CS-221 Data Structures and Algorithm

Semester Project Report

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology, Topi, Swabi 23460, Pakistan

Members:

- 2021016 (Abdul Moiz)
- 2021059 (Ahmad Farid)
- 2021451 (Muhammad Sameer Shahzad)
- 2021578 (Sermad Mehdi)
- 2021363 (Baqar Raza)

Table of Contents:

Task 1:

Introduction:

We were given an input data Sample data Iris. We read the file through fstream and made 2d array and stored all of its data in it. Afterwards, we sent the array to a coefficient function which through the formula implemented gave us a coefficient matrix and we stored it in a coefficient .txt file. We passed this file into a mean function which calculated mean of different cols and set all the values in the file which were higher or equal to a 1 and the rest to 0. We then stored this data of 1 and 0 in a discretized file. We then used this file in the qt creator and converted it into bitmap using qmake. We read and stored the discretized files data in an array and converted the array to a bitmap. Where 1s were given a green color and the 0s were given a black color. We then added the zoom in and zoom out functionality in it.

Code Implementation:

Libraries used :

```
#include<iostream>
#include<cmath>
#include<fstream>
```

The correlation function that makes a correlation matrix from the discretized matrix. It uses the Pearson's correlation coefficient. The values of correlation matrix range -1 to 1 inclusive.

```
void corelation(float ** arr, float rows, float columns){

//opening file for storing coeff data
ofstream fout;
    fout.open("coeff.txt");

//a variable for displaying
float count=1;

//first lopp which tell basically how many times the whole process is going to be repeated =rows
for(int i=0;i<rows;i++){

float  cor=0;
float  sumx=0;
float meanx;
float  sumx2=0;


float x;
float y;

//for calculating sum of 1 row so k= columns
  for(int k=0;k<columns;k++){// k=0,k=1


    sumx+=arr[i][k];
    float x2= arr[i][k]*arr[i][k];
    sumx2+=x2;
    meanx=sumx/columns;
    }
```

Code to open , read and store the data:

```
  //open a file to read from
  ifstream reading("Sample data-1-IRIS.txt");

//reading rows and columns from the data
for(int i=0;i<2;i++){
   reading>>arr2[i];
}
//assigning rows and columns
int rows=arr2[0];
int  columns=arr2[1];


// making an array from the rows and data collected
float   **array;
array = new float   *[rows];
for(int  i = 0; i <rows; i++)
array[i] = new float  [columns];

//reading from file and storing data into an array
  for(int i=0;i<rows;i++)
  for(int  j=0;j<columns;j++)
      reading>>array[i][j];


 //sending array to correlation
 corelation(array, rows ,columns);
```

Pearson's correlation coefficient formula:

- This will give correlation matrix , which will be in range of -1 to 1.
- Diagonals will always be 1.

```
//formula for calculation pearson's corelation
float base=(((columns*sumx2)-(sumx*sumx))*((columns*sumy2)-(sumy*sumy)));
```

Calculating mean:

```
//calculating mean
mean=sum/rows;

meanarr[i]=mean;
}
//opening file to store mean values
ofstream out;
out.open("mean.txt");

//storing data
for(int i=0;i<rows;i++)
out<<meanarr[i]<<" ";

out.close();
```

Creating the discretized matrix of 1's and 0's:

```
//making a discretize matrix
ofstream outs;
outs.open("discretize.txt");

for(int i=0;i<rows;i++){
for(int j=0;j<rows;j++){

if(corl[j][i]<meanarr[i]){
corl[j][i]=0;
}
else{
    corl[j][i]=1;
}
}
}
for(int i=0;i<rows;i++){
for(int j=0;j<rows;j++){
outs<<corl[i][j]<<" ";
}
outs<<endl;
}
```

- After calculating median/mean of each column of the correlation matrix
- Set all the values in that column that are above the calculated ***median/mean to 1 and rest to 0***.
- This will generate ***discretized matrix***.

Visualization:

Libraries used in QT:

```
#include "mainwindow.h"
#include <QApplication>
#include <QFile>
#include <QTextStream>
#include <QImage>
#include <QLabel>
#include <QDir>
#include <QPixmap>
#include <QGraphicsView>
#include <QPushButton>
#include <QGraphicsScene>
#include <QGraphicsPixmapItem>
#include <QPixmap>
#include <QImage>
#include <QGraphicsProxyWidget>
```

Opening File that contains matrix data:

```
// Open the file containing the matrix data
QFile file("D:/Study/3rd sem/dsa/project/project code/discretize.txt");
if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
  qDebug() << "Error opening file";
  return 1;
}
```

Making bitmap by pixel values. Setting 1 as green and 0 as black:

```
// Create a QImage object with the same dimensions as the matrix
QImage image(cols, rows, QImage::Format_RGB32);

// Iterate over each element in the matrix and set the pixel value in the QImage
for (int y = 0; y < rows; y++) {
  for (int x = 0; x < cols; x++) {
    // Set the pixel value at (x, y) to green if the element is 1, or black if it is 0
    if (matrix[y][x] == 1) {
      image.setPixel(x, y, qRgb(0, 255, 0));
    } else {
      image.setPixel(x, y, qRgb(0, 0, 0));
    }
  }
}
QPixmap pixmap = QPixmap::fromImage(image);
QGraphicsView view;
```

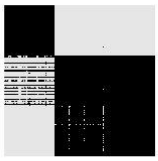
Created zoom in button (same as zoom out button):

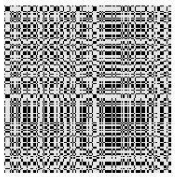
```
// Create QPushButton objects for the zoom in and zoom out buttons
QPushButton zoomInButton("Zoom In");
QPushButton zoomOutButton("Zoom Out");

// Connect the zoom in button's clicked signal to a slot function
QObject::connect(&zoomInButton, &QPushButton::clicked, [&]() {
    // Zoom in on the pixmap item
    static qreal scaleFactor = 1.0;
    scaleFactor += 0.1;
    pixmapItem->setScale(scaleFactor);
});
```

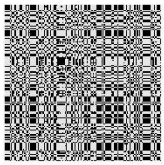
Displays the bitmap and the buttons on a panel:

```
// Show the view
view.show();
```

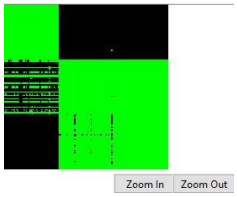
Visualization of discretized matrix:

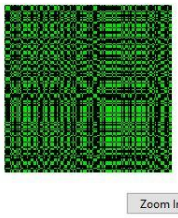


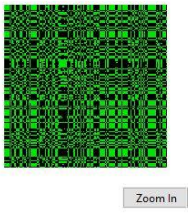




Color coded matrix visualization:







Task 2:

Introduction:

We have the data matrix now, and we permute it through shuffling its rows via selecting random indexes using the ***rand() function*** and then basically assigning the rows random indexes so that they get shuffled. Then we ***color code*** the resulting matrix the same way we did in task 1. Then we calculate the signature value of each row by finding the sum of each row and then calculating the mean, and finally, multiplying the mean with the sum. Then we call a ***sort function*** to rearrange the matrix in sorted order according to the signature of each row.

Now we apply task 1 to this matrix and display the color-coded image.

Code Implementation:

Permute function :

```
102  void Permute(vector<vector<double>> data, int rows, int cols, string fileName)
103  {
104      // Seed the rand() function with the current time
105      srand(time(NULL));
106
107      for (int i = 0; i < rows; i++) {
108          int j = rand() % rows;
109
110          // Swap the current row with the row at the random index
111          vector<double> temp(cols);
112          for (int k = 0; k < cols; k++) {
113              temp[k] = data[i][k];
114              data[i][k] = data[j][k];
115              data[j][k] = temp[k];
116          }
117      }
118
119      // Open the file in write mode
120      ofstream outFile;
121      outFile.open(fileName);
122
123      // Write the permuted matrix to the file
124      for (int i = 0; i < rows; i++)
125      {
126          for (int j = 0; j < cols; j++)
127          {
128              outFile << data[i][j] << " ";
129          }
130          outFile << endl;
131      }
132
133      // Close the file
134      outFile.close();
135  }
```

- The function uses **srand()** to randomize data so that data can be shuffled.
- Random values are attained by rand() to randomize the rows
- Swap logic for rows only used as seen in line 12.
- It also sends the created permuted matrix to file.

Function “DeepCopyMatrix”:

This function is used to copy all the data of one matrix to another. So the original matrix stays as it is we copy it in a new variable.

```
void DeepCopyMatrix(const vector<vector<double>>& Matrix1, vector<vector<double>>& Copy, int rows, int cols)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            Copy[i][j] = Matrix1[i][j];    //Copying M1----->M2
        }

    }
}
```

Previous

Function “mean”:

```cpp
double mean(const vector<double>& row) {
    int n = row.size();
    if (n == 0) {
        return 0.0;
    }
    double sum = 0.0;
    for (double x : row) {
        sum = sum+ x;
    }
    return sum / n;
}
```

- Then mean which is sum of entries divided by total row size is taken so that signature value of the row can be formed.

Function “**compareRowsBySignature**”:

- A function that uses signature value of rows and uses it to compare two rows.
- Method: multiplying mean a row with the sum of entries of row.

```cpp
// A comparator function that compares the signature value of two rows
bool compareRowsBySignature(const vector<double>& row1, const vector<double>& row2) {
    double signature1 = 0.0;
    double signature2 = 0.0;
    int n1 = row1.size();
    int n2 = row2.size();
    if (n1 > 0) {
        signature1 = mean(row1) * accumulate(row1.begin(), row1.end(), 0.0);
    }
    if (n2 > 0) {
        signature2 = mean(row2) * accumulate(row2.begin(), row2.end(), 0.0);
    }
    return signature1 < signature2;
}
```

- Then it compares two signatures, it returns the row with the smaller signature value.

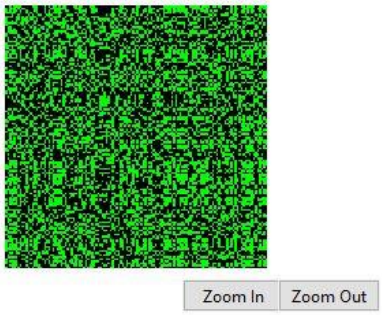
- We used **permuted(shuffled)** matrix for signature technique.
- ***A sort function*** is used to sort the rows of the matrix accordingly to their signature

```cpp
DeepCopyMatrix(OG_MATRIX, Permute_Matrix, rows, cols);

Permute(Permute_Matrix, rows, cols, "permuted_Matrix.txt");

cout << endl << "Sorted Matrix by Signature Method - Sign compared between two rows" << endl;
DeepCopyMatrix(OG_MATRIX, Sign_Sort_Matrix, rows, cols);

sort(Sign_Sort_Matrix.begin(), Sign_Sort_Matrix.end(), compareRowsBySignature);

// Print the sorted matrix
for (const vector<double>& row : Sign_Sort_Matrix) {
    for (double x : row) {
        cout << x << " ";
    }
    cout << endl;
}
// Open a file for writing
ofstream sfile("signature_matrix.txt");

// Iterate over the rows and columns of the matrix
for (int i = 0; i < Sign_Sort_Matrix.size(); i++) {
    for (int j = 0; j < Sign_Sort_Matrix[i].size(); j++) {
        // Write the element to the file
        sfile << Sign_Sort_Matrix[i][j] << " ";
    }
    // Add a newline after each row
    sfile << endl;
}


return 0;
```

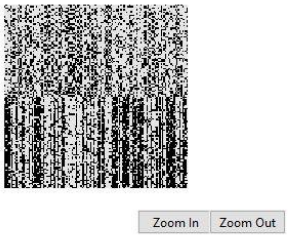
- The matrix is sent to the file by for loop so it can be visualized.

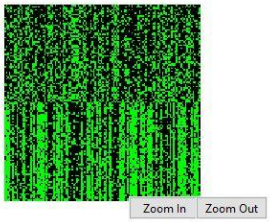
Visualization :

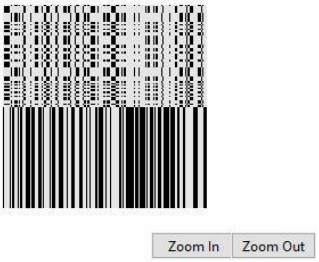
Sample data 2 before and after permutation respectively:

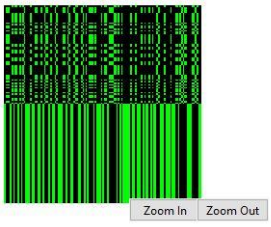


Similarity matrix:









Task 3:

Introduction:

We first implement the graph data structure in C++ by creating the edge, node, and graph structs and their respective functions like ***addEdge()***. We remove the edges which are below a certain threshold by comparing each edge of the correlation matrix with the threshold and only putting the ones that are bigger than the edge in a ***filteredMatrix*** at those same indexes. Essentially setting all the edges below the threshold to 0. We find the node with maximum weight from the filtered matrix; this will be the cluster node. We push the cluster node along with its neighbor nodes into a vector. We repeat this process until every cluster is found with its unique neighbors. This will be assured by using a bool check.

Code Implementation:

Libraries used :

```cpp
#include <iostream>
#include <vector>
#include <fstream>
using namespace std;
```

Making structures to show edges, nodes and graph:

- We make struct to represent an edge.
- We make a struct to represent the node.
- We make a struct to represent a graph.

```cpp
// A structure to represent an edge in the graph
struct Edge {
    int neighbor;
    double weight;
};

// A structure to represent a node in the graph
struct Node {
    int id;
    vector<Edge> edges;
    double weight;
};

// A structure to represent a weighted graph
struct Graph {
    int numNodes;
    Node* nodes;
};
```

- To get the determine which nodes have a value greater than the threshold value in the correlation matrix and we make the rest of the nodes 0
- We push them into ***filteredMatrix.***

```cpp
vector<vector<double>> filterGraph(const vector<vector<double>>& correlationMatrix, double threshold) {
    int rows = correlationMatrix.size();
    int cols = correlationMatrix[0].size();
    vector<vector<double>> filteredMatrix(rows, vector<double>(cols));

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (correlationMatrix[i][j] >= threshold) {
                filteredMatrix[i][j] = correlationMatrix[i][j];
            }
            else {
                filteredMatrix[i][j] = 0;
            }
        }
    }

    return filteredMatrix;
}
```

- we find the maximum weight of a node of a filtered matrix.
- We sum the row data of a row if the threshold is less than or equal to the filtered matrix at that index.
- We push the highest weighted node into the cluster along with its neighboring nodes.

```cpp
vector<int> getHighestWeightCluster(const vector<vector<double>>& filteredMatrix, double threshold) {
    int rows = filteredMatrix.size();
    vector<bool> visited(filteredMatrix.size(), false);

    vector<int> cluster;
    vector<int> neighbors;

    // Find the node with the highest weight
    int highestWeightNode = -1;
    double highestWeight = -1;
    for (int i = 0; i < rows; i++) {
        if (i < 0 || i >= rows) continue; // skip invalid indices
        double weight = 0;
        for (int j = 0; j < rows; j++) {
            if (filteredMatrix[i][j] >= threshold) {
                weight += filteredMatrix[i][j];
            }
        }
        if (weight > highestWeight) {
            highestWeight = weight;
            highestWeightNode = i;
        }
    }

    // Add the highest weight node to the cluster
    cluster.push_back(highestWeightNode);
    visited[highestWeightNode] = true;

    // Find the neighbors of the highest weight node
    for (int j = 0; j < rows; j++) {
        if (filteredMatrix[highestWeightNode][j] >= threshold && !visited[j]) {
            neighbors.push_back(j);
            visited[j] = true;
        }
    }

    return cluster;
    return neighbors;
}
```

- We set its check to true so that the neighbors remain unique.
- We repeat this process until no cluster is less.
- Then we return the cluster and neighbors.

- This part of code is in main function

```
137
138
139     // Print the cluster and its neighbors
140     cout << "Cluster nodes: ";
141     for (int node : cluster) {
142         cout << node << " ";
143     }
144     cout << endl;
145     cout << "Neighbors: ";
146     for (int node : cluster) {
147         for (int j = 0; j < rows; j++) {
148             if (filteredMatrix[node][j] >= threshold && j != node) {
149                 cout << j << " ";
150             }
151         }
152     }
153     cout << endl;
154
```

- The two for loops are giving us the cluster nodes along with its neighbors

- Its reading data from the file and writing it into the correlation matrix.

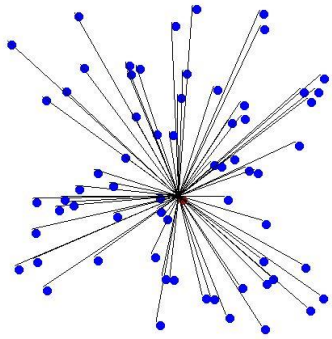
- This is our **graph**.

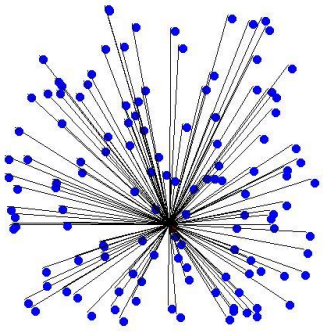
- The user is being asked to input the **threshold** value.

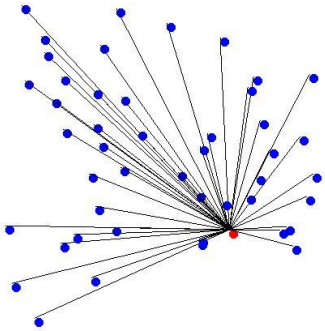
```
98  int main() {
99      // Read the matrix from a text file
100     ifstream file("correlation_matrix.txt");
101     int rows = 125, cols = 125;
102     vector<vector<double>> correlationMatrix(rows, vector<double>(cols));
103     for (int i = 0; i < rows; i++) {
104         for (int j = 0; j < cols; j++) {
105             file >> correlationMatrix[i][j];
106         }
107     }
108
109     // Print the matrix
110     for (int i = 0; i < rows; i++) {
111         for (int j = 0; j < cols; j++) {
112             cout << correlationMatrix[i][j] << " ";
113         }
114         cout << endl;
115     }
116
117
118     double threshold;
119     // Read in the threshold value from the user
120     cout << "Enter the threshold value: ";
121     cin >> threshold;
122
```

Visualization:

Clusters:







Comparing task 2 with task 3:

Task 1 primarily focuses on rearranging and visualizing the data matrix, while Task 2 focuses on creating and analyzing a graph representation of the data.

First difference :

- Task 2 signature of two adjacent is found and then matrix is sorted known as signature matrix.

- Task 3 correlation is found to make this matrix, a graph.

Second difference :

- Task 2 takes permuted(shuffled) matrix converted into signature matrix (by calculating mean of rows) matrix. Then we applied Task 1, which is discretized matrix by judging by the mean/median, into 1 and 0s;
- Task 3 takes threshold input and puts values 0 that are below threshold, basically removing edges.

Third difference :

- Task 2 visualizes bitmap (color coded).
- We make similarity matrix using signature technique.

- Task 3 visualizes clusters.
- Returns cluster node also neighboring nodes and make up an entire cluster.

Task Distribution:

Visualization: Learned the QT library, shared knowledge and implemented together

Ahmad Farid

Sameer Shehzad

Sermad Mehdi

Back end: Tasks were divided and were done sitting together

Sermad Mehdi

Baqar Raza

Ahmad Farid

Abdul Moiz

Documentation: This was done under supervision of Backend and frontend coders.

Abdul Moiz

Sameer Shehzad

Baqar Raza