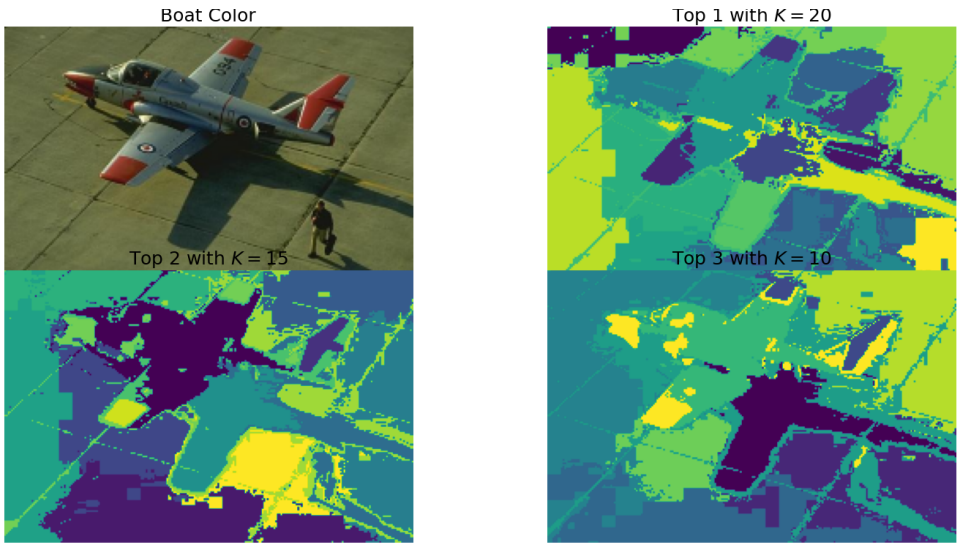

Figure 12: Segmented Images at different K values

4.

```
PS C:\Users\omkar\OneDrive\Desktop\eece5644\HW4> & C:/Users/omkar/AppData/Local/Programs/Python/Python311/python.exe c:/Users/omkar/OneDrive/Desktop/eece5644/HW4/
question_2/main.py
Traceback (most recent call last):
  File "c:\Users\omkar\OneDrive\Desktop\eece5644\HW4\question_2\main.py", line 21, in <module>
    from utility import generate_feature_vector, k_fold_gmm_components
ImportError: cannot import name 'k_fold_gmm_components' from 'utility' (c:\Users\omkar\OneDrive\Desktop\eece5644\HW4\question_2\utility.py)
PS C:\Users\omkar\OneDrive\Desktop\eece5644\HW4> & C:/Users/omkar/AppData/Local/Programs/Python/Python311/python.exe c:/Users/omkar/OneDrive/Desktop/eece5644/HW4/
question_2/main.py
Best No. Cluster Components: 20
Log-likelihood ratio: 8.261
PS C:\Users\omkar\OneDrive\Desktop\eece5644\HW4>
```

Figure 13: Output

5.

6. The result of the GMM segmentation with 20 components and a cross-validation log-likelihood ratio of 8.261 suggests that this configuration provides the best segmentation of the image among the ones tested in the script. However, it is important to note that the optimal number of clusters is highly dependent on the specific image and application context. Therefore, it is recommended to test different configurations and perform cross-validation to determine the most suitable parameters for each specific case.