CS479 Programming Assignment 2

Ashlee Ladouceur and Adam Lychuk

Due Date: 3/26/2019

Date Turned In: 3/26/2019

Division of work:

- Ashlee Ladouceur designed and developed Question 1 and 2 code
- Adam Lychuk designed and developed Question 3 code
- Ashlee Ladouceur wrote the Implementation Details for Question 1 and 2, Results and Discussion for Question 1 and 2, and the Program Listings for Question 1 and 2
- Adam Lychuk wrote the Technical Discussion, the Implementation Details for Question 3, and Results and Discussion for Question 3, and the Program Listings for Question 3

Technical Discussion

Maximum likelihood estimation is used to estimate the mean and covariance of a given distribution. This is often useful when handling distributions in which you do not know or have access to the parameters that created those distributions. The equations to obtain the mean and covariance matrices are simplified for easy calculation and interestingly the point estimate mean of the distribution is equal to the arithmetic mean.

$$\widehat{\mu} = \frac{1}{n} \sum_{k=1}^{n} x_k \qquad \widehat{\Sigma} = \frac{1}{n} \sum_{k=1}^{n} (x_k - \widehat{\mu})(x_k - \widehat{\mu})^t$$

Equation 1: Maximum likelihood estimation formulas for the sample mean(left) and the sample variance(right).

These estimates of the mean and covariance matrices can be easily substituted into Bayesian classification methods, allowing us to classify gaussian distributions for which we do not know the parameters used to create the initial distribution. Questions 1 and 2 focus on comparing the accuracy of these estimates to the actual parameter used to create them.

Skin color has been found to follow a peaked gaussian distribution when examined with normalized color space values that attempt to discard luminance information. The popular RGB color space can be converted to normalized RGB which attempts to minimize luminance values by discarding the color blue.

$$Rnorm = R / (R + G + B) \; Gnorm = G / (R + G + B)$$

Equation 2: RGB normalization formulas.

The RGB color space can also be converted YCbCr color space. In separating the luminance Y from the chroma values Cb and Cr we get a stronger normal distribution using just Cb and Cr.

$$\begin{gathered} Y = 0.299R + 0.587G + 0.114B \\ Cb = -0.169R - 0.332G + 0.500B \\ Cr = 0.500R - 0.419G - 0.081B \end{gathered}$$

Equation 3: YCbCr normalization formulas.

After normalizing the color values Bayesian Threshold Estimation can be applied to each value, using the parameters computed using ML, to decide whether a pixel is skin pixel or not a skin pixel.

$$g(x) = \frac{1}{2\pi|\Sigma|^{1/2}} exp[-\tfrac{1}{2}(x-\mu)^{t}\Sigma^{-1}(x-\mu)]$$

Equation 4: Threshold discriminant.

$$g(x) = exp[-\tfrac{1}{2}(x-\mu)^{t}\Sigma^{-1}(x-\mu)]$$

Equation 5: Threshold discriminant.

The discriminant in equation 4 is compared against a threshold and decided upon using ROC curves to classify the pixel. The discriminant is normalized in equation 5 g(x)/g(mu) to be used with the normalized values of the new color space normalized RGB, or YCbCr.

Implementation Details

Question 1 and 2

In order to estimate the sample mean and covariance using the samples generated from the first project, Maximum Likelihood (ML) estimation theory is utilized. Once the samples from Project 1 are available to the main function, the samples are passed to the MLE.h and MLE.cpp files documented in the Program Listings section of this document. The sample mean is estimated by returning the sum divided by the size of the data. The sample covariance is estimated by returning the transposed sum divided by the size of the data. The main function then saves these new parameters for the next step.

The next step classifies the data according to the estimated parameters of the distributions. The classification files from Project 1, Classifier.h and Classifer.cpp, is utilized for this. These files use Bayes' for classification and adds one function for adding a threshold function to case three for Question 3.

For 1(b) and 2(b), the code must select 1,000 random samples from the 100,000 samples generated in project 1 to estimate parameters from a random subset to explore ML estimations on smaller sets of data. To do this within the code, a random index in the range of 100,000 is selected 1,000 times and saved into a subset for mean and covariance estimation using the rand() function. Those selected samples are then saved in the main function to be classified and tested.

It is important to note that Question 1 and 2 are incredibly similar and both use many of the same functions from Project 1. To summarize the implementation, 1(a) and 2(a) first estimate the sample mean and covariance according to their respective samples from Project 1, and then classifies them and outputs the performance to the terminal. 1(b) and 2(b) both select 1000 random data points from the overall samples to estimate the sample mean and covariance, which are then classified, and the performance is again output on the terminal.

Question 3

Basic skin detection is performed using maximum likelihood estimation and thresholding with a Bayesian discriminant. Three images are provided along with reference of each providing the position of skin pixels. Images are represented and manipulated using the functionality of the OpenCV library, where each image is treated as Mat object. This functionality is included in the highgui and imgproc libraries. Estimation is performed on the first image using maximum likelihood estimation. Once in the RGB color space, and then again in the YCbCr color space. ML estimation is implemented in MLE.h and MLE.cpp which are also used in Question 1 and 2 and discussed above.

Collecting sample “skin data”, and then estimating its mean and covariance matrices, is performed by iteration over the entire training image in function calculate_ML_image contained in part-3.cpp. While iterating over the image, conversions are performed first transforming from the BGR color space to a normalized RGB space defined in equation 2 of the technical discussion. The normalized blue channel is discarded and the normalized R and G are stored. The BGR color space is then converted again to the YCrCb color space using equation 3 which is also provided in the technical discussion. Y the luminance is discarded, and Cr and Cb stored. The pixel is then checked to be a skin pixel or a non skin pixel. If the pixel is a skin pixel it is then pushed to a vector. The vector is then passed to the maximum likelihood functions for both color spaces. The functions output, the estimated mu and covariance of the distribution of skin pixels, is stored in a struct containing all ML estimates.

With the newly obtained estimates the thresholding classification is then performed on the two other images in both the normalized RGB and YCbCr color spaces. This process is contained in threshold_classification function in part-3.cpp. The images are again iterated over. Conversions are made again and then inputted into the normalized thresholding function implemented in Classifier.cpp. It computes the discriminant using equation 5 in the technical discussion and then checks it against the provided threshold. The classification is performed at multiple thresholds in increments of .05 and the false negatives, false positives, total skin pixels, and total non-skin pixels are used to generate ROC curves for the classification technique. The data used to generate the curves is outputted to a file and the curves show both false rejection for the RGB and YCrCb color spaces, and False Acceptance for the RGB and YCrCb color spaces plotted against each other.

Results and Discussion

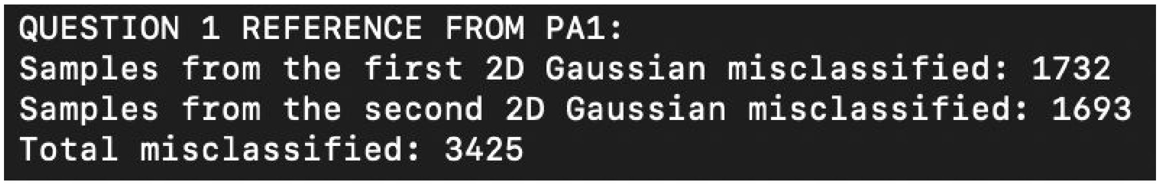
Question 1

Question 1 Preparation

Question 1 uses the 200,000 samples generated in Project 1, question 1 using the 2D Gaussian distributions shown in Figure 1. To ensure ease and accuracy of comparison, the code for Project 2 includes the code from Project 1 to generate and reference the classification accuracy of Project 1. The output from the terminal for the samples when the covariance is known (from Project 1) is shown in Figure 2. This is important for comparison purposes for 1(a) and 1(b).

$$\mu_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \Sigma_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \mu_2 = \begin{bmatrix} 4 \\ 4 \end{bmatrix} \Sigma_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figure 1: The 2D Gaussian distributions used for Question 1.

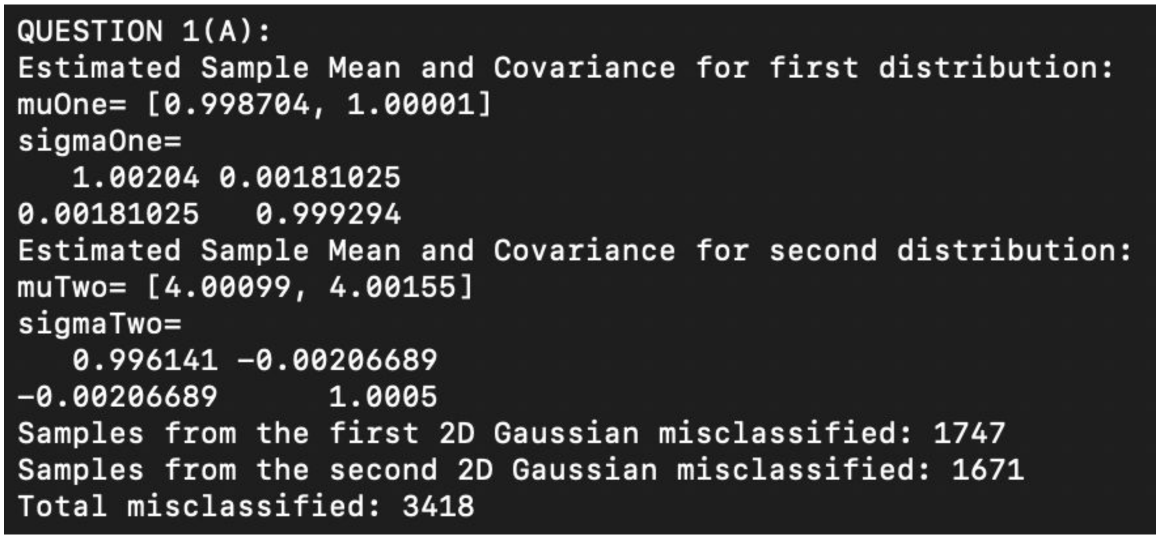


```
QUESTION 1 REFERENCE FROM PA1:
Samples from the first 2D Gaussian misclassified: 1732
Samples from the second 2D Gaussian misclassified: 1693
Total misclassified: 3425
```

Figure 2: Output in terminal that classifies the question 1 samples knowing the distributions.

Question 1(a)

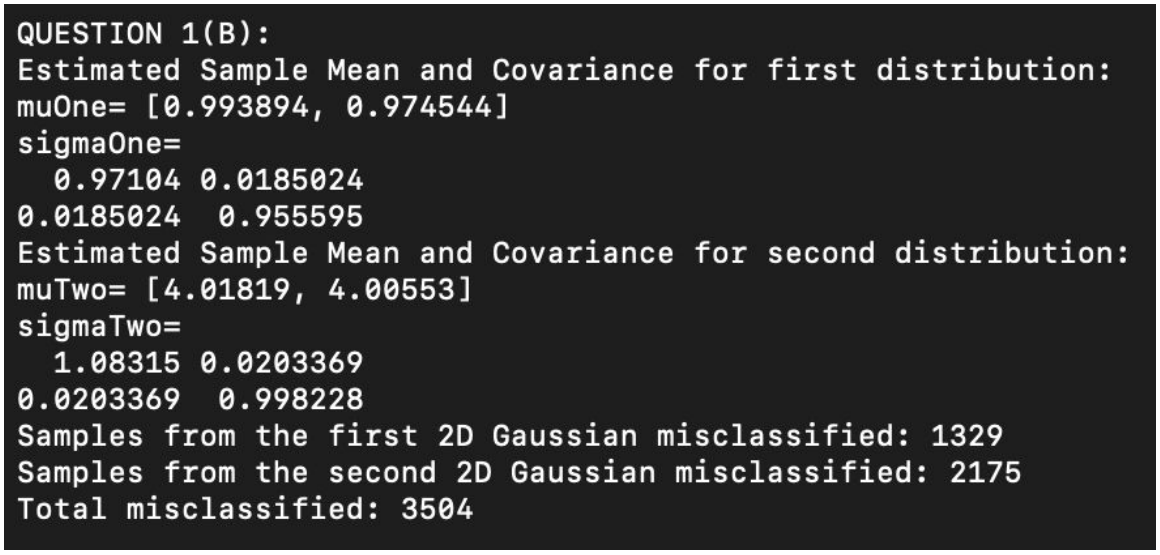
Question 1(a) uses the samples generated from Project 1 to estimate the parameters of each distribution using Maximum Learning (ML) estimation, and then classifies those samples assuming $P(w_1) = P(w_2)$. The output for Question 1(a) parameter estimations and classification performance is shown in Figure 3.



```
QUESTION 1(A):
Estimated Sample Mean and Covariance for first distribution:
muOne= [0.998704, 1.00001]
sigmaOne=
   1.00204 0.00181025
0.00181025   0.999294
Estimated Sample Mean and Covariance for second distribution:
muTwo= [4.00099, 4.00155]
sigmaTwo=
   0.996141 -0.00206689
-0.00206689      1.0005
Samples from the first 2D Gaussian misclassified: 1747
Samples from the second 2D Gaussian misclassified: 1671
Total misclassified: 3418
```

Figure 3: Output in terminal that displays ML estimations of samples from Project 1, question 1. The mu and sigma for both distributions are estimated, and then classified.

Question 1(b)
Question 1(b) randomly selects 1,000 samples from each sample distributions of 100,000 (hundredth of each sample distribution), then estimates the parameters of the subset of samples using ML estimations. The output for Question 1(b) parameter estimations and classification performance is shown in Figure 4.



```
QUESTION 1(B):
Estimated Sample Mean and Covariance for first distribution:
muOne= [0.993894, 0.974544]
sigmaOne=
  0.97104 0.0185024
0.0185024  0.955595
Estimated Sample Mean and Covariance for second distribution:
muTwo= [4.01819, 4.00553]
sigmaTwo=
  1.08315 0.0203369
0.0203369  0.998228
Samples from the first 2D Gaussian misclassified: 1329
Samples from the second 2D Gaussian misclassified: 2175
Total misclassified: 3504
```

Figure 4: Output in terminal that displays ML estimations of the random 1/100 of samples from Project 1, question 1. The mu and sigma for both distributions are estimated, and then classified.

Question 1 Analysis

Table 1 summarizes the misclassification performance for the estimations in 1(a) and 1(b), compared to the performance when the distribution parameters are known in PA1. The performance rate is nearly identical in PA1 and 1(a), showing the ML estimation performs great with such a large sample set. However, as shown in 1(b), when 1/100 of the samples are randomly selected, the performance falls greatly. You can see this is mostly due to the poor sigma two estimation. The lesson learned in question 1 is that a large sample set is necessary for proper estimation and performance when using ML estimation.

	Known Parameters PA1	Estimated Parameters 1(a)	Estimated Parameters 1(b)
Misclassified Sample One	1732	1747	1952
Misclassified Sample Two	1693	1671	8014
Total Misclassified	3425	3418	9966

Table 1: Misclassifications of Project 1 when sample distribution is known, 1(a) where sample distribution is estimated, and 1(b) where sample subset distribution is estimated.

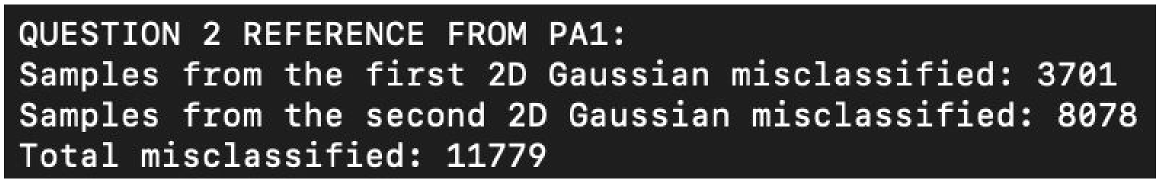
Question 2

Question 2 Preparation

Question 2 uses the 200,000 samples generated in Project 1, question 2 using the 2D Gaussian distributions shown in Figure 5. To ensure ease and accuracy of comparison, the code for Project 2 includes the code from Project 1 to generate and reference the classification accuracy of Project 1. The output from the terminal for the samples when the covariance is known (from Project 1) is shown in Figure 6. This is important for comparison purposes for 2(a) and 2(b).

$$\mu_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \Sigma_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mu_2 = \begin{bmatrix} 4 \\ 4 \end{bmatrix} \quad \Sigma_2 = \begin{bmatrix} 4 & 0 \\ 0 & 8 \end{bmatrix}$$

Figure 5: The 2D Gaussian distributions used for Question 2.

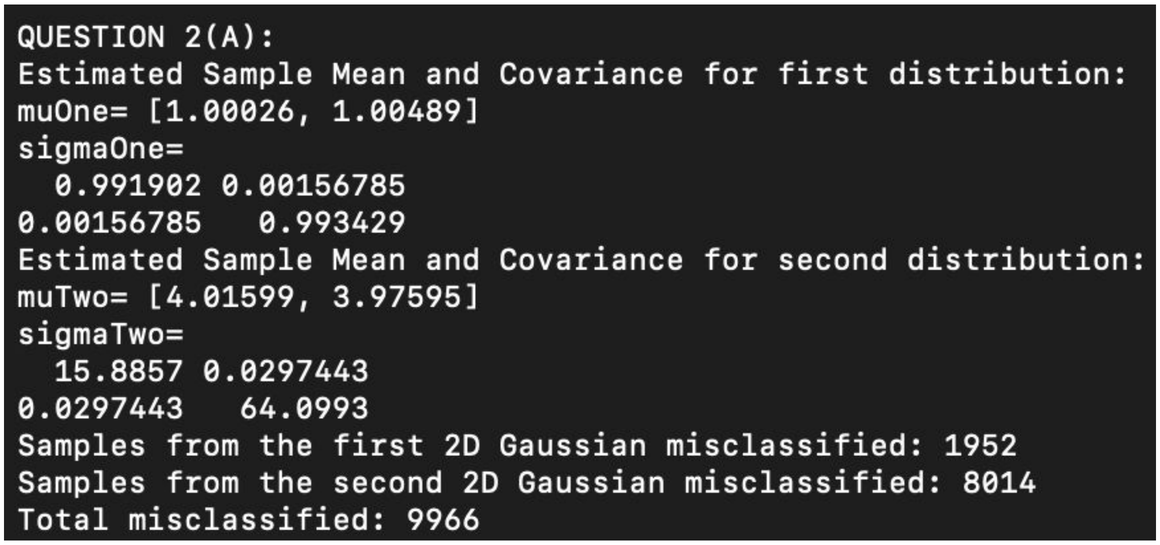


```
QUESTION 2 REFERENCE FROM PA1:
Samples from the first 2D Gaussian misclassified: 3701
Samples from the second 2D Gaussian misclassified: 8078
Total misclassified: 11779
```

Figure 6: Output in terminal that classifies the question 2 samples knowing the distributions.

Question 2(a)

Question 2(a) uses the samples generated from Project 1 to estimate the parameters of each distribution using Maximum Learning (ML) estimation, and then classifies those samples assuming $P(w_1) = P(w_2)$. The output for Question 2(a) parameter estimations and classification performance is shown in Figure 7.

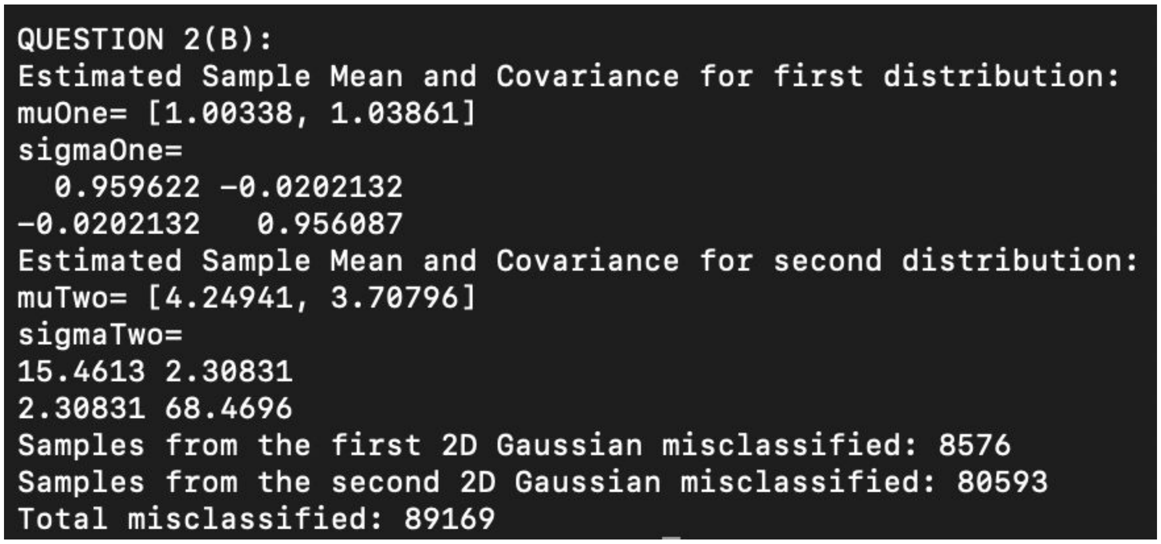


```
QUESTION 2(A):
Estimated Sample Mean and Covariance for first distribution:
muOne= [1.00026, 1.00489]
sigmaOne=
  0.991902 0.00156785
0.00156785   0.993429
Estimated Sample Mean and Covariance for second distribution:
muTwo= [4.01599, 3.97595]
sigmaTwo=
  15.8857 0.0297443
0.0297443   64.0993
Samples from the first 2D Gaussian misclassified: 1952
Samples from the second 2D Gaussian misclassified: 8014
Total misclassified: 9966
```

Figure 7: Output in terminal that displays ML estimations of samples from Project 1, question 2. The mu and sigma for both distributions are estimated, and then classified.

Question 2(b)

Question 2(b) randomly selects 1,000 samples from each sample distributions of 100,000 (hundredth of each sample distribution), then estimates the parameters of the subset of samples using ML estimations. The output for Question 2(b) parameter estimations and classification performance is shown in Figure 8.



```
QUESTION 2(B):
Estimated Sample Mean and Covariance for first distribution:
muOne= [1.00338, 1.03861]
sigmaOne=
  0.959622 -0.0202132
-0.0202132   0.956087
Estimated Sample Mean and Covariance for second distribution:
muTwo= [4.24941, 3.70796]
sigmaTwo=
15.4613 2.30831
2.30831 68.4696
Samples from the first 2D Gaussian misclassified: 8576
Samples from the second 2D Gaussian misclassified: 80593
Total misclassified: 89169
```

Figure 8: Output in terminal that displays ML estimations of the random 1/100 of samples from Project 1, question 2. The mu and sigma for both distributions are estimated, and then classified.

Question 2 Analysis

Table 2 summarizes the misclassification performance for the estimations in 2(a) and 2(b), compared to the performance when the distribution parameters are known in PA1. The estimated samples in 2(a) perform better than PA1 when the parameters were known. This again confirms that having plenty of samples is key in ML estimation. The estimations in this entire question for sigma two were incredibly off, and this shows to be especially detrimental in 2(b) classification when combined with a small sample set.

	Known Parameters PA1	Estimated Parameters 2(a)	Estimated Parameters 2(b)
Misclassified Sample One	3701	1952	8576
Misclassified Sample Two	8078	8014	80593
Total Misclassified	11779	9966	89169

Table 2: Misclassifications of Project 1 when sample distribution is known, 2(a) where sample distribution is estimated, and 2(b) where sample subset distribution is estimated.

Question 3

Question 3 used ML estimation from questions 1 and 2 to implement skin detection as proposed in A Real Time Face Tracker [Yang96]. Skin detection using this methodology was performed on two ppm images, and the ML estimates were calculated using another separate ppm image. Thresholding the normalized discriminant function described in equation 4 allowed the classifier to make a decision of skin or non skin for each pixel in the image. Varying this threshold and then performing the classification generated different levels of false negatives and false positives. Plotting the results of the varied threshold against each other we generated the ROC curves.

When analyzing the curves, classification results vary directly with the chosen threshold. In using the curves we can find the threshold that provides a minimized number of false negative and false positive classified pixels. It is also found that conversion to the YCrCb color space provides much more accurate classification results when compared to normalized RGB. This is because all luminance values which often distort the distribution of skin color are discarded when using YCrCb values for threshold classification.

Program Listings

Parts-1-2.cpp

```
// Libraries used
#include <iostream>
#include <Eigen/Dense>
#include <vector>
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace Eigen;
using namespace std;

// Files used
#include "Classifier.h"
#include "MLE.h"

// Return random index (for 1/100 samples)
int randIndex(int size)
{
	return (rand() / (float)RAND_MAX) * size;
}

int main()
{
	srand(time(NULL));

	// Vectors used
	vector<Vector2f> one;
	vector<Vector2f> two;
	vector<Vector2f> misclassified;

	// Sigma/Mu
	Matrix2f sigmaOne;
	Vector2f muOne;
	Matrix2f sigmaTwo;
	Vector2f muTwo;

	// Estimated Sigma/Mu
	Matrix2f estimatedSigOne;
	Vector2f estimatedMuOne;
	Matrix2f estimatedSigTwo;
	Vector2f estimatedMuTwo;

	// For generations
```

```
Classifier generator;

// ** QUESTION 1 SAMPLE GENERATION **

// Set up parameters
muOne << 1.0, 1.0;
sigmaOne << 1.0, 0.0, 0.0, 1.0;
muTwo << 4.0, 4.0;
sigmaTwo << 1.0, 0.0, 0.0, 1.0;

// Variables
int misclassOne = 0;
int misclassTwo = 0;

// Let's make 'em
one = generator.generateSamples(muOne, sigmaOne);
two = generator.generateSamples(muTwo, sigmaTwo);

// For reference since not saved from PA1
for(int i = 0; i < 100000; i++)
{
        if(Classifier::caseOne(one[i], muOne, muTwo, sigmaOne(0,0), sigmaTwo(0,0)) == 2)
        {
                misclassOne++;
                misclassified.push_back(one[i]);
        }
        if(Classifier::caseOne(two[i], muOne, muTwo, sigmaOne(0,0), sigmaTwo(0,0)) == 1)
        {
                misclassTwo++;
                misclassified.push_back(two[i]);
        }
}

// Output to terminal
cout << "QUESTION 1 REFERENCE FROM PA1:" << endl;
cout << "Samples from the first 2D Gaussian misclassified: " << misclassOne << endl;
cout << "Samples from the second 2D Gaussian misclassified: " << misclassTwo << endl;
cout << "Total misclassified: " << misclassOne + misclassTwo << endl;

// ** QUESTION 1(A) **

// Estimate stats
estimatedMuOne = MLE::sampleMean(one);
estimatedMuTwo = MLE::sampleMean(two);
estimatedSigOne = MLE::sampleCovariance(one, estimatedMuOne);
estimatedSigTwo = MLE::sampleCovariance(two, estimatedMuTwo);

// Reconfigure variables
```

```
	misclassified.clear();
	misclassOne = 0;
	misclassTwo = 0;

	// Classify based on MLE
	for(int i = 0; i < 100000; i++)
	{
		if(Classifier::caseOne(one[i], estimatedMuOne, estimatedMuTwo, estimatedSigOne(0,0),
estimatedSigTwo(0,0)) == 2)
		{
			misclassOne++;
			misclassified.push_back(one[i]);
		}
		if(Classifier::caseOne(two[i], estimatedMuOne, estimatedMuTwo, estimatedSigOne(0,0),
estimatedSigTwo(0,0)) == 1)
		{
			misclassTwo++;
			misclassified.push_back(two[i]);
		}
	}

	// Output the estimates/stats
	cout << "\nQUESTION 1(A):" << endl;
	cout << "Estimated Sample Mean and Covariance for first distribution: \nmuOne= [" <<
estimatedMuOne(0) << ", " << estimatedMuOne(1) << "]" << endl;
	cout << "sigmaOne=" << endl;
	cout << estimatedSigOne << endl;
	cout << "Estimated Sample Mean and Covariance for second distribution: \nmuTwo= [" <<
estimatedMuTwo(0) << ", " << estimatedMuTwo(1) << "]" << endl;
	cout << "sigmaTwo=" << endl;
	cout << estimatedSigTwo << endl;
	cout << "Samples from the first 2D Gaussian misclassified: " << misclassOne << endl;
	cout << "Samples from the second 2D Gaussian misclassified: " << misclassTwo << endl;
	cout << "Total misclassified: " << misclassOne + misclassTwo << endl;

	// ** QUESTION 1(B) **

	// 1/100 of samples
	vector<Vector2f> smallone;
	vector<Vector2f> smalltwo;

	// Randomly select from one and two for smaller sample size
	for(int i = 0; i < 1000; i++)
	{
		int idxOne = randIndex(one.size());
		int idxTwo = randIndex(two.size());
		smallone.push_back(one[idxOne]); one.erase(one.begin() + idxOne);
		smalltwo.push_back(two[idxTwo]); two.erase(two.begin() + idxTwo);
	}

	// Estimate stats using small sample size
	estimatedMuOne = MLE::sampleMean(smallone);
	estimatedMuTwo = MLE::sampleMean(smalltwo);
	estimatedSigOne = MLE::sampleCovariance(smallone, estimatedMuOne);
	estimatedSigTwo = MLE::sampleCovariance(smalltwo, estimatedMuTwo);

	// Reconfigure variables
	misclassified.clear();
	misclassOne = 0;
	misclassTwo = 0;

	// Classify based on MLE
	for(int i = 0; i < 100000; i++)
	{
		if(Classifier::caseOne(one[i], estimatedMuOne, estimatedMuTwo, estimatedSigOne(0,0),
estimatedSigTwo(0,0)) == 2)
		{
			misclassOne++;
			misclassified.push_back(one[i]);
		}
		if(Classifier::caseOne(two[i], estimatedMuOne, estimatedMuTwo, estimatedSigOne(0,0),
estimatedSigTwo(0,0)) == 1)
		{
			misclassTwo++;
			misclassified.push_back(two[i]);
		}
	}

	// Output the estimates/stats
	cout << "\nQUESTION 1(B):" << endl;
	cout << "Estimated Sample Mean and Covariance for first distribution: \nmuOne= [" <<
estimatedMuOne(0) << ", " << estimatedMuOne(1) << "]" << endl;
	cout << "sigmaOne=" << endl;
	cout << estimatedSigOne << endl;
	cout << "Estimated Sample Mean and Covariance for second distribution: \nmuTwo= [" <<
estimatedMuTwo(0) << ", " << estimatedMuTwo(1) << "]" << endl;
	cout << "sigmaTwo=" << endl;
	cout << estimatedSigTwo << endl;
	cout << "Samples from the first 2D Gaussian misclassified: " << misclassOne << endl;
	cout << "Samples from the second 2D Gaussian misclassified: " << misclassTwo << endl;
	cout << "Total misclassified: " << misclassOne + misclassTwo << endl;

	// ** QUESTION 2 SAMPLE GENERATION **

	// Set up parameters
	muOne << 1.0, 1.0;
```

```
sigmaOne << 1.0, 0.0, 0.0, 1.0;
muTwo << 4.0, 4.0;
sigmaTwo << 4.0, 0.0, 0.0, 8.0;

// Reconfigure variables
misclassOne = 0;
misclassTwo = 0;
misclassified.clear();

// Let's make 'em
one = generator.generateSamples(muOne, sigmaOne);
two = generator.generateSamples(muTwo, sigmaTwo);

// For reference since since not saved from PA1
for(int i = 0; i < 100000; i++)
{
        if(Classifier::caseThree(one[i], muOne, muTwo, sigmaOne, sigmaTwo) == 2)
        {
                misclassOne++;
                misclassified.push_back(one[i]);
        }
        if(Classifier::caseThree(two[i], muOne, muTwo, sigmaOne, sigmaTwo) == 1)
        {
                misclassTwo++;
                misclassified.push_back(two[i]);
        }
}

// Output to terminal
cout << "\n\nQUESTION 2 REFERENCE FROM PA1:" << endl;
cout << "Samples from the first 2D Gaussian misclassified: " << misclassOne << endl;
cout << "Samples from the second 2D Gaussian misclassified: " << misclassTwo << endl;
cout << "Total misclassified: " << misclassOne + misclassTwo << endl;

// ** QUESTION 2(A) **

// Estimate stats
estimatedMuOne = MLE::sampleMean(one);
estimatedMuTwo = MLE::sampleMean(two);
estimatedSigOne = MLE::sampleCovariance(one, estimatedMuOne);
estimatedSigTwo = MLE::sampleCovariance(two, estimatedMuTwo);

// Reconfigure variables
misclassified.clear();
misclassOne = 0;
misclassTwo = 0;

// Classify based on MLE
```

```
	for(int i = 0; i < 100000; i++)
	{
		if(Classifier::caseThree(one[i], estimatedMuOne, estimatedMuTwo, estimatedSigOne, 
estimatedSigTwo) == 2)
		{
			misclassOne++;
			misclassified.push_back(one[i]);
		}
		if(Classifier::caseThree(two[i], estimatedMuOne, estimatedMuTwo, estimatedSigOne, 
estimatedSigTwo) == 1)
		{
			misclassTwo++;
			misclassified.push_back(two[i]);
		}
	}

	// Output the estimates/stats
	cout << "\nQUESTION 2(A):" << endl;
	cout << "Estimated Sample Mean and Covariance for first distribution: \nmuOne= [" << 
estimatedMuOne(0) << ", " << estimatedMuOne(1) << "]" << endl;
	cout << "sigmaOne=" << endl;
	cout << estimatedSigOne << endl;
	cout << "Estimated Sample Mean and Covariance for second distribution: \nmuTwo= [" << 
estimatedMuTwo(0) << ", " << estimatedMuTwo(1) << "]" << endl;
	cout << "sigmaTwo=" << endl;
	cout << estimatedSigTwo << endl;
	cout << "Samples from the first 2D Gaussian misclassified: " << misclassOne << endl;
	cout << "Samples from the second 2D Gaussian misclassified: " << misclassTwo << endl;
	cout << "Total misclassified: " << misclassOne + misclassTwo << endl;

	// ** QUESTION 2(B) **

	// Reconfigure variables
	smallone.clear();
	smalltwo.clear();

	// Generate 1/100 of samples
	for(int i = 0; i < 1000; i++)
	{
		int idxOne = randIndex(one.size());
		int idxTwo = randIndex(two.size());
		smallone.push_back(one[idxOne]); one.erase(one.begin() + idxOne);
		smalltwo.push_back(two[idxTwo]); two.erase(two.begin() + idxTwo);
	}

	// Estimate stats based on small sample size
	estimatedMuOne = MLE::sampleMean(smallone);
	estimatedMuTwo = MLE::sampleMean(smalltwo);
```

```
		estimatedSigOne = MLE::sampleCovariance(smallone, estimatedMuOne);
		estimatedSigTwo = MLE::sampleCovariance(smalltwo, estimatedMuTwo);

		// Reconfigure variables
		misclassified.clear();
		misclassOne = 0;
		misclassTwo = 0;

		// Classify based on MLE
		for(int i = 0; i < 100000; i++)
		{
			if(Classifier::caseOne(one[i], estimatedMuOne, estimatedMuTwo, estimatedSigOne(0,0),
estimatedSigTwo(0,0)) == 2)
			{
				misclassOne++;
				misclassified.push_back(one[i]);
			}
			if(Classifier::caseOne(two[i], estimatedMuOne, estimatedMuTwo, estimatedSigOne(0,0),
estimatedSigTwo(0,0)) == 1)
			{
				misclassTwo++;
				misclassified.push_back(two[i]);
			}
		}

		// Output the estimates/stats
		cout << "\nQUESTION 2(B):" << endl;
		cout << "Estimated Sample Mean and Covariance for first distribution: \nmuOne= [" <<
estimatedMuOne(0) << ", " << estimatedMuOne(1) << "]" << endl;
		cout << "sigmaOne=" << endl;
		cout << estimatedSigOne << endl;
		cout << "Estimated Sample Mean and Covariance for second distribution: \nmuTwo= [" <<
estimatedMuTwo(0) << ", " << estimatedMuTwo(1) << "]" << endl;
		cout << "sigmaTwo=" << endl;
		cout << estimatedSigTwo << endl;
		cout << "Samples from the first 2D Gaussian misclassified: " << misclassOne << endl;
		cout << "Samples from the second 2D Gaussian misclassified: " << misclassTwo << endl;
		cout << "Total misclassified: " << misclassOne + misclassTwo << endl;
}
```

MLE.cpp

```
// Libraries used
#include "MLE.h"
#include <iostream>

// Sample mean
Vector2f MLE::sampleMean(vector<Vector2f> data)
```

```
{
	Vector2f sum;
	sum << 0.0, 0.0;
	for(vector<int>::size_type i = 0; i < data.size(); i++)
	{
		sum += data[i];
	}
	return sum / data.size();
}

// Covariance
Matrix2f MLE::sampleCovariance(vector<Vector2f> data, Vector2f sampleMean)
{
	Matrix2f sum;
	sum <<		0.0, 0.0, 0.0, 0.0;
	for(vector<int>::size_type i = 0; i <  data.size(); i++)
	{
		sum += (sampleMean - data[i])*((sampleMean - data[i]).transpose());
	}
	return sum / data.size();
}
```

MLE.h

```
#ifndef MLE_H
#define MLE_H

// Libraries used
#include <Eigen/Dense>
#include <vector>
using namespace std;
using namespace Eigen;

class MLE
{
public:
	// Sample mean
	static Vector2f sampleMean(vector<Vector2f>);

	// Sample covariance
	static Matrix2f sampleCovariance(vector<Vector2f>, Vector2f);
};
#endif
```

Threshold Function Added to Classifiers.cpp

```
// Threshold Bayes
bool Classifier::thresholdCaseThree(Vector2f x, Vector2f mu, Matrix2f sigma, float threshold)
{
        float p = exp((-0.5 * (x - mu).transpose() * sigma.inverse() * (x - mu)));
        return (p > threshold);
}
```

Part-3.cpp

```
// Libraries used
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <Eigen/Dense>
#include <vector>
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace Eigen;
using namespace std;
using namespace cv;
// Files used
#include "Classifier.h"
#include "MLE.h"

const char* original_window = "Original Image";
const char* reference_window = "Reference image";
const char* classified_window = "Classified Image";

struct MLStats{
   Vector2f muSkin;
   Matrix2f varianceSkin;
   Vector2f muSkinYCC;
   Matrix2f varianceSkinYCC;
};

void calculate_ML_image(Mat& training_image, Mat& reference_image, MLStats &stats);
void threshold_classification(Mat& test_image, Mat& reference_image, Mat& classified_image,
      MLStats &stats, float thresMin, float thresMax, const char filename[][20]);

int main()
{
   Mat classifiedImage;
   // Loading in images to perform the classification on
   Mat training_image = imread("../P2_Data/Data_Proj2/Training_1.ppm", IMREAD_COLOR);
   Mat test_image_1 = imread("../P2_Data/Data_Proj2/Training_3.ppm", IMREAD_COLOR);
   Mat test_image_2 = imread("../P2_Data/Data_Proj2/Training_6.ppm", IMREAD_COLOR);
   // Loading in defined reference images for training of the classifier and comparison
   // of results against the classified results. For each pixel in the image, it has been
   // manually determined which pixel is a skin pixel.
   Mat reference_image_tr = imread("../P2_Data/Data_Proj2/ref1.ppm", IMREAD_COLOR);
```

```
    Mat reference_image_1 = imread("../P2_Data/Data_Proj2/ref3.ppm", IMREAD_COLOR);
    Mat reference_image_2 = imread("../P2_Data/Data_Proj2/ref6.ppm", IMREAD_COLOR);
    // are calculated using calculateMLSkin
    MLStats statsML;
    calculate_ML_image(training_image, reference_image_tr, statsML);
    const char filenametr3[2][20] = {"train3_RGB_ROC.txt", "train3_YCC_ROC.txt"};
    const char filenametr6[2][20] = {"train6_RGB_ROC.txt", "train6_YCCS_ROC.txt"};
    threshold_classification(test_image_1, reference_image_1, classifiedImage, statsML, -1, 0, filenametr3);
    threshold_classification(test_image_1, reference_image_1, classifiedImage, statsML, -1, 0, filenametr6);
    return 0;
}


void calculate_ML_image(Mat& training_image, Mat& reference_image, MLStats &stats)
{
    vector<Vector2f> skinSamplesRGB, nonSkinSamplesRGB, skinSamplesYCC, nonSkinSamplesYCC;
    float totalRGB, normR, normG, normCb, normCr;
    totalRGB = normR = normG = normCb = normCr = 0;
    for (int i = 0; i < training_image.rows; i++) {
        for (int j = 0; j < training_image.cols; j++) {
            // using RGB color space for calculations
            totalRGB = (float)training_image.at<Vec3b>(i,j)[0] +
                    (float)training_image.at<Vec3b>(i,j)[1] +
                    (float)training_image.at<Vec3b>(i,j)[2];
            if (totalRGB != 0) {
                normR = (float)training_image.at<Vec3b>(i,j)[2] / totalRGB;
                normG = (float)training_image.at<Vec3b>(i,j)[1] / totalRGB;
            }
            // using YCC color space for calculations
            normCb = -0.169 * (float)training_image.at<Vec3b>(i,j)[2] -
                    0.332 * (float)training_image.at<Vec3b>(i,j)[1] +
                    0.5 * (float)training_image.at<Vec3b>(i,j)[0];
            normCr = 0.5 * (float)training_image.at<Vec3b>(i,j)[2] -
                    0.419 * (float)training_image.at<Vec3b>(i,j)[1] -
                    0.081 * (float)training_image.at<Vec3b>(i,j)[0];
            if ( ( (float)reference_image.at<Vec3b>(i,j)[2] ) != 0 &&
                 ( (float)reference_image.at<Vec3b>(i,j)[1] ) != 0 &&
                 ( (float)reference_image.at<Vec3b>(i,j)[0] ) != 0 != 0 ) {
                skinSamplesRGB.push_back(Vector2f(normR, normG));
                skinSamplesYCC.push_back(Vector2f(normCb, normCr));
            }
        }
    }
    stats.muSkin = MLE::sampleMean(skinSamplesRGB);
    stats.varianceSkin = MLE::sampleCovariance(skinSamplesRGB, stats.muSkin);
    stats.muSkinYCC = MLE::sampleMean(skinSamplesYCC);
    stats.varianceSkinYCC = MLE::sampleCovariance(skinSamplesYCC, stats.muSkinYCC);
}
```

```
void threshold_classification(Mat& test_image, Mat& reference_image, Mat& classified_image, MLStats
&stats, float thresMin, float thresMax, const char filename[][20])
{
    vector <float> falseNegatives, falsePositives, falseNegativesYCC, falsePositivesYCC;
    float totalRGB, normR, normG, normCb, normCr;
    float skinTotal, nonSkinTotal, falseNegative, falsePositive,
            skinTotalYCC, nonSkinTotalYCC, falseNegativeYCC, falsePositiveYCC;
    bool classifiedAsSkin, isSkin;
    totalRGB = normR = normG = normCb = normCr = skinTotalYCC = nonSkinTotalYCC =
             falseNegativeYCC = falsePositiveYCC = 0;
    for(float threshold = thresMin; threshold <= thresMax+0.02; threshold+=.05) {
        skinTotal = nonSkinTotal = falseNegative = falsePositive = 0;
        for (int i = 0; i < test_image.rows; i++) {
            for (int j = 0; j < test_image.cols; j++) {
                // using RGB color space for calculations
                totalRGB = (float) test_image.at<Vec3b>(i, j)[0] +
                           (float) test_image.at<Vec3b>(i, j)[1] +
                           (float) test_image.at<Vec3b>(i, j)[2];
                if (totalRGB != 0) {
                    normR = (float) test_image.at<Vec3b>(i, j)[2] / totalRGB;
                    normG = (float) test_image.at<Vec3b>(i, j)[1] / totalRGB;
                }
                // using YCC color space for calculations
                normCb = -0.169 * (float) test_image.at<Vec3b>(i, j)[2] -
                         0.332 * (float) test_image.at<Vec3b>(i, j)[1] +
                         0.5 * (float) test_image.at<Vec3b>(i, j)[0];
                normCr = 0.5 * (float) test_image.at<Vec3b>(i, j)[2] -
                         0.419 * (float) test_image.at<Vec3b>(i, j)[1] -
                         0.081 * (float) test_image.at<Vec3b>(i, j)[0];
                classifiedAsSkin = Classifier::thresholdCaseThree(Vector2f(normR, normG), stats.muSkinYCC,
stats.varianceSkinYCC, threshold);

                isSkin = ( (float)reference_image.at<Vec3b>(i,j)[2] ) != 0 &&
                         ( (float)reference_image.at<Vec3b>(i,j)[1] ) != 0 &&
                         ( (float)reference_image.at<Vec3b>(i,j)[0] ) != 0;

                if(classifiedAsSkin) {
                    skinTotal++;
                }
                else {
                    nonSkinTotal++;
                }

                if(isSkin && !classifiedAsSkin) {
                    falseNegative++;
                }
                else if(!isSkin && classifiedAsSkin) {
                    falsePositive++;
                }
                // YCC classification
                classifiedAsSkin = Classifier::thresholdCaseThree(Vector2f(normCb, normCr), stats.muSkin,
stats.varianceSkin, threshold);

                if(classifiedAsSkin) {
                    skinTotalYCC++;
                }
                else {
                    nonSkinTotalYCC++;
                }
                if(isSkin && !classifiedAsSkin) {
                    falseNegativeYCC++;
                }
                else if(!isSkin && classifiedAsSkin) {
                    falsePositiveYCC++;
                }

            }
        }
        falseNegatives.push_back(falseNegative / nonSkinTotal);
        falsePositives.push_back(falsePositive / skinTotal);
        falseNegativesYCC.push_back(falseNegativeYCC / nonSkinTotalYCC);
        falsePositivesYCC.push_back(falsePositiveYCC / skinTotalYCC);
        /*
        cout << "Threshold: " << threshold << ": " << endl;
        cout << falsePositive << endl;
        cout << skinTotal << endl;
        cout << "\tFalse Negative Rate: \t" << falseNegative / nonSkinTotal << endl;
        cout << "\tFalse Positive Rate: \t" << falsePositive / skinTotal << endl;
        cout << "Threshold: " << threshold << ": " << endl;
        cout << falsePositiveYCC << endl;
        cout << skinTotalYCC << endl;
        cout << "\tFalse Negative Rate: \t" << falseNegativeYCC / nonSkinTotalYCC << endl;
        cout << "\tFalse Positive Rate: \t" << falsePositiveYCC / skinTotalYCC << endl;
        */
    }
    ofstream outputFile;

    outputFile.open(filename[0]);

    outputFile << "Threshold\tFalseNegative\tFalsePositive" << endl;

    for(float threshold = thresMin, i = 0; threshold <= thresMax+0.02; threshold+=.05, i++)
    {
        outputFile << threshold << "\t" << falseNegatives[i] << "\t" << falsePositives[i] << endl;
```

```
    }

    outputFile.close();

    outputFile.open(filename[1]);

    outputFile << "Threshold\tFalseNegative\tFalsePositive" << endl;

    for(float threshold = thresMin, i = 0; threshold <= thresMax+0.02; threshold+=.05, i++)
    {
        outputFile << threshold << "\t" << falseNegativesYCC[i] << "\t" << falsePositivesYCC[i] << endl;
    }

    outputFile.close();
}
```