GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES AND TECHNOLOGY – GIKI

FACULTY OF COMPUTER SCIENCE AND ENGINEERING -FCSE

CLUSTERING IN MACHINE LEARNING

UNSUPERVISED LEARNING IN CPP USING OOP AND GUI

Instructor:

Dr. Prof. Zahid Haleem

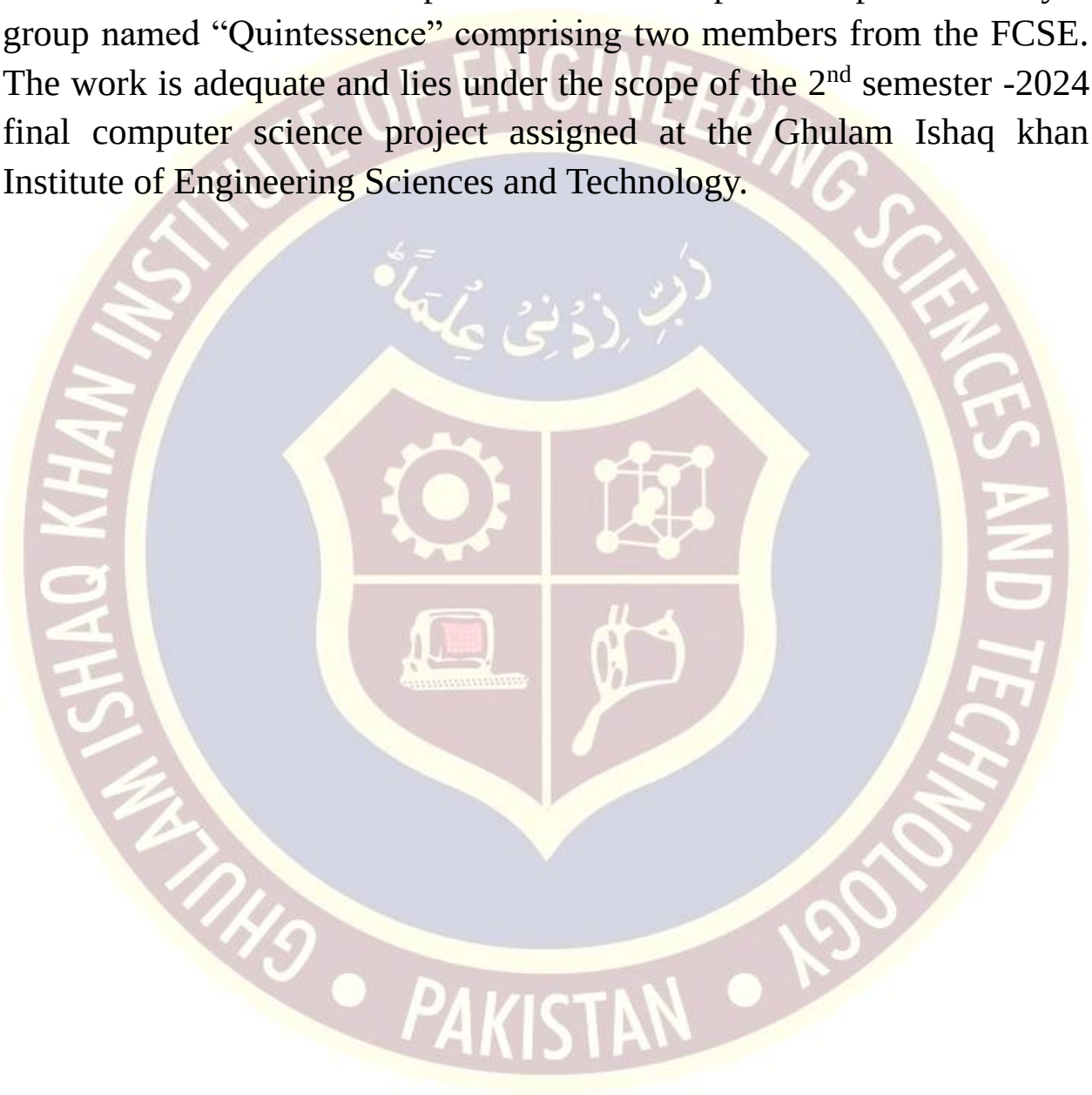
Group Members – QUINTESSENCE:

Muhammad Haris - 2023428

Mumtaz Ali - 2023559

CERTIFICATE OF RECOGNITION:

It is certified that the work presented in this report was performed by a group named “Quintessence” comprising two members from the FCSE. The work is adequate and lies under the scope of the 2nd semester -2024 final computer science project assigned at the Ghulam Ishaq khan Institute of Engineering Sciences and Technology.



Instructor: Examiner:

_____ _____

ABSTRACT:

This project implements various data analysis techniques within a Windows Forms application using C++. The application provides functionalities for loading data from a text file, calculating correlation matrices, discretizing data, displaying bitmaps, shuffling datasets, and applying unsupervised machine learning concepts like Signature Technique. It leverages concepts of statistics, linear algebra, and basic machine learning algorithms to perform data analysis tasks. The project aims to offer a practical tool for exploring datasets and understanding their relationships through visualization and manipulation.

ACKNOWLEDGEMENT:

We cannot express enough thanks to our instructor Dr. Prof Zahid Haleem for his support and encouragement. We offer our sincere appreciation for such learning opportunities provided to students by the Ghulam Ishaq Khan Institute.

TABLE OF CONTENTS:

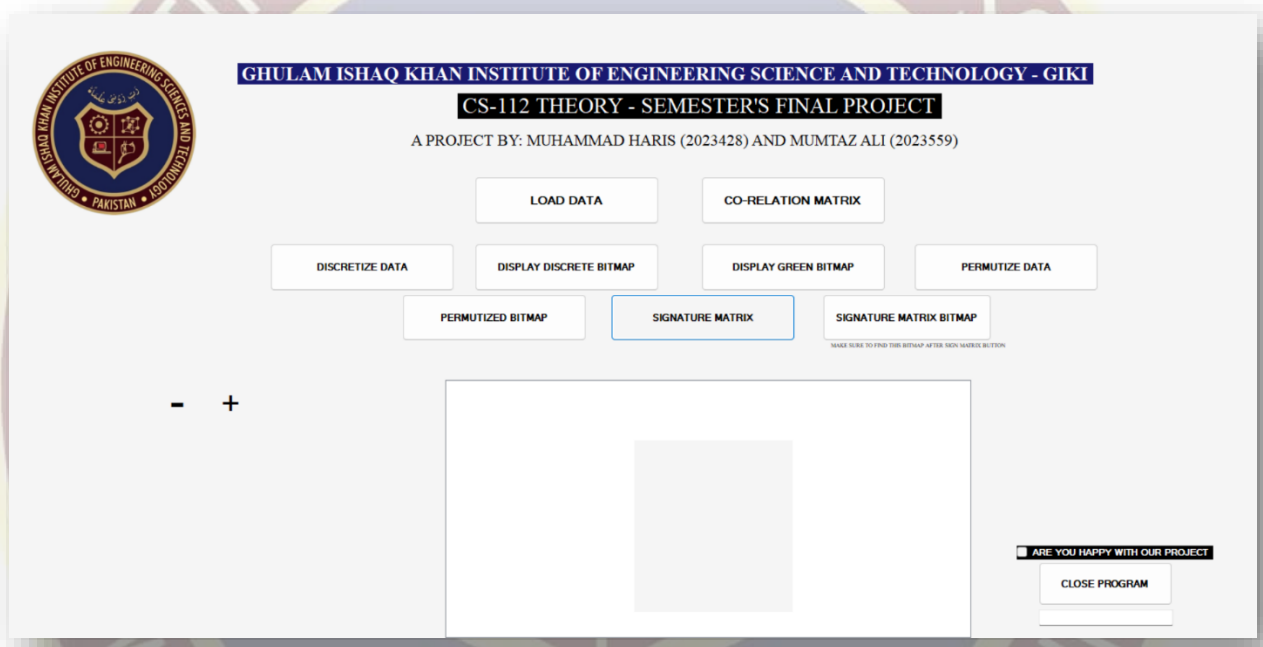
1. INTRODUCTION:

Data analysis plays a pivotal role in various fields ranging from scientific research to business decision-making. Understanding and extracting insights from data are essential for making informed choices and driving progress. In recent years, the proliferation of data collection mechanisms has led to an exponential increase in the volume and complexity of datasets. Consequently, there is a growing demand for efficient tools and techniques to analyze and interpret this wealth of information.

In this regard, the project encompasses the implementation of several key functionalities, including data loading from text files, calculation of correlation matrices, discretization of data, display of bitmap images, dataset shuffling, and the application of unsupervised machine learning algorithms such as the Signature Technique. These functionalities are supported by underlying concepts from statistics, linear algebra, and basic machine learning, enabling users to explore datasets comprehensively and extract meaningful insights.

- Underlying Concepts
  - Statistics
  - Linear Algebra
  - Basic Machine Learning Algorithms

2. PROJECT STRUCTURE:



LOADING DATA:

The data set used in this project is taken from the well-known dataset providing source named UCI repository. The data set is specifically named as the iris data set and can be accessed from the following link:

http://archive.ics.uci.edu/ml/datasets/Iris

CHAPTER NO:1

TASK 1

Button 1: LOAD DATA:

DESCRIPTION:

This button retrieves data from a text file named "abc.txt" and displays it in a list box within the application.

Steps:

1. The button is clicked by the user.
2. The application reads the contents of the text file "abc.txt".
3. It parses the first two lines of the file to determine the dimensions of the data (number of equations and variables).
4. It then reads the remaining lines of the file, each representing a row of data.
5. Each line of data is added as an item in the list box for display.
6. The data loading process is completed, and the list box now contains the loaded data.

Loading Function:

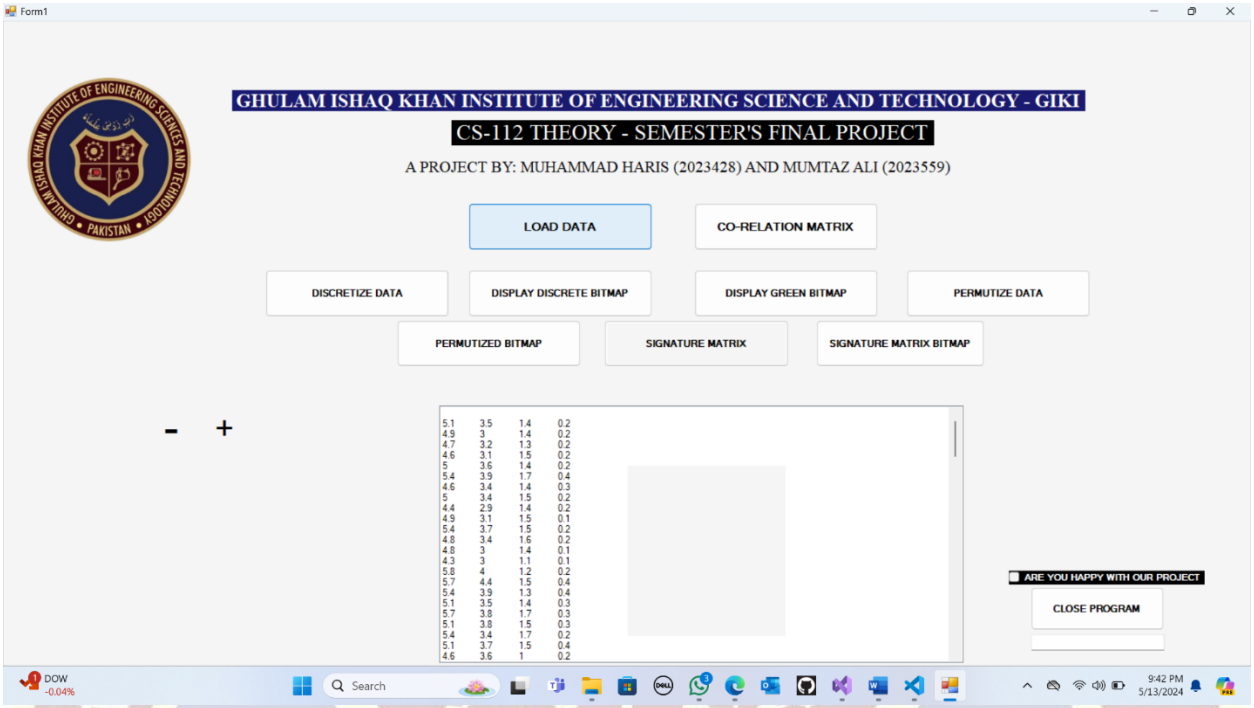
```
array<array<double>^>^ loadData() {
    StreamReader^ sr = File::OpenText("abc.txt");
    int eq = int::Parse(sr->ReadLine());
    int var = int::Parse(sr->ReadLine());
    array<array<double>^>^ arr = gcnew array<array<double>^>(eq);
    for (int i = 0; i < eq; ++i)
        arr[i] = gcnew array<double>(var);
    array<array<double>^>^ arra = gcnew array<array<double>^>(var);
    for (int i = 0; i < var; ++i)
        arra[i] = gcnew array<double>(eq);
```

```
    String^ lineo = sr->ReadLine();
    while (lineo != nullptr)
    {
        lineo = sr->ReadLine();
    }
    sr->Close();

    StreamReader^ sa = File::OpenText("abc.txt");

    String^ dummy1 = sa->ReadLine();
    String^ dummy2 = sa->ReadLine();
    sa->ReadLine();
    String^ line;
    for (int row = 0; row < eq; ++row)
    {
        line = sa->ReadLine();
        if (line != nullptr) // Checking if line is not null before splitting
        {
            array<String^>^ numbers = line->Split('\t');
            for (int col = 0; col < var; ++col)
            {
                arr[row][col] = double::Parse(numbers[col]);
                arra[col][row] = arr[row][col];
            }
        }
    }
    return arr;
}
```

Actual Button:

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
    listBox1->Items->Clear();
    StreamReader^ sr = File::OpenText("abc.txt");
    int eq = int::Parse(sr->ReadLine());
    int var = int::Parse(sr->ReadLine());

    listBox1->Items->Clear(); // Clear the list box before displaying new content

    String^ line;
    while ((line = sr->ReadLine()) != nullptr)
    {listBox1->Items->Add(line);}
    sr->Close();
}
```

Screenshot:



Button 2: CO-RELATION MATRIX:

DESCRIPTION:

This button calculates the correlation matrix of the loaded data and presents it in the list box for visualization.

FUNCTIONALITY:

1. The button is clicked by the user.
2. The application retrieves the data loaded from the text file.
3. It calculates the correlation matrix using a mathematical formula.
4. For each pair of variables, it computes the correlation coefficient, which represents the strength and direction of the linear relationship between them.
5. The correlation coefficients are then added as items in the list box for display.
6. The correlation matrix is now visible to the user in the list box.

These buttons provide essential functionality for loading and analyzing data within the application.

$$r = \frac{\sum(X-\bar{X})(Y-\bar{Y})}{\sqrt{\sum(X-\bar{X})^2}\sqrt{(Y-\bar{Y})^2}}$$

Where, $\bar{X}$ = mean of X variable

$\bar{Y}$ = mean of Y variable

Correlation Matrix Calculation Function:

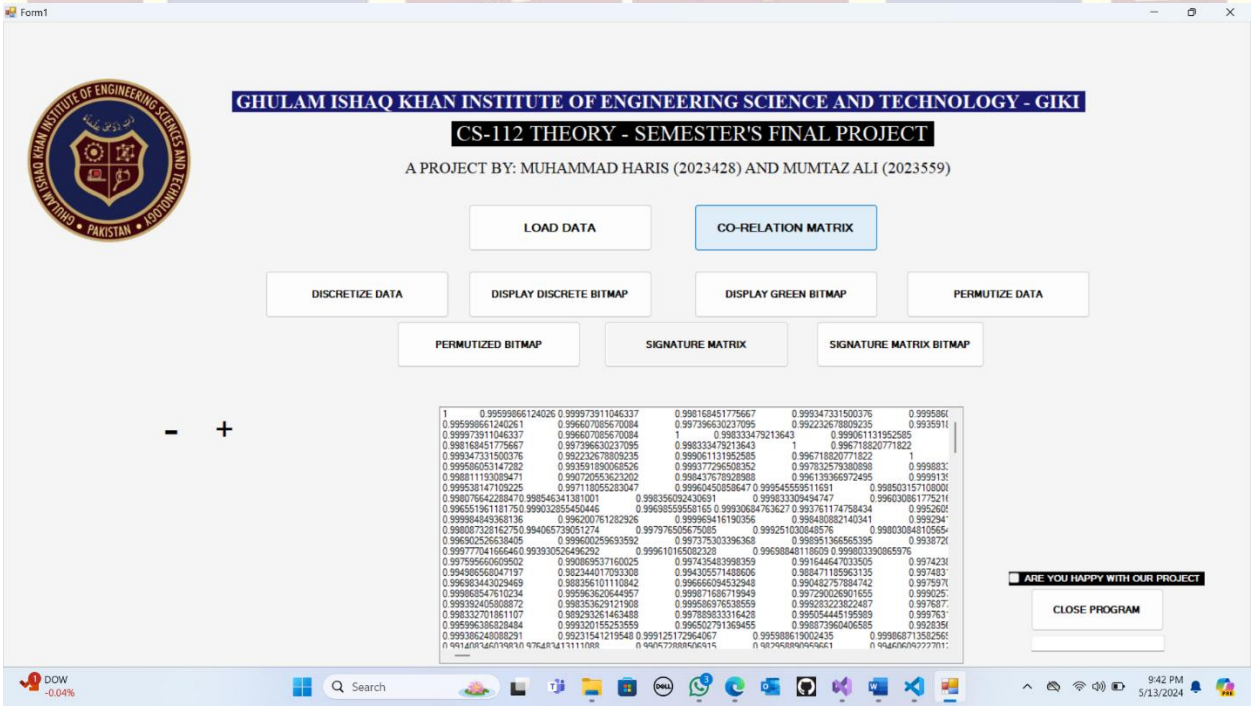
```
array<array<double>^>^ CalculateCorrelationMatrix(array<array<double>^>^ arr) {
    int rows = arr->Length;
    int cols = arr[0]->Length;

    array<array<double>^>^ correlationMatrix = gcnew array<array<double>^>(rows);
    for (int i = 0; i < rows; i++)
        correlationMatrix[i] = gcnew array<double>(rows);


    for (int i = 0; i < rows; i++) {
        for (int j = i; j < rows; j++) {
            double sumX = 0.0;
            double sumY = 0.0;
            for (int k = 0; k < cols; k++) {
                sumX += arr[i][k];
                sumY += arr[j][k];
            }
            double meanX = sumX / cols;
            double meanY = sumY / cols;
            double meanXsum = 0;
            double meanYsum = 0;
            for (int k = 0; k < cols; k++) {
                meanXsum += pow((arr[i][k] - meanX), 2);
                meanYsum += pow((arr[j][k] - meanY), 2);
```

```
            }
            double numerator = 0;
            for (int k = 0; k < cols; k++) {
                numerator += (arr[i][k] - meanX) * (arr[j][k] - meanY);
            }
            double denominator = sqrt((meanXsum) * (meanYsum));
            correlationMatrix[i][j] = numerator / denominator;
            correlationMatrix[j][i] = correlationMatrix[i][j]; // symmetric
matrix
            listBox1->Items->Add(correlationMatrix[i][j]);
        }
    }
    return correlationMatrix;
}
```

Actual Button:

```
private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e)
{    listBox1->Items->Clear();
    array<array<double>^>^ arr = loadData();
    CalculateCorrelationMatrix(arr);}
```

Screenshot:



Button 3: DISCRETIZE DATA

FUNCTIONALITY:

1. Load Data:
2. Calculate Correlation Matrix:
3. Discretize Correlation Matrix:

- Convert the correlation matrix into a binary matrix (0s and 1s).
- Each element in the discretized matrix represents the presence or absence of a relationship between two variables.
- The discretization is based on comparing each correlation value with the mean of all correlation values.
- If a correlation value is greater than or equal to the mean, it is set to 1; otherwise, it is set to 0.

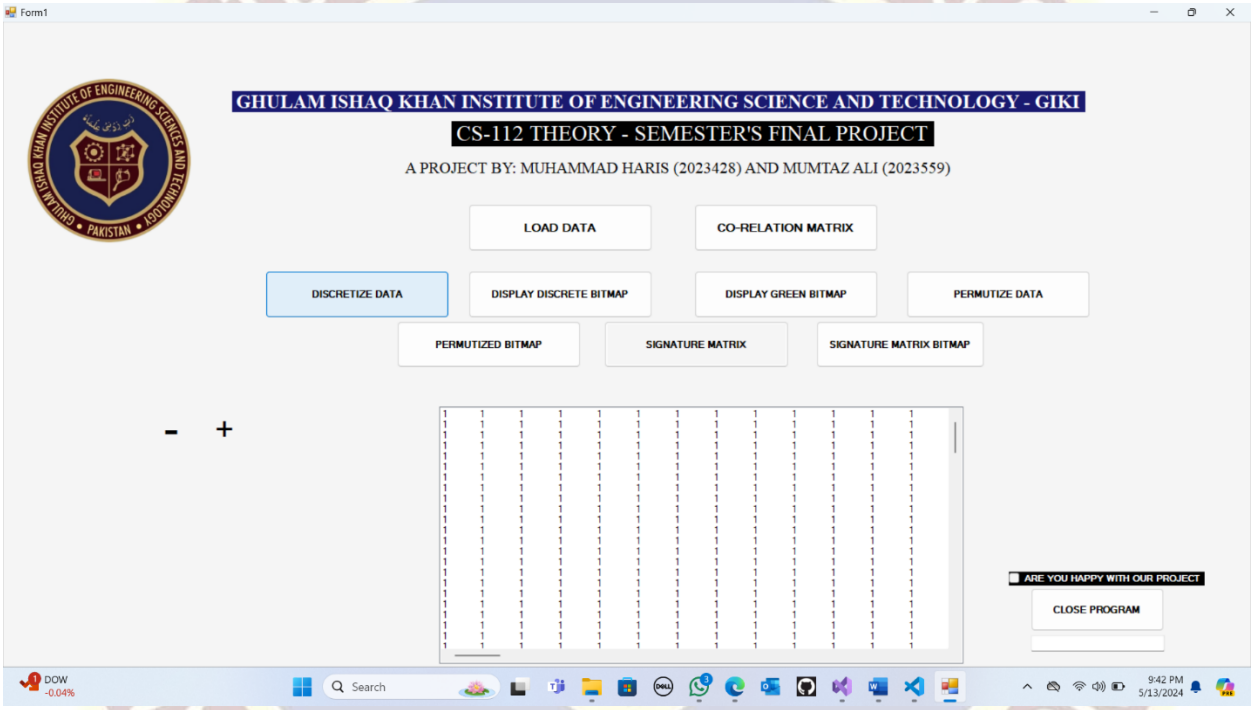
Discretization Function:

```
        void Discretize(array<array<double>^>^ correlationMatrix, bool
displayBitmap) {
            int rows = correlationMatrix->Length;
            array<array<int>^>^ discretizedMatrix = gcnew
array<array<int>^>(rows);

            for (int i = 0; i < rows; i++) {
                discretizedMatrix[i] = gcnew array<int>(rows); // Initialize each
row before assigning values
            }

            for (int i = 0; i < rows; i++) {
                double sum = 0;
                for (int j = 0; j < rows; j++) {
                    sum += correlationMatrix[i][j];
                }
                double mean = sum / rows;
                for (int j = i; j < rows; j++) {
                    if (correlationMatrix[i][j] >= mean)
                        discretizedMatrix[i][j] = 1;
                    else
                        discretizedMatrix[i][j] = 0;
                    // Since it's a symmetric matrix, set the corresponding
element too
                    discretizedMatrix[j][i] = discretizedMatrix[i][j];
                }
            }}
```

Actual Button:

```
private: System::Void button3_Click(System::Object^ sender, System::EventArgs^ e)
{
    array<array<double>^>^ arr = loadData();
    array<array<double>^>^ correlationMatrix = CalculateCorrelationMatrix(arr);
    Discretize(correlationMatrix, false);

}
```

Screenshot:



Button 4: DISPLAY DISCRETE BITMAP

FUNCTIONALITY:

1. Load Data:
2. Calculate Correlation Matrix:
3. Discretize Correlation Matrix:
4. Display Discretized Data:

- Display the discretized correlation matrix in a list box.
- Each row in the list box represents a row in the discretized matrix.

- The binary values (0s and 1s) indicate the presence or absence of relationships between variables.
- This provides an overview of the significant relationships within the dataset.

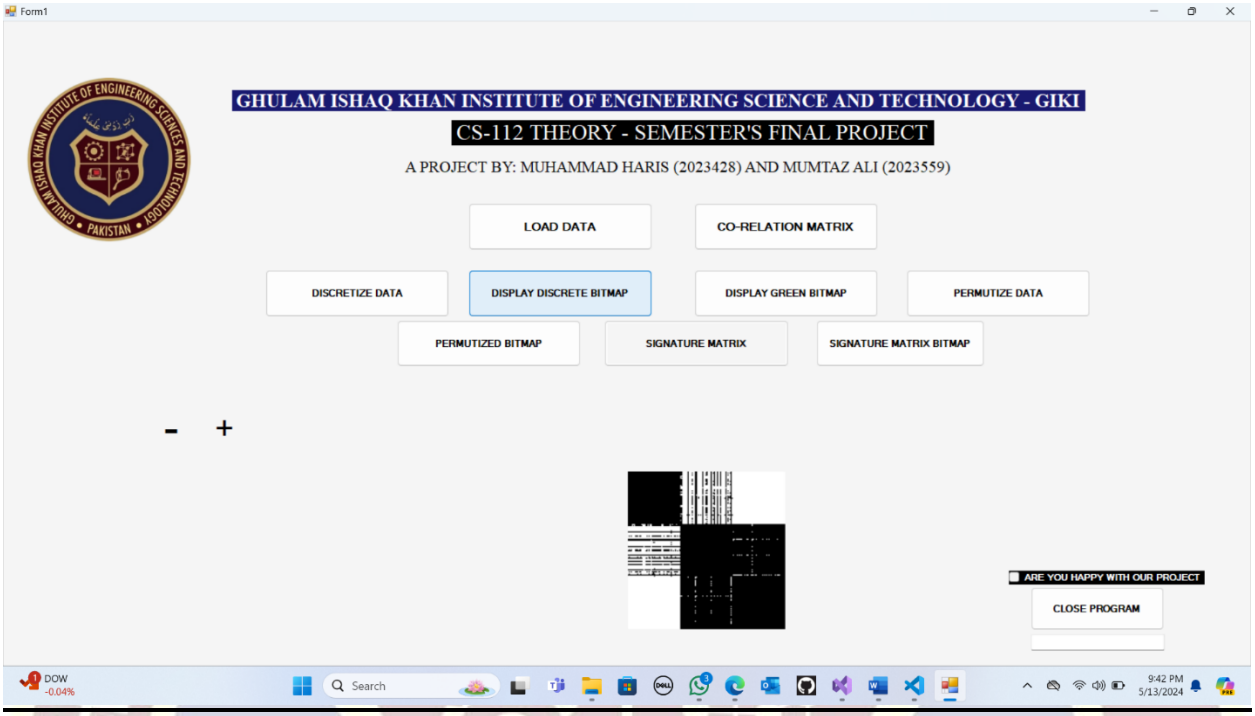
Discretized Data's Display Function:

```
        void DiscretizeandDisplay(array<array<double>^>^ correlationMatrix, bool
displayBitmap) {
            int rows = correlationMatrix->Length;
            array<array<int>^>^ discretizedMatrix = gcnew
array<array<int>^>(rows);

            for (int i = 0; i < rows; i++) {
                discretizedMatrix[i] = gcnew array<int>(rows); // Initialize each
row before assigning values
            }

            for (int i = 0; i < rows; i++) {
                double sum = 0;
                for (int j = 0; j < rows; j++) {
                    sum += correlationMatrix[i][j];
                }
                double mean = sum / rows;
                for (int j = i; j < rows; j++) {
                    if (correlationMatrix[i][j] >= mean)
                        discretizedMatrix[i][j] = 1;
                    else
                        discretizedMatrix[i][j] = 0;
                    // Since it's a symmetric matrix, set the corresponding
element too
                    discretizedMatrix[j][i] = discretizedMatrix[i][j];
                }
            }}
```

Actual Button:

```
private: System::Void button4_Click(System::Object^ sender, System::EventArgs^ e)
{
    array<array<double>^>^ arr = loadData();
    array<array<double>^>^ correlationMatrix = CalculateCorrelationMatrix(arr);
    DiscretizeandDsiplay(correlationMatrix, True);

}
```

Screenshot:



LABEL 4 AND LABEL 5: ZOOM IN AND OUT MECHANISM

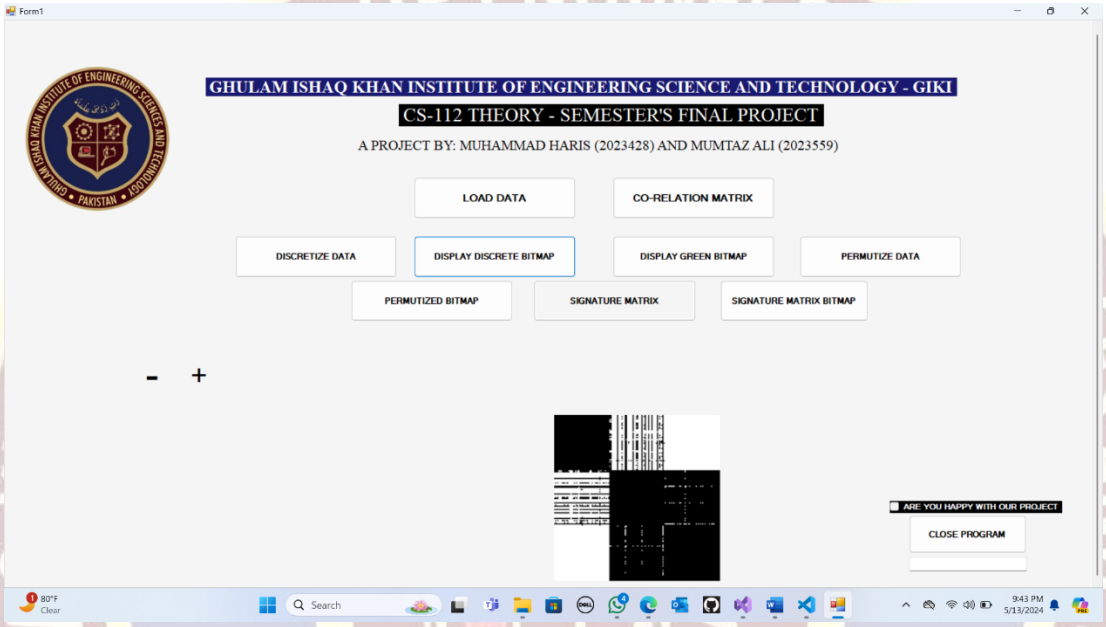
Actual labels:

```
private: System::Void label4_Click(System::Object^ sender, System::EventArgs^ e)
{    pictureBox1->Height *= 1.2;
    pictureBox1->Width *= 1.2;}
private: System::Void label5_Click(System::Object^ sender, System::EventArgs^ e)
{    pictureBox1->Height *= 0.833333;
    pictureBox1->Width *= 0.833333;
}
```

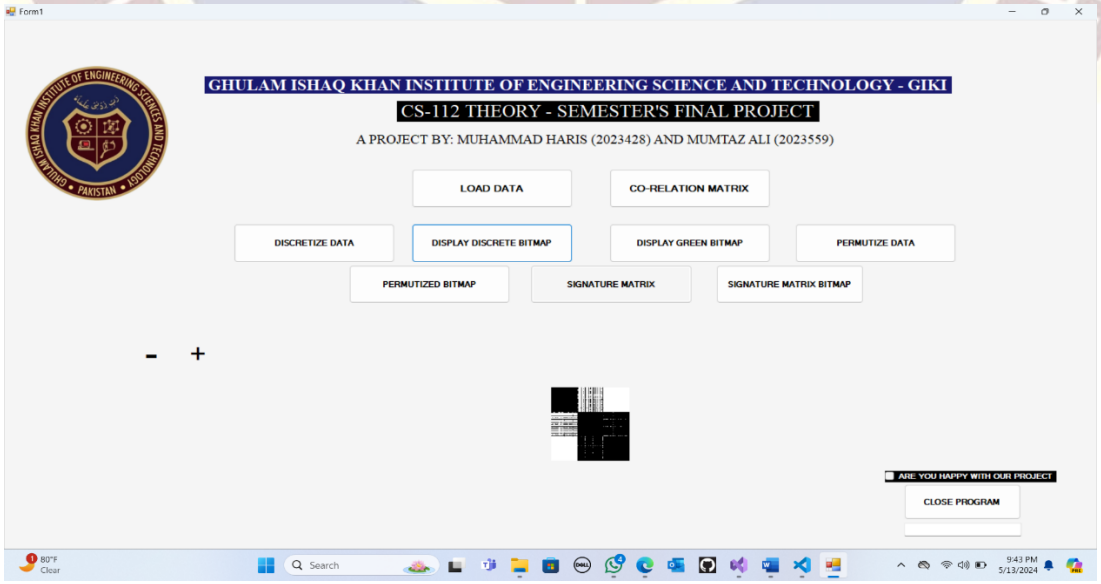
- These are event handlers for click events on two labels in a Windows Forms application, likely written in C++/CLI.
- The first event handler is associated with `label4` and triggers when it is clicked.
- Inside the event handler, the height and width of `pictureBox1` are increased by 20% each time it's clicked, making it larger.

- The second event handler is associated with `label5` and triggers when it is clicked.
- Inside this event handler, the height and width of `pictureBox1` are decreased by 16.6667% each time it's clicked, making it smaller.

Screenshot(1) :DEMO ZOOM IN



Screenshot(2) :DEMO ZOOM OUT



Button 5: DISPLAY GREEN DISCRETE BITMAP

FUNCTIONALITY:

1. Load Data:
2. Calculate Correlation Matrix:
3. Discretize Correlation Matrix:
4. Display Discretized Data:
5. Display Green Bitmap of discretized data:
    - The function `DisplayGreenBitmap` takes a 2D array `correlationMatrix` containing correlation values between variables and creates a bitmap image where pixel colors represent these correlation values.
    - It iterates over all elements of the `correlationMatrix` to find the maximum correlation value. This value is used later to scale the correlation values for color representation.
    - -It creates a new bitmap object with dimensions `rows x rows`, where `rows` is the length of the `correlationMatrix`. Each pixel in this bitmap will represent a correlation value.
    - It iterates over each element in the `correlationMatrix` again and calculates the corresponding shade of green based on the correlation value. The shade is determined by scaling the correlation value to the range `[0, 255]` and setting the green component of the color accordingly.
    - The calculated green value is clamped between 0 and 255 to ensure it stays within the valid range.
    -
    - It sets the color of the pixel in the bitmap corresponding to the current correlation value using the `SetPixel` method.
    - Finally, it assigns the generated bitmap to a PictureBox control named `pictureBox1`. This action effectively displays the bitmap in the user interface.

Green Bitmap's Display Function:

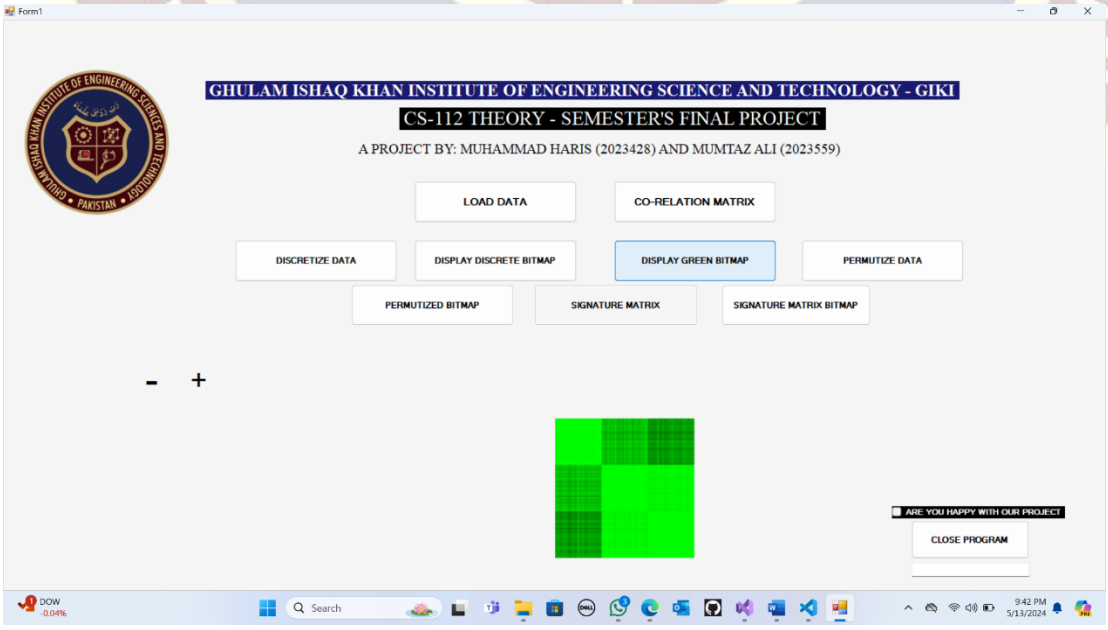
```
void DisplayGreenBitmap(array<array<double>^>^ correlationMatrix) {
          int rows = correlationMatrix->Length;
          int cols = correlationMatrix[0]->Length;
          // Find the maximum correlation value for scaling
          double max = correlationMatrix[0][0];
          for (int i = 0; i < rows; i++) {
              for (int j = 0; j < rows; j++) {
                  if (correlationMatrix[i][j] > max)
```

```
                    max = correlationMatrix[i][j];
                }}// Create the bitmap
            System::Drawing::Bitmap^ b = gcnew System::Drawing::Bitmap(rows,
rows);          System::Drawing::Color c;
            // Set pixel colors based on correlation values
            for (int i = 0; i < rows; i++) {
                for (int j = 0; j < rows; j++) {
                    int green = (int)((correlationMatrix[i][j] / max) * 255); //
Scale the correlation value to the range [0, 255]
                    green = Math::Min(Math::Max(green, 0), 255); // Ensure that
green stays within the valid range
                    c = System::Drawing::Color::FromArgb(0, green, 0);
                    b->SetPixel(i, j, c);
                }           }           // Display the bitmap
            pictureBox1->Image = b;}
```

Actual Button:

```
private: System::Void button5_Click_1(System::Object^ sender, System::EventArgs^
e)
{
    array<array<double>^>^ arr = loadData();
    array<array<double>^>^ correlationMatrix = CalculateCorrelationMatrix(arr);

    // Display the green bitmap
    DisplayGreenBitmap(correlationMatrix);
}
```

Screenshot:



CHAPTER NO:2

TASK 2

Button 6: PERMUTIZE:

FUNCTIONALITY:

1. Load Data:
2. Calculate Correlation Matrix:
3. Discretize Correlation Matrix
4. Permutize/Shuffle the dataset.
5. Display Permutize data.

- The `ShuffleDataset` function takes an array of arrays of doubles (`arr`) as input and returns a shuffled version of it.
- It first determines the number of rows (`rows`) and columns (`cols`) in the input array.
- Then, it creates a deep copy of the input array to avoid modifying the original data.
- The function initializes a new array `copy` with the same dimensions as the input array.
- It iterates over each element of the input array, copying its values to the corresponding position in the `copy` array.
- After making a copy, the function shuffles the rows of the `copy` array using the Fisher-Yates shuffle algorithm to ensure that the rows are randomly reordered.
- It uses `srand(time(NULL))` to seed the random number generator with the current time.
- Then, it iterates from the last row to the second row, randomly swapping each row with another row before it in the array.
- Finally, it returns the shuffled copy of the input array.

- In the actual button click event handler (`button6_Click`), it first loads data using a `loadData` function (not shown here).
- Then, it calls `ShuffleDataset` to get a shuffled version of the loaded data.
- After shuffling, it presumably displays the shuffled dataset using a `DisplayDataset` function (not shown here).

Permutation function:

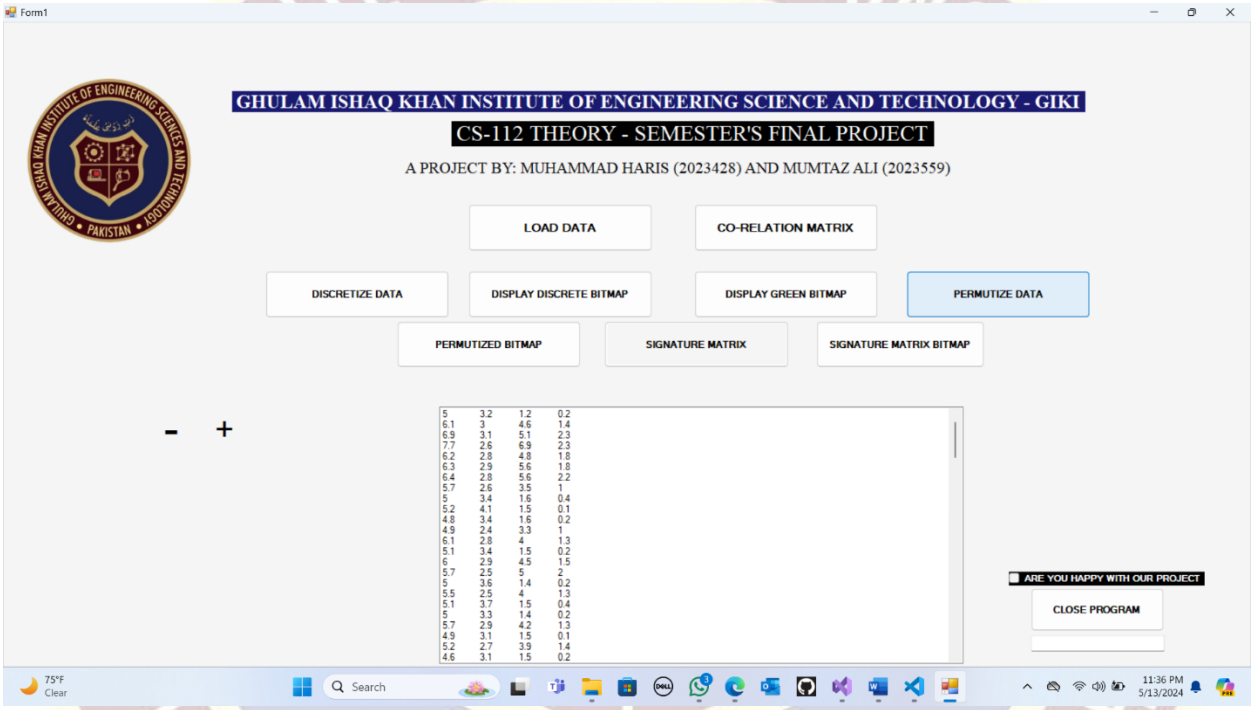
```
    array<array<double>^>^ ShuffleDataset(array<array<double>^>^ arr) {
    int rows = arr->Length;
    int cols = arr[0]->Length;

    // Create a copy of the dataset
    array<array<double>^>^ copy = gcnew array<array<double>^>(rows);
    for (int i = 0; i < rows; i++) {
        copy[i] = gcnew array<double>(cols);
        for (int j = 0; j < cols; j++) {
            copy[i][j] = arr[i][j];
        }
    }

    // Shuffle the dataset using Fisher-Yates shuffle to ensure no repition of
rows already shuffled
    srand(time(NULL));

    for (int i = rows - 1; i > 0; i--) {
        int j = rand() % (i + 1);  // Generate a random index between 0 and i
        array<double>^ tempRow = copy[i];
        copy[i] = copy[j];
        copy[j] = tempRow;
    }

    return copy;
}
```

Actual Button:

```
private: System::Void button6_Click(System::Object^ sender, System::EventArgs^ e)
{
    this->Controls->Add(listBox1);
    this->Controls->Remove(pictureBox1);
    // Load data
```

```
    array<array<double>^>^ arr = loadData();
    int rows = arr->Length;
    int cols = arr[0]->Length;
    // Create a copy of the dataset
    array<array<double>^>^ copy = ShuffleDataset(arr);
    DisplayDataset(copy);
}
```

Screenshot:



Button 7: DISPLAY PERMUTIZE DATA's BITMAP:

1. Load Data:
2. Calculate Correlation Matrix:
3. Discretize Correlation Matrix
4. Permutize/Shuffle the dataset.
5. Display Permutize data.
6. Display Permutize data's bitmap.

- The function `DisplayBlackWhiteBitmap` takes a 2D array of doubles (`array<array<double>^>^ correlationMatrix`) as input and displays it as a black and white bitmap.

- It first determines the number of rows in the correlation matrix.
- Then, it finds the maximum correlation value in the matrix, which will be used for scaling.
- It creates a bitmap object with dimensions equal to the number of rows in the matrix.
- Next, it iterates over each element in the correlation matrix.
- For each element, it calculates a grayscale value based on the ratio of the correlation value to the maximum correlation value. If this ratio is greater than 0.5, it sets the color to white (255), otherwise to black (0).
- It then sets the pixel color in the bitmap accordingly.
- Finally, it displays the bitmap in a picture box (`pictureBox1`). Note that `pictureBox1` needs to be defined and available in the scope of this function.
- This function assumes that the input correlation matrix is square (same number of rows and columns).

Display Bitman Function:

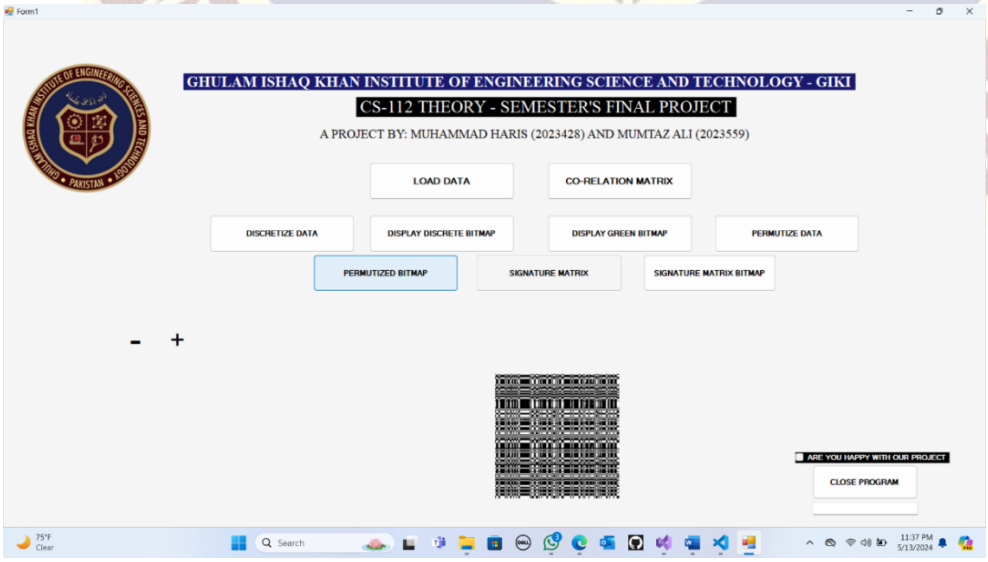
```
void DisplayBlackWhiteBitmap(array<array<double>^>^ correlationMatrix) {
    int rows = correlationMatrix->Length;

    // Create the bitmap
    System::Drawing::Bitmap^ b = gcnew System::Drawing::Bitmap(rows, rows);
    System::Drawing::Color c;
    // Find the mean of each row
    double mean=0;
    double sum=0.0;
    for (int i = 0; i < rows; i++) {
        sum = 0.0;
        for (int j = 0; j < rows; j++) {
            sum += correlationMatrix[i][j];
        }
        mean = sum / rows;
    // Set pixel colors based on correlation values
        for (int j = 0; j < rows; j++) {
            if(correlationMatrix[i][j]>=mean)
```

```
                c = System::Drawing::Color::Black;
            else
                c = System::Drawing::Color::White;
            b->SetPixel(i, j, c);
        }
    }

    // Display the bitmap
    pictureBox1->Image = b;
}
```

Actual Button:

```
private: System::Void button7_Click(System::Object^ sender, System::EventArgs^ e)
{

    this->Controls->Add(pictureBox1);
    this->Controls->Remove(listBox1);
    // Load data
    array<array<double>^>^ arr = loadData();
    int rows = arr->Length;
    int cols = arr[0]->Length;
    // Create a copy of the dataset
    array<array<double>^>^ copy = ShuffleDataset(arr);
    DisplayDataset(copy);
    // Calculate the correlation matrix of the shuffled dataset
    array<array<double>^>^ correlationMatrix = CalculateCorrelationMatrix(copy);
    // Display the green bitmap
    DisplayBlackWhiteBitmap(correlationMatrix);
}
```

Screenshot:



Button 8: SIGNATURE MATRIX:

1. Load Data:
2. Calculate Correlation Matrix:
3. Discretize Correlation Matrix
4. Permutize/Shuffle the dataset.
5. Display Permutize data.
6. Display Permutize data's bitmap.
7. Finding the Signature matrix.

Note: There are some functions in the button no. 8 used that are first defined below and then the actual button's code is shown;

- The function `SortArray` takes an original 2D array (`array<array<double>^>^ originalArray`) and a 1D array of sorted signatures (`array<double>^ sortedSignatures`) as input. It sorts the original array based on the sorted signatures.
- It first determines the number of rows and columns in the original array.
- Then, it iterates over each row of the original array using two nested loops.
- In the inner loop, it compares adjacent elements of the sorted signatures array.
- If the signature at index `j` is greater than the signature at index `j + 1`, it swaps the signatures and the corresponding rows in the original array.
- The swapping of rows is done by creating a temporary row array to hold the values of the row to be swapped.
- After swapping, the `swapped` flag is set to true.
- If no elements were swapped in the inner loop, indicating that the array is already sorted, the outer loop breaks.
- Finally, it returns the sorted original array.
- This function implements the bubble sort algorithm to sort the original array based on the sorted signatures.

Sorting Function:

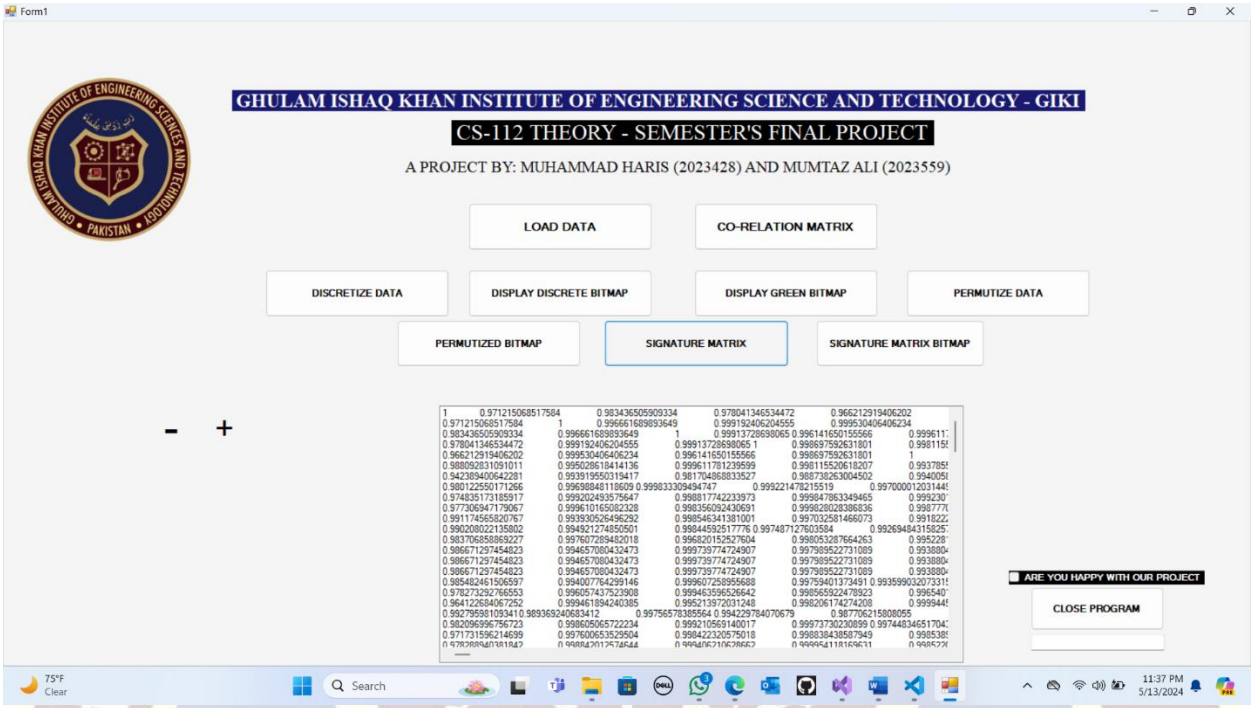
```
// Function to sort the original 2D array based on the sorted 1D signature matrix
array<array<double>^>^ SortArray(array<array<double>^>^ originalArray,
array<double>^ sortedSignatures) {
    int rows = originalArray->Length;
    int cols = (rows > 0) ? originalArray[0]->Length : 0;
    for (int i = 0; i < rows - 1; i++) {bool swapped = false;
        for (int j = 0; j < rows - i - 1; j++) {
            if (sortedSignatures[j] > sortedSignatures[j + 1]) {
                // Swap signatures
                double tempSignature = sortedSignatures[j];
                sortedSignatures[j] = sortedSignatures[j + 1];
                sortedSignatures[j + 1] = tempSignature;

                // Swap corresponding rows in the original array
                array<double>^ tempRow = gcnew array<double>(cols);

                for (int k = 0; k < cols; k++)
                {
                     tempRow[k]= originalArray[j][k];
                    originalArray[j + 1][k] = originalArray[j][k];
                    originalArray[j][k] = tempRow[k];}
                swapped = true;}}
        // If no two elements were swapped in the inner loop, then the array is
sorted
        if (!swapped) {
            break;
        }    }    return originalArray;}
```

Actual Button:

```
private: System::Void button8_Click(System::Object^ sender, System::EventArgs^ e)
{
    this->Controls->Add(listBox1);
    this->Controls->Remove(pictureBox1);
array<array<double>^>^ arr = loadData();
    int rows = arr->Length;
    int cols = (rows > 0) ? arr[0]->Length : 0;
if (rows <= 0 || cols <= 0) {
        MessageBox::Show("Error: Invalid dataset.", "Error",
MessageBoxButtons::OK, MessageBoxIcon::Error);
        return;}
    array<array<double>^>^ copy = ShuffleDataset(arr);
    // Recover the image clusters using the Signature technique
    // Declare sum and mean arrays with sizes
```

```
    double* sum = new double[rows];
    double* mean = new double[rows];
    for (int i = 0; i < 150; i++) {
        sum[i] = 0.0;
        mean[i] = 0.0;
    }

    array<double>^ signatures = gcnew array<double>(rows);

    // Calculate sum of values in each row and store it in the sum array
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            sum[i] += copy[i][j];
        }
    }

    // Calculate mean of each row
    for (int i = 0; i < rows; i++) {
        mean[i] = sum[i] / cols; // Change from 4 to cols
    }

    // Multiply the sum of each row with its mean to produce a signature for each
row
    for (int i = 0; i < rows; i++) {
        signatures[i] = sum[i] * mean[i];
    }

    // Sort the signature matrix and update the original array accordingly
    array<array<double>^>^ sortedCopy = SortArray(copy, signatures);
    finalSignMatrix = CalculateCorrelationMatrix(sortedCopy);
    listBox1->Items->Clear();
    for (int i = 0; i < rows; i++) {
        String^ rowString = "";
        for (int j = 0; j < rows; j++) {
            // Access elements of finalSignMatrix using single indexing
            rowString += finalSignMatrix[i][j].ToString() + "\t";
        }
        listBox1->Items->Add(rowString);
    }

    // Display the final result
    DisplaySignGBP(finalSignMatrix);
}
```

Screenshot:



Button 9: DISPLAY SIGNATURE MATRIX:

1. Load Data:
2. Calculate Correlation Matrix:
3. Discretize Correlation Matrix
4. Permutize/Shuffle the dataset.
5. Display Permutize data.
6. Display Permutize data's bitmap.
7. Finding the Signature matrix.
8. Display Signature Matrix.

- The function `DisplaySignGBP` takes a 2D array of doubles (`array<array<double>^>^ correlationMatrix`) as input and displays it as a green-black bitmap, where the intensity of green represents the correlation values.
- It first determines the number of rows in the correlation matrix.

- Then, it finds the maximum correlation value in the matrix, which will be used for scaling.
- It creates a bitmap object with dimensions equal to the number of rows in the matrix.
- Next, it iterates over each element in the correlation matrix.
- For each element, it scales the correlation value to the range [0, 255] and uses this value to determine the intensity of green. Green intensity is set to 0 if the correlation is 0 and increases as the correlation value approaches the maximum correlation.
- It ensures that the green intensity stays within the valid range [0, 255].
- It then sets the pixel color in the bitmap using the green intensity, while keeping red and blue channels at 0, resulting in shades of green.
- Finally, it displays the bitmap in a picture box (`pictureBox1`). Note that `pictureBox1` needs to be defined and available in the scope of this function.

Display Signature Matrix function:

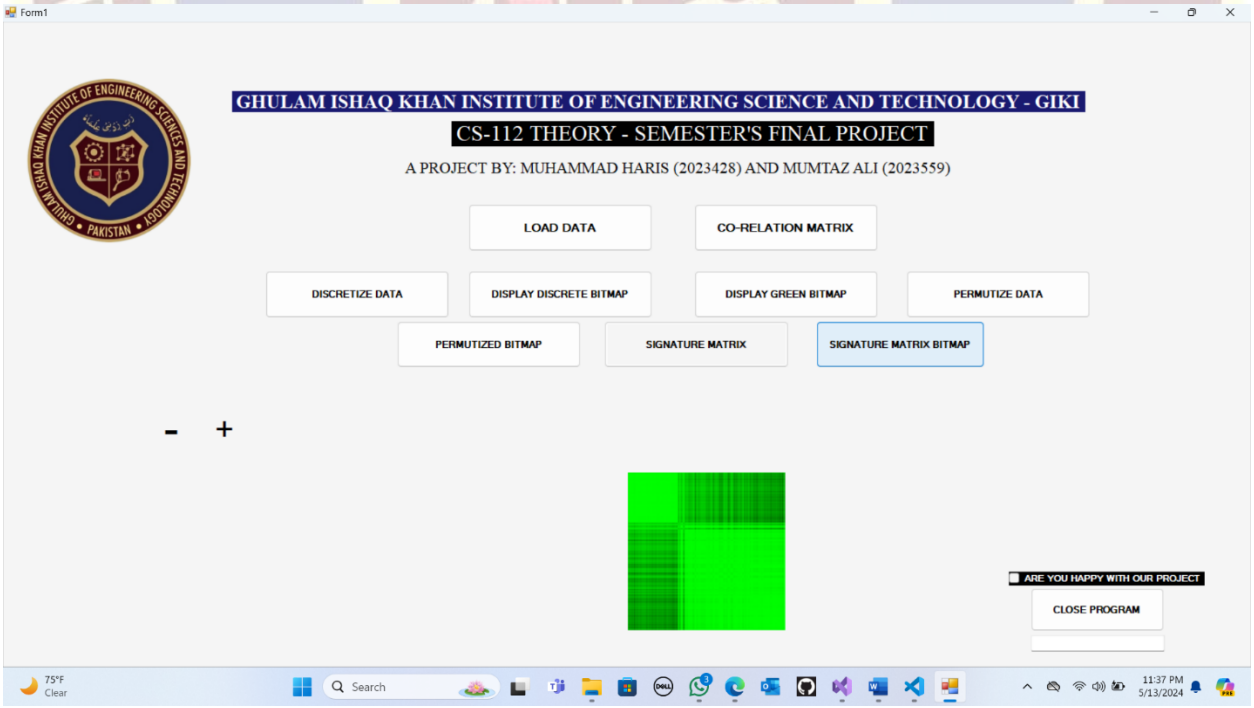
```
void DisplaySignGBP(array<array<double>^>^ SigncorrelationMatrix) {
    int rows = SigncorrelationMatrix->Length;
    // Create the bitmap
    System::Drawing::Bitmap^ b = gcnew System::Drawing::Bitmap(rows, rows);
    System::Drawing::Color c;

    for (int i = 0; i < rows; i++) {
        // Find the maximum correlation value for scaling
        double max = SigncorrelationMatrix[0][0];
        for (int j = i; j < rows; j++) { // Only consider the upper triangle of
the correlation matrix
            if (SigncorrelationMatrix[i][j] > max)
                max = SigncorrelationMatrix[i][j];
        }



        // Set pixel colors based on correlation values
```

```
        for (int j = i; j < rows; j++) {
            int green = (int)((abs(SigncorrelationMatrix[i][j]) / max) * 255); //
Scale the correlation value to the range [0, 255]
            c = System::Drawing::Color::FromArgb(0, green, 0);
            b->SetPixel(i, j, c);
            b->SetPixel(j, i, c); // Set the corresponding element in the lower
triangle of the bitmap
        }
    }

    // Display the bitmap
    pictureBox1->Image = b;
}
```

Actual Button:

```
private: System::Void button9_Click(System::Object^ sender, System::EventArgs^ e)
{
    this->Controls->Add(pictureBox1);
    this->Controls->Remove(listBox1);
    // Display the final result
    DisplaySignGBP(finalSignMatrix);
}
```

Screenshot:



Button 10: CLOSE PROGRAM:

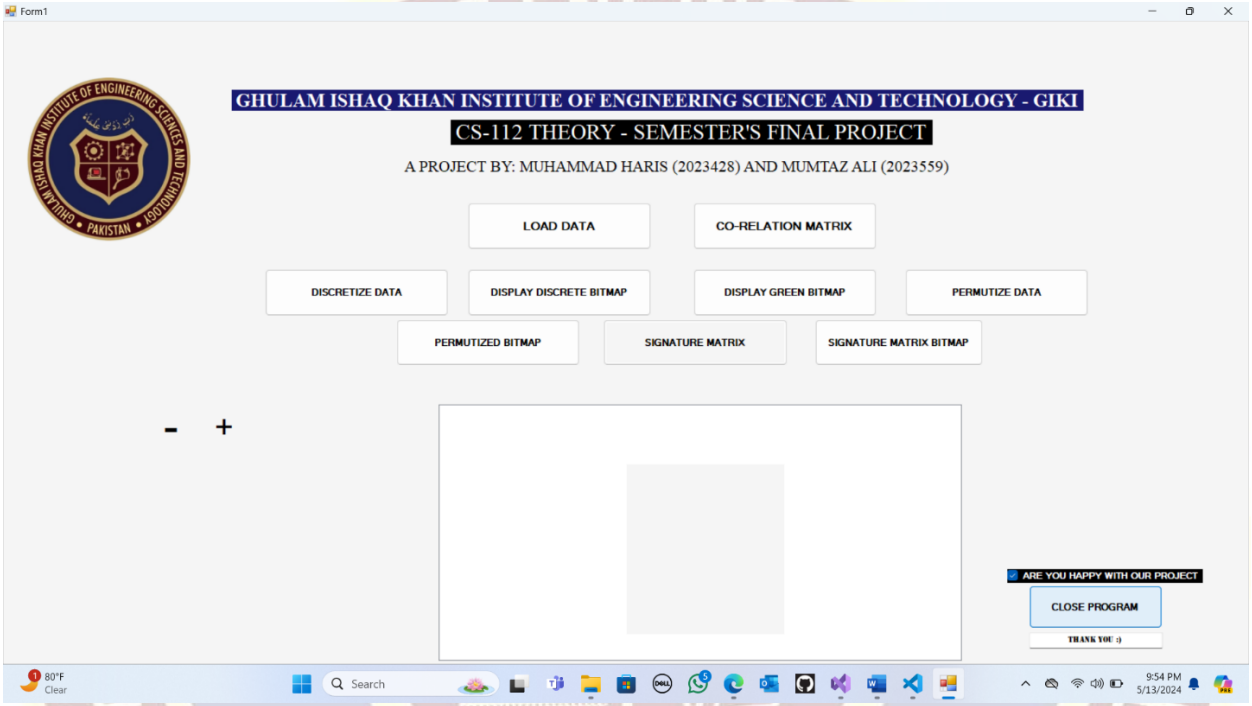
1. Load Data:
2. Calculate Correlation Matrix:
3. Discretize Correlation Matrix
4. Permutize/Shuffle the dataset.
5. Display Permutize data.
6. Display Permutize data's bitmap.
7. Finding the Signature matrix.
8. Display Signature Matrix.
9. Close the program.
    - It checks whether a checkbox (presumably named checkBox1) is checked.
    - If the checkbox is not checked:
    - It sets the text of a textbox (presumably named textBox1) to "ARE YOU NOT HAPPY :(".
    - It enters a loop that continues until the checkbox is checked. Within this loop, it repeatedly calls Application::DoEvents() to allow the application to handle other events while waiting for the checkbox to be checked.
    - If the checkbox is checked:
    - It sets the text of the textbox to "THANK YOU :)".
    - It closes the current form and exits the program.

Actual Button:

```
private: System::Void button10_Click(System::Object^ sender, System::EventArgs^
e) {
    // Check if the checkbox is checked
    if (!checkBox1->Checked) {
        // Show a message in the textbox
        textBox1->Text = "ARE YOU NOT HAPPY :(";

        // Loop until the checkbox is checked
        while (!checkBox1->Checked) {
            Application::DoEvents(); // Allow the application to handle other
events
        }
    }
```

```
    else {
        // Show a message in the textbox
        textBox1->Text = "THANK YOU :)";
// Close the form and exit the program        this->Close();}}
```

Screenshot:



CHAPTER NO: 3

TASK 3

NOTE: This report is prepared keeping in view the instructions for a detailed write-up, explaining each step taken to achieve every single task in the project.

WORK DISTRIBUTION:

1. **MUHAMMAD HARIS (50%)**
2. **MUMTAZ ALI (50%)**

Each part of the project was done by both student's hand in hand; from logic building, to writing the actual code to implementation, design, and optimization. With the mutual commitment to advancing in the project with learning as much as there is to be learned from the project.

If one student was on the computer, the other was with a pen and paper drawing all the logical reasonings required for the code and vice versa. The transitioning took place until the whole code was compiled and ready for submission.