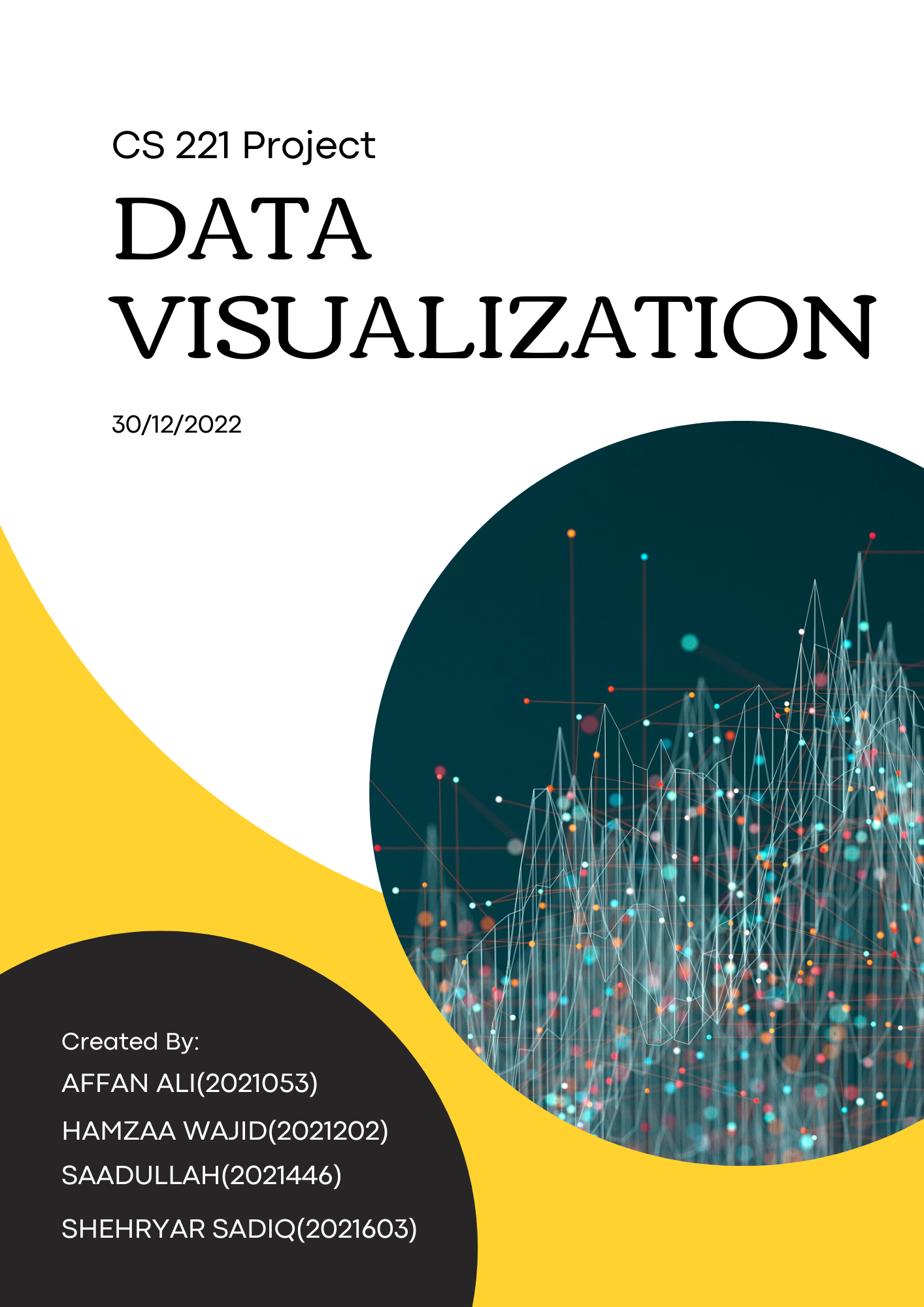
****

**CODING PART**

This project code contains a class named **data\_clustering** in which all the functions of the program resides . The variables of the class are

row ,column, data matrix 2D array of type double , correlation matrix 2D array of type double , signature 1D array of type double ,node 1D array of type double ,2d vector which is storing clusters, 1D bool cluster check array that checks weather the given node has been included in cluster or not ,and a counter that is used to resize an cluster vector whenever the number of clusters increases.

**FUNCTIONS**

**Get\_data()**

This function is taking all the data that is in the file and storing that in 2d matrix of dimensions ROW\*COLUMN . The function is doing that in a two nested while loops .The first loop first sets the rows and columns by reading them from line 1 and 2 of the file .On moving to line 4 , another while loop runs that reads the entire line and tokenizes the four different data members and store them in different columns of data matrix array.

void get\_data()

{

fstream my\_file;

    my\_file.open("Sample data-1-IRIS.TXT",ios::in);

    int line =1;

    int row\_count=0;

    while(!my\_file.eof())

    {

        string x="hamza";

        string line\_data;

        getline(my\_file,line\_data);

        if(line==1)

        {

            row=stoi(line\_data);

            array=new double\*[row];

        }

        else if(line==2)

        {

            column=stoi(line\_data);

            for(int i=0;i<row;i++)

            {

                array[i]=new double[column];

            }

        }

        else if(line>3){

            string element={NULL};

            int i=0;

                int j=0;

            while(line\_data[i]!='\0')

            {

                if(line\_data[i+1]=='\t'||line\_data[i+1]=='\0')

                {

                    element+=line\_data[i];

                   element[0]='0';

                   double element2=stod(element);

                   array[row\_count][j]=element2;

                   i+=2;

                   element={NULL};

                   if(j==3)

                   {

                    break;

                   }

                   j++;

                }

                else{

                element+=line\_data[i];

                i++;

                }

            }

            row\_count++;

            line\_data.clear();

        }

            line++;

    }

}

**Task1**

**Upper part ():**

This function performs the numerator part of the Pearson’s correlation formula on the rows which are passed into it as parameters. After performing the function, it returns a value of double data type.

double upper\_part(int row1,int row2)

{

    double answer=0;

    for(int i=0;i<column;i++)

    {

        answer+=((array[row1][i]-mean(row1))\*(array[row2][i]-mean(row2)));

    }

    return answer;

}

**Lower part ():**

This function performs the denominator part of the Pearson’s correlation formula on the rows which are passed into it as parameters. After performing the function, it returns a value of double data type.

double lower\_part(int row1,int row2)

{

   double xtotal=0;

   double ytotal=0;

   for(int i=0;i<column;i++)

   {

    xtotal+=((array[row1][i]-mean(row1))\*(array[row1][i]-mean(row1)));

     ytotal+=((array[row2][i]-mean(row2))\*(array[row2][i]-mean(row2)));

   }

   double total=xtotal\*ytotal;

   total=sqrt(total);

   return total;

}

**Correlation Matrix ():**

This function creates a 2-D matrix having a size of N\*N with N representing the no. of rows in the sample data. It then performs the Pearson’s correlation coefficient on the rows of sample data using the upper part and lower part functions and then places the answer into the matrix. The row and column of the matrix correspond to the first row and the second row, used in the Pearson’s formula, respectively.

void correlationMatrix()

{

   corelation\_matrix=new double\*[row];

   for(int i=0;i<row;i++)

   {

    corelation\_matrix[i]=new double[row];

   }

   for(int i=0;i<row;i++)

   {

    for(int j=0;j<row;j++)

    {

        corelation\_matrix[i][j]=upper\_part(i,j)/lower\_part(i,j);

    }

   }

}

**Print array ():**

This function prints the data matrix.

void print\_array()

{

    for(int i=0;i<row;i++)

    {

        for(int j=0;j<column;j++)

        {

            cout<<'('<<array[i][j]<<")";

        }

        cout<<endl;

    }

}

**Mean\_columns():**

This function takes the mean of the columns, which is taken as parameter, of the correlation matrix.

double mean\_column(int column)

{

    double total=0;

    for(int i=0;i<row;i++)

    {

        total+=corelation\_matrix[column][i];

    }

    return total/double(row);

}

**Update\_corelation():**

This function updates the correlation matrix by taking the mean of column, using it as a threshold and assigning 0 to values below the threshold and assigning 1 to values above the threshold.

void update\_corelation()

{

    for(int i=0;i<row;i++)

    {

        for(int j=0;j<row;j++)

        {

            if(corelation\_matrix[i][j]<=mean\_column(j))

            {

                corelation\_matrix[i][j]=0;

            }

            else{

                corelation\_matrix[i][j]=1;

            }

        }

    }

}

**update\_corelation\_for\_green\_shade():**

This function updates the correlation matrix by assigning a value of 255 to values of 1 and everything else is being assigned with a value derived from a formula which divides the previous value by the maximum value in the column and then multiplies it by 255. The updated values of the matrix now represent the opacity of the green shade in the visualization with a value between 0 and 255.

void update\_correlation\_for\_green\_shade()

  {

    for(int colum=0;colum<row;colum++)

    {

        double max\_column=max\_column\_elements(colum);

        for(int ro=0;ro<row;ro++)

        {

            if(corelation\_matrix[ro][colum]==1)

            {

                corelation\_matrix[ro][colum]=255;

            }

            else

            {

            corelation\_matrix[ro][colum]=(corelation\_matrix[ro][colum]/max\_column)\*255;

            }

        }

    }

  }

**TASK 2**

**swap\_rows()**

This function is taking two rows as parameters and swapping all the elements in both rows

void swap\_rows(int row1,int row2)

{

   for(int i=0;i<column;i++)

   {

    double temp=array[row1][i];

    array[row1][i]=array[row2][i];

    array[row2][i]=temp;

   }

}

**permute\_data\_matrix()**

This function is continuously swapping different rows in a for loop that runs the number of row times.This function is using (swap\_row\_elements) function to swap the rows .

void permute\_data\_matrix()

{

   for(int i=0;i<row;i++)

   {

    srand(time(0));

    int x=rand()%row;

    if(i!=x)

    {

        swap\_rows(i,x);

    }

   }

}

**sum\_row\_elements()**

This function is returning the sum of all the elements of a particular row of data matrix array.

double sum\_row\_elements(int row1)

{

    double sum=0;

   for(int i=0;i<column;i++)

   {

     sum+=array[row1][i];

   }

   return sum;

}

**mean()**

This function is returning the average value of all elements in a particular row of data matrix .This function is using (sum\_row\_elements) function to get the total of all row elements and the it is dividing that total with the number of columns.

double mean(int row1)

{

   double total= sum\_row\_elements(row1);

    return total/double(column);

}

**signature\_value()**

This function is returning the signature value of a provided row by multiplying the sum of all elements of row with the average of row elements .

double signature\_value(int row)

{

    double total=sum\_row\_elements(row);

    double mean\_=mean(row);

    return total\*mean\_;

}

**SignatureArrayFunction()**

This function is storing the signature values of all rows in a 1D signature array.

void signatureArrayFunction()

{

    signature\_array=new double[row];

    for(int i=0;i<row;i++)

    {

        signature\_array[i]=signature\_value(i);

    }

}

**rearrangeFunction()**

This function is not only sorting the signature array with respect to signature value but also swapping the rows with respect to its signature value .

void rearrangeFunction()

{

   for(int i=0;i<row;i++)

   {

    for(int j=i+1;j<row;j++)

    {

        if(signature\_array[i]>signature\_array[j])

        {

            swap(signature\_array[i],signature\_array[j]);

            swap\_rows(i,j);

        }

    }

   }

}

**TASK 3**

**Weighted\_graph()**

This function is first calling (correlationMatrix) function for creating the correlation matrix.Then it is taking value from user which must lie between -1 and 1 .This is being ensured by while loop.This value is acting as a threshold .The elements of correlation matrix which are below this threshold are being equated to 30000 which is acting as infinity . The is removing the edges between certain nodes.This function is also storing weights of all the nodes, which are calculated by adding all the weights of edges connected to a node.

void weighted\_graph()

   {

      correlationMatrix();

      float value;

      cout<<"Please enter the value between -1 and 1\n";

      cin>>value;

      while(value<-1||value>1)

      {

        cout<<"Invalid input .Please enter the value again\n";

        cin>>value;

      }

      for(int i=0;i<row;i++)

      {

        for(int j=0;j<row;j++)

        {

            if(corelation\_matrix[i][j]<value)

            {

                corelation\_matrix[i][j]=30000;

            }

        }

      }

      node\_array=new double[row];

      for(int i=0;i<row;i++)

      {

        double node\_weight=0;

        for(int j=0;j<row;j++)

        {

            if(corelation\_matrix[i][j]!=30000)

            {

              node\_weight+=corelation\_matrix[i][j];

            }

        }

        cout<<endl;

        node\_array[i]=node\_weight;

      }

      cluster\_check=new bool[row];

      for(int i=0;i<row;i++)

      {

        cluster\_check[i]=false;

      }

   }

**cluster\_complete\_check()**

this function is checking weather all the nodes have become part of any cluster .This function is of bool type.

bool cluster\_complete\_check()

   {

      bool answer=true;

      for(int i=0;i<row;i++)

      {

        if(cluster\_check[i]==false)

        {

            answer=false;

        }

      }

    return answer;

   }

**max\_weight\_index()**

This function is returning the max weighted node which is not a part of any cluster .

int max\_weight\_index()

  {  double max=-2;

     int index=0;

     for( int i=0;i<row;i++)

     {

        if(max<node\_array[i]&&cluster\_check[i]==0)

        {

            cout<<"x";

            max=node\_array[i];

            index=i;

        }

     }

     return index;

  }

**clustering()**

This function has a while loop that will run until all clusters have been created . All the clusters are being stored in a 2D vector named cluster\_array .Inside while loop , a data variable **value** is storing the index of node in correlation matrix which has the maximum weight . Then all the edges connected to that node ,that are not previously part of any cluster, are being pushed to a particular index(starting from zero) of the cluster array .In this way , the 2D vector is storing all the clusters.

void clustering()

  {

      while(!cluster\_complete\_check())

      {

        int value=max\_weight\_index();

        //cout<<"Value "<<value<<endl;

        cluster\_array.resize(counter+1);

        //cluster\_array[counter].push\_back(value);

        for(int i=0;i<row;i++)

        {

            if(corelation\_matrix[value][i]!=30000&&cluster\_check[i]==0)

            {

          //cout<<"hello world";

                cluster\_array[counter].push\_back(i);

                cluster\_check[i]=1;

            }

        }

        counter++;

        cluster\_check[value]=1;

      }

      for(int i=0;i<cluster\_array.size();i++)

      {

        for(int j=0;j<cluster\_array[i].size();j++)

        {

            cout<<cluster\_array[i][j]<<"  ";

        }

        cout<<endl;

      }

  }

**Comparison between task 2 and 3**

In task 2, the data matrix is being permuted and then being color coded after which it is being displayed. After this, the image clusters from task1 are being recovered by using the signature technique on the data matrix after which the recovered image is then color coded and displayed. In task 3, a weighted graph is being created of the permuted matrix after which the edges which are below a certain threshold (provided by user) are removed. After this, the node with the highest weight is forming a cluster with its neighbors and this process repeats until no nodes are left. (A node which has already become a part of a cluster cannot become part of another cluster.) After this the clusters are displayed as an image. The image displayed in task 2 and the one displayed in task 3 are different because the image clusters formed are different because they are obtained through different methods.

**VISUALIZATION PART**

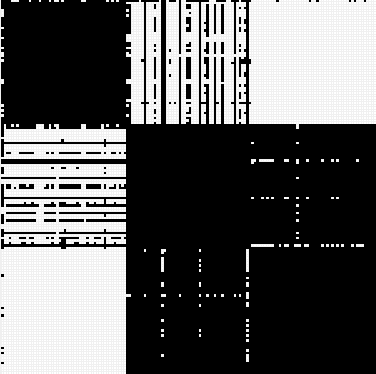
There are 5 tasks that the visualization part has to perform. These Five tasks are:

1. Show Bitmap
2. Visualize Graph
3. Show Correlation Matrix
4. Permute the Data Matrix
5. Signature Function
6. **Bitmap Visualization**

After the calculation of the bitmap from correlation matrix it is printed on the window by the click of a pushbutton. The printing is done through a QPrintEvent Function that prints points on the screen coordinates according to the bitmap values.

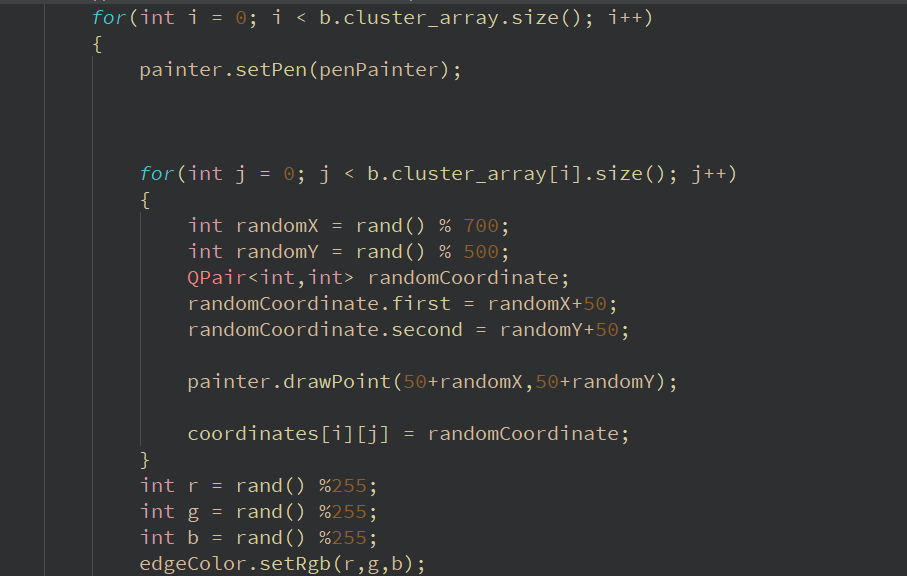


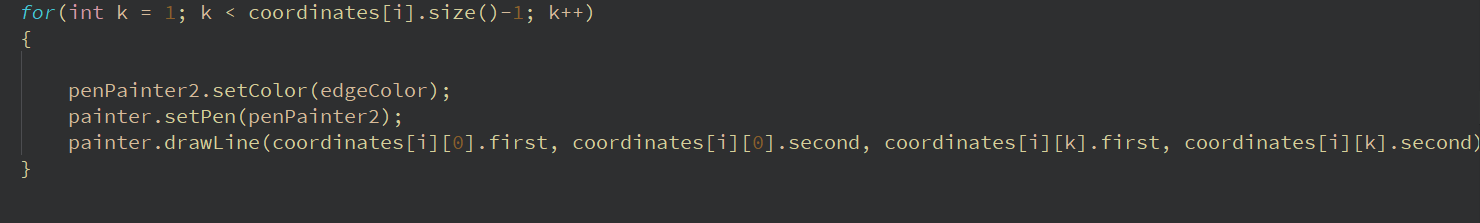
**QT Output:**



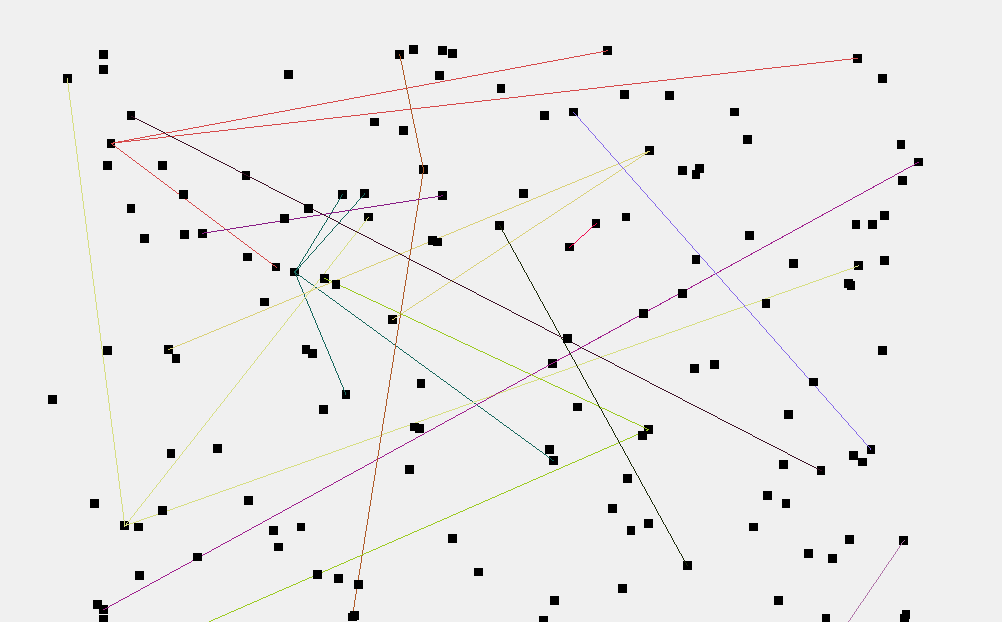
1. **Visualize Graph Clusters**

When clusters are formed from the correlation matrix using the threshold set by input option they are all displayed at the same time on the screen separated by different colors showing that they belong to different clusters. This makes it to see all data at the same time

****

****

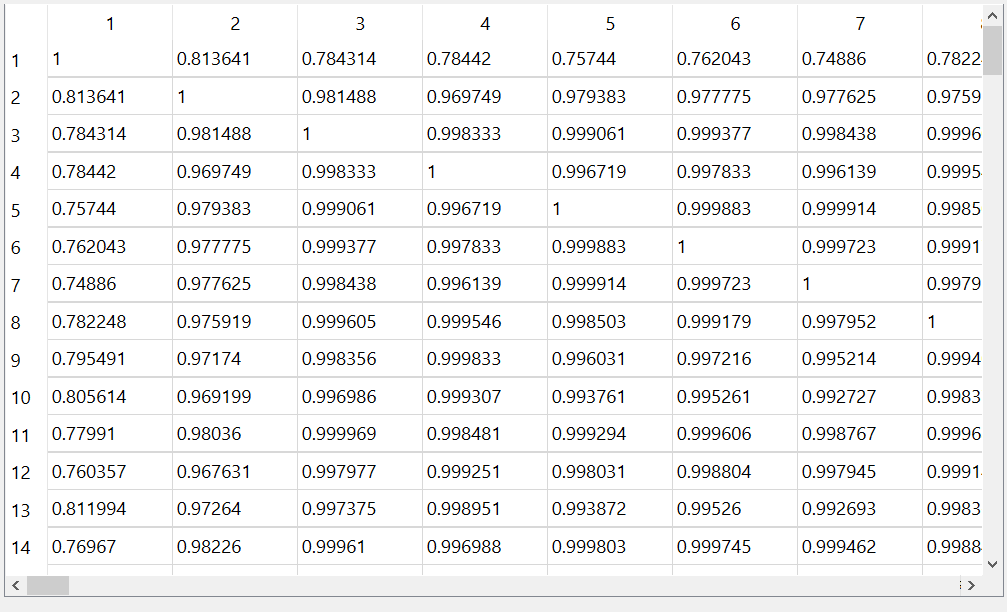
**QT Output:**

****

1. **Show Correlation Matrix**

The correlation matrix is also displayed in the visualization part in the form of table with a size NxN where N is the number of rows.

**Qt Output:**

****

1. **Permute Data Matrix:**

There is a pushbutton on the screen which runs the backend function of the permutation of data matrix and calculates the new correlation matrix and bitmap.

1. **Signature Function:**

Same case as permutation of data matrix a pushbutton performs the task of calculating a signature value.

**Contributions**

Visualization(qt) : Affan Ali khan 100%

Coding : Hamza Wajid 40%,

Muhammad Saadullah 40%

Sheharyar Sadiq 20%

Debugging: Sheharyar Sadiq 50%

Hamza Wajid 20%

Muhammad Saadullah 30%

Code modification with respect to qt: Hamza Wajid 60%

Muhammad Saadullah 40%

Report: Sheharyar Sadiq 100%