Data clustering

Fuzzy c-means clustering

by:

CN

AI (Semester 5)

Table Of Contents

1 Introduction

The most pertinent functions will be discussed as they related to the algorithm directly. Discussion of auxiliary functions like printing or retreving data from a .txt file will be omitted in the breakdown sections. The entire code is in the *Code* section of this report.

2 fuzzy.h breakdown

```cpp
std::vector<std::vector<double>> partitionInit(int c, int len){

 std::vector<std::vector<double>> U(c ,std::vector<double>(len, 0));

 int n;
 int max = 10;
 int error=0;
 for(size_t i = 0; i<U[0].size(); i++){
  for(size_t j = 0; j<U.size()-1; j++){
   n = r(max);
   max -= n;
   U[j][i] = static_cast<double>(n) / 10.0;
   error += n;
  }
 U[U.size()-1][i] = static_cast<double>(10.0-error)/10.0;
 n = 0;
 error=0;
 max = 10;
}
 //printM(U);
 return U;
};
```

fuzzy.h

- Initializes the partition matrix $U_f \in \mathbb{R}^{c \times N}$ satisfying the following conditions.

$$\underset{\substack{1 \le i \le c \\ 1 \le k \le N}}{\forall} u_{ik} \in [0, 1]$$

$$\underset{1 \le k \le N}{\forall} \sum_{i=1}^{c} u_{ik} = 1$$

$$\underset{1 \le i \le c}{\forall} \sum_{k=1}^{N} u_{ik} \in [0, N]$$

```cpp
std::vector<std::vector<double>> clusterCenters(int c, double m, const std::vector<std::vector<double>> X, const
std::vector<std::vector<double>> U){

 std::vector<std::vector<double>> V(X.size() ,std::vector<double>(c, 0.0));

 double n;
 double d;
 size_t row = U.size();
 size_t col = U[0].size();
 std::vector<double> v;

 for(size_t i = 0; i<X.size(); i++){
  //std::vector<double> u = U[i];
  std::vector<double> x = X[i];
  for(size_t I = 0; I<row; I++){
   for(size_t J = 0; J<col; J++){
    n += pow(U[I][J],m)*x[J];
    d += pow(U[I][J],m);
   }
   v.push_back(n/d);
   n = 0;
   d=0;
  }
 }
```

fuzzy.h

The cluster center matrix v_i is computed by

$$\underset{1\leq i\leq c}{\forall} v_i = \frac{\sum_{k=1}^{N} u_{ik}^m \, \mathbf{x}_k}{\sum_{k=1}^{N} u_{ik}^m} \tag{1}$$

```cpp
std::vector<std::vector<double>> euclideanNorm (int c, const std::vector<std::vector<double>> X, const
std::vector<std::vector<double>> V){

 size_t row = X.size();
 size_t col = X[0].size();
 std::vector<std::vector<double>> D(c, std::vector<double>(col,0));
 double d;

 for(size_t i = 0; i<c; i++){
  for(size_t J = 0; J<col; J++){
   for(size_t I = 0; I<row; I++){
     d+=pow((X[I][J] - V[I][i]),2);
   }
   D[i][J] = sqrt(d);
   d = 0;
  }
 }

 return D;
 }
```

fuzzy.h

We compute the distance d_{cxN} between the data points X_{ixN} to the prototypes V_{ixc} using the euclidean norm

$$d_{cxN} = \|\mathbf{x}_N - \mathbf{v}_c\|^2 = (\mathbf{x}_N - \mathbf{v}_c)^T(\mathbf{x}_N - \mathbf{v}_c) \tag{2}$$

```
double fcmCriterion(int m, const std::vector<std::vector<double>> U, const std::vector<std::vector<double>> D) {
 printM(U,"U");
 double criterion;
 size_t row = U.size();
 size_t col = U[0].size();
   for(size_t i = 0; i<row; i++){
    for(size_t j = 0; j<col; j++){
     criterion += pow(U[i][j],m) * pow(D[i][j],2);
    }
 }
 return criterion;
}
```

fuzzy.h

We can compute the criterion function J_m

$$J_m(U_f, V_f) = \sum_{i=1}^{c} \sum_{k=1}^{N} (u_{ik})^m \, d_{ik}^2 \tag{3}$$

```cpp
std::vector<std::vector<double>> updateU (int m,std::vector<std::vector<double>> D){

 size_t row = D.size();
 size_t col = D[0].size();
 std::vector<std::vector<double>> U(row, std::vector<double>(col, 0.0));
 double d;

 for (size_t i = 0; i < row; i++) {
  for (size_t k = 0; k < col; k++) {
   U[i][k] = pow(D[i][k],2/(1-m));
   for (size_t j = 0; j < row; j++) {
    d+= pow(D[j][k],2/(1-m));
   }

   U[i][k] /= d;
   d = 0;
  }
 }

 return U;
}
```

fuzzy.h

Finally we update the partition matrix U_{cxN}. Since there are no elements with a distance of zero to the prototypes, $I_k = \emptyset$ and U_{cxN} is

$$\underset{1\leq i\leq c}{\forall} U_{ik} = \frac{d_{ik}^{\frac{2}{1-m}}}{\sum_{j=1}^{c} d_{ik}^{\frac{2}{1-m}}} \tag{4}$$

the function *updateU* computes a simplification of this equation example

$$U_{11} = \frac{d_{11}^{\frac{2}{1-m}}}{d_{11}^{\frac{2}{1-m}} + d_{21}^{\frac{2}{1-m}}} = \frac{1}{1 + \left(\frac{d_{11}}{d_{21}}^2\right)} \tag{5}$$

```cpp
#include <iostream>
#include <vector>

#include "fuzzy.h"
#include "readFromFile.h"

double fuzzyClustering(int,double,int,std::vector<std::vector<double>>&);

int main(){

 std::vector<std::vector<double>> X = readDataFromFile("exampleSix.txt");

 double e = 10E-10;

 fuzzyClustering(2,e,2,X);

}

double fuzzyClustering(int m, double e, int C, std::vector<std::vector<double>> &X){
//*****SETUP
 int len = X[0].size();
 int c  = C;
 std::vector<std::vector<double>> U = partitionInit(c,len);
 //U ={ {0.5,0.7, 0.0, 0.0, 1.0, 0.4, 0.6, 0.2},{0.5, 0.3, 1.0, 1.0, 0.0, 0.6, 0.4, 0.8}};

 //*****1st iteration
 std::vector<std::vector<double>> V = clusterCenters(c,m,X,U);

 std::vector<std::vector<double>> D = euclideanNorm(c,X,V);

 double FCM_Criterion= fcmCriterion(m,U,D);

 double tempCriterion;

 //*****nth iteration
 do{
  U = updateU(m,D);
  V = clusterCenters(c,m,X,U);
  D = euclideanNorm(c,X,V);
  tempCriterion = FCM_Criterion;
  FCM_Criterion = fcmCriterion(m,U,D);
 }while(abs(tempCriterion) > FCM_Criterion - e);

 std::cout<<"criterion "<<FCM_Criterion<<"\n";

 return FCM_Criterion;
}
```

FCM.cc

This is the principle function in the main file *FCM.cc*. This function encapsulates the functions previously discussed. Successive computations of the FCM criterion are made until the desired accuracy is acheived. This is acheived by using the algorithm provided in the instruction to this lab

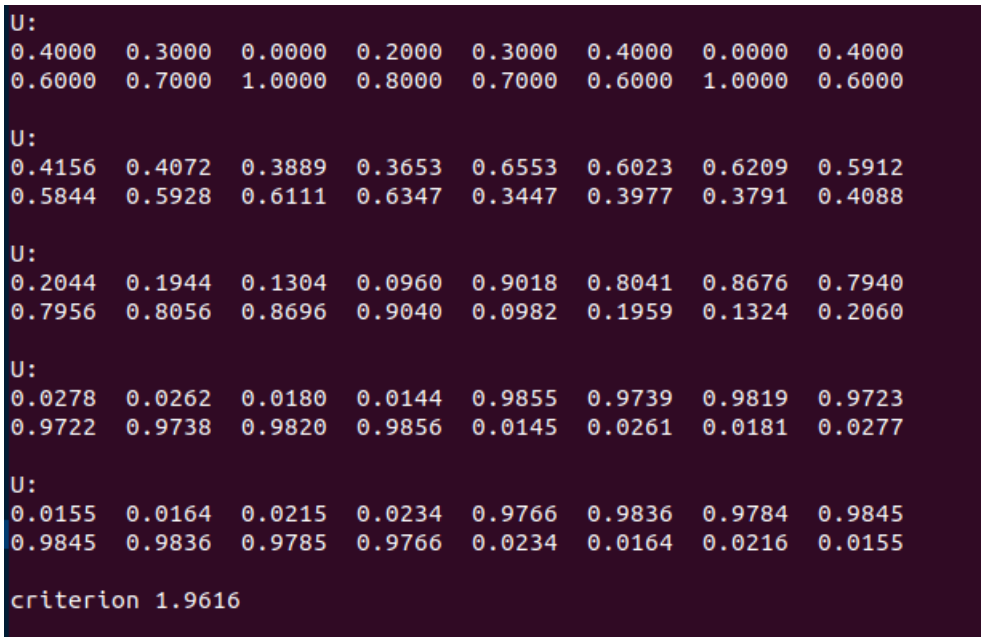
- 1. Set: $c(1 < c < N)$, $m \in (1, +\infty)$, and the iteration index $t = 0$. Initialize the fuzzy partition matrix $U_f^{(0)}$ which fulfills the conditions mentioned above.
- 2. Based on (1), calculate the cluster centers $V_f^{(t)} = [v_1^{(t)}, v_2^{(t)}, \ldots, v_c^{(t)}]$ using $U_f^{(t)}$.
- 3. Update the fuzzy partition matrix $U_f^{(t+1)}$ based on (4).
- 4. If $||U_f^{(t+1)} - U_f^{(t)}|| > \varepsilon$, then $t \leftarrow t + 1$ and go to step 2.

Please note that a function was written from *readFromFile.h*. This is used to include the data from different tasks while keeping the main function clean.

3 Program Output: Exercise 2

We demonstrate the functionality of this program using m=2, c=2, e= 1×10^{-10} and the following data set

```
X =  2.5, 3.0, 3.0, 3.5, 5.5, 6.0, 6.0, 6.5
     3.5, 3.0, 4.0, 3.5, 5.5, 6.0, 5.0, 5.5
```



output FCM.cc, exercise 2, c = 2

Here we see that after four iterations plus the initialization phase, the FCM criterion is 1.9616.

We repeat this same task but for number of clusters = 3

```
0.8972  0.9052  0.9430  0.9658  0.0081  0.0340  0.0165  0.0374
0.0655  0.0608  0.0382  0.0233  0.0878  0.1766  0.1375  0.1826
0.0373  0.0340  0.0189  0.0109  0.9041  0.7894  0.8460  0.7799

U:
0.9653  0.9642  0.9557  0.9532  0.0075  0.0112  0.0085  0.0113
0.0182  0.0188  0.0234  0.0249  0.6341  0.3691  0.5889  0.3677
0.0165  0.0170  0.0208  0.0219  0.3584  0.6198  0.4027  0.6210

criterion 1.4323

$ aa
U:
0.0000  0.6000  0.1000  0.3000  0.4000  0.7000  0.6000  0.2000
0.6000  0.3000  0.4000  0.3000  0.2000  0.2000  0.1000  0.6000
0.4000  0.1000  0.5000  0.4000  0.4000  0.1000  0.3000  0.2000

U:
0.1553  0.1563  0.1076  0.1013  0.7273  0.5933  0.7007  0.5935
0.3655  0.3734  0.3488  0.3666  0.1523  0.2204  0.1675  0.2212
0.4793  0.4703  0.5436  0.5321  0.1204  0.1863  0.1318  0.1853

U:
0.0318  0.0297  0.0123  0.0070  0.9646  0.9292  0.9561  0.9265
0.2935  0.3017  0.2640  0.2779  0.0206  0.0400  0.0254  0.0414
0.6747  0.6685  0.7236  0.7151  0.0148  0.0308  0.0185  0.0321

U:
0.0084  0.0085  0.0107  0.0111  0.9549  0.9668  0.9582  0.9684
0.4869  0.5214  0.4672  0.5052  0.0225  0.0166  0.0209  0.0158
0.5046  0.4701  0.5221  0.4838  0.0226  0.0166  0.0209  0.0158

criterion 1.4567

$
```

output FCM.cc, exercise 2, c = 3

4 Program Output: Exercise 5

We demonstrate the functionality of this program using m=2, c=2, e= 1×10^{-10} and the following data set

```
X =  2.5, 3.0, 3.0, 3.5, 5.5, 6.0, 6.0, 4.5
     3.5, 3.0, 4.0, 3.5, 5.5, 6.0, 5.0, 4.5
```

```
U:
0.8867  0.9335  0.8908  0.9933  0.1429  0.2149  0.1861  0.0289
0.1133  0.0665  0.1092  0.0067  0.8571  0.7851  0.8139  0.9711

U:
0.9669  0.9705  0.9698  0.9761  0.0138  0.0714  0.0386  0.2963
0.0331  0.0295  0.0302  0.0239  0.9862  0.9286  0.9614  0.7037

U:
0.9781  0.9779  0.9745  0.9739  0.0041  0.0379  0.0231  0.3884
0.0219  0.0221  0.0255  0.0261  0.9959  0.9621  0.9769  0.6116

criterion 3.0194

$ aa
U:
0.4000  0.1000  0.5000  0.8000  0.1000  0.0000  0.2000  0.4000
0.6000  0.9000  0.5000  0.2000  0.9000  1.0000  0.8000  0.6000

U:
0.8568  0.8726  0.9226  0.9734  0.1511  0.2285  0.1826  0.0555
0.1432  0.1274  0.0774  0.0266  0.8489  0.7715  0.8174  0.9445

U:
0.9651  0.9667  0.9731  0.9769  0.0147  0.0734  0.0392  0.3013
0.0349  0.0333  0.0269  0.0231  0.9853  0.9266  0.9608  0.6987

U:
0.9779  0.9776  0.9748  0.9741  0.0041  0.0378  0.0230  0.3900
0.0221  0.0224  0.0252  0.0259  0.9959  0.9622  0.9770  0.6100

criterion 3.0189
```

output FCM.cc, exercise 5, c = 2

Here we see that after three iterations plus the initialization phase, the FCM criterion is about 3.02.

We repeat this same task but for number of clusters = 3

```
0.6366  0.6141  0.7934  0.7671  0.1388  0.1827  0.1455  0.0052
0.1645  0.1730  0.0903  0.0986  0.4998  0.4522  0.4857  0.2036
0.1988  0.2128  0.1162  0.1342  0.3614  0.3650  0.3688  0.7911

U:
0.9087  0.9042  0.9426  0.9388  0.0076  0.0467  0.0260  0.0495
0.0334  0.0345  0.0190  0.0195  0.9287  0.7417  0.8005  0.0857
0.0579  0.0613  0.0384  0.0417  0.0637  0.2116  0.1735  0.8647

U:
0.9398  0.9344  0.9059  0.8897  0.0074  0.0196  0.0210  0.0069
0.0168  0.0178  0.0217  0.0233  0.9455  0.9044  0.8662  0.0085
0.0434  0.0478  0.0724  0.0870  0.0470  0.0760  0.1128  0.9846

criterion 1.5159

$ aa
U:
0.4000  1.0000  1.0000  0.7000  0.1000  0.2000  0.4000  0.2000
0.2000  0.0000  0.0000  0.0000  0.8000  0.5000  0.1000  0.0000
0.4000  0.0000  0.0000  0.3000  0.1000  0.3000  0.5000  0.8000

U:
0.8554  0.8766  0.9068  0.9658  0.0002  0.0305  0.0448  0.0037
0.0419  0.0338  0.0227  0.0073  0.9987  0.8758  0.7704  0.0038
0.1027  0.0896  0.0705  0.0269  0.0011  0.0937  0.1848  0.9925

U:
0.9303  0.9298  0.8946  0.8855  0.0078  0.0163  0.0258  0.0002
0.0178  0.0171  0.0214  0.0206  0.9523  0.9282  0.8573  0.0002
0.0520  0.0530  0.0840  0.0939  0.0400  0.0554  0.1169  0.9997

criterion 1.5125

$
```

output FCM.cc, exercise 5, c = 3

5 Program Output: Exercise 6

We demonstrate the functionality of this program using m=2, c=2, e= 1×10^{-10} and the following data set

```
X =  2.5, 3.0, 3.0, 3.5, 5.5, 6.0, 6.0, 6.5, 25.0
     3.5, 3.0, 4.0, 3.5, 5.5, 6.0, 5.0, 5.5, 25.0
```

```
U:
0.0117  0.0110  0.0068  0.0061  0.0025  0.0076  0.0038  0.0090  0.9409
0.9883  0.9890  0.9932  0.9939  0.9975  0.9924  0.9962  0.9910  0.0591

U:
0.0052  0.0047  0.0028  0.0022  0.0026  0.0061  0.0032  0.0068  1.0000
0.9948  0.9953  0.9972  0.9978  0.9974  0.9939  0.9968  0.9932  0.0000

U:
0.0051  0.0046  0.0027  0.0022  0.0026  0.0062  0.0033  0.0069  1.0000
0.9949  0.9954  0.9973  0.9978  0.9974  0.9938  0.9967  0.9931  0.0000

criterion 27.8667

$ aa
U:
1.0000  0.8000  0.5000  1.0000  0.1000  1.0000  0.5000  0.3000  0.1000
0.0000  0.2000  0.5000  0.0000  0.9000  0.0000  0.5000  0.7000  0.9000

U:
0.9801  0.9820  0.9913  0.9936  0.9467  0.8891  0.9376  0.8798  0.3002
0.0199  0.0180  0.0087  0.0064  0.0533  0.1109  0.0624  0.1202  0.6998

U:
0.9932  0.9937  0.9962  0.9968  0.9978  0.9943  0.9970  0.9934  0.0038
0.0068  0.0063  0.0038  0.0032  0.0022  0.0057  0.0030  0.0066  0.9962

U:
0.9949  0.9954  0.9973  0.9978  0.9974  0.9938  0.9967  0.9931  0.0000
0.0051  0.0046  0.0027  0.0022  0.0026  0.0062  0.0033  0.0069  1.0000

criterion 27.8667
```

output FCM.cc, exercise 6, c = 2

Here we see that after three iterations plus the initialization phase, the FCM criterion is about 27.9. This is due to the outlier value of 25. A better result is obtained by increasing the number of clusters.

We repeat this same task but for number of clusters = 3

```
0.0003  0.0003  0.0003  0.0003  0.0003  0.0003  0.0003  0.0003  1.0000
0.9845  0.9836  0.9780  0.9759  0.0235  0.0162  0.0216  0.0154  0.0000
0.0152  0.0162  0.0217  0.0238  0.9762  0.9834  0.9781  0.9843  0.0000

criterion 1.9610

$ aa
U:
1.0000  0.0000  1.0000  0.1000  0.0000  0.2000  0.5000  0.1000  0.3000
0.0000  0.4000  0.0000  0.9000  0.7000  0.1000  0.2000  0.6000  0.2000
0.0000  0.6000  0.0000  0.0000  0.3000  0.7000  0.3000  0.3000  0.5000

U:
0.6637  0.6320  0.7488  0.6919  0.1758  0.2538  0.1772  0.2374  0.2742
0.2974  0.3284  0.2299  0.2851  0.7982  0.6413  0.7861  0.6462  0.2906
0.0389  0.0396  0.0214  0.0230  0.0260  0.1050  0.0367  0.1164  0.4352

U:
0.8486  0.8457  0.9104  0.9098  0.1606  0.0104  0.1545  0.0284  0.0164
0.1476  0.1506  0.0879  0.0886  0.8385  0.9894  0.8444  0.9712  0.0196
0.0038  0.0037  0.0017  0.0016  0.0009  0.0001  0.0011  0.0004  0.9640

U:
0.9803  0.9800  0.9815  0.9812  0.0233  0.0166  0.0231  0.0169  0.0000
0.0194  0.0197  0.0183  0.0186  0.9764  0.9831  0.9766  0.9828  0.0000
0.0003  0.0003  0.0002  0.0002  0.0003  0.0003  0.0003  0.0004  1.0000

U:
0.9845  0.9836  0.9781  0.9761  0.0239  0.0161  0.0218  0.0151  0.0000
0.0152  0.0162  0.0216  0.0237  0.9758  0.9836  0.9778  0.9845  0.0000
0.0003  0.0003  0.0003  0.0003  0.0003  0.0003  0.0003  0.0003  1.0000

criterion 1.9610

$
```

output FCM.cc, exercise 6, c = 3

6 Conclusion

- **Pros and Cons**

  FCM gives the best result for overlapped data sets and comparatively better then the k-means algorithm.
  Unlike k-means where data points must exclusively belong to one cluster center here the data points are assigned membership to each cluster center as a result of which data point may belong to more then one cluster center.

- **Cons**

  Apriori specification of the number of clusters.
  Euclidean distance computations could be troublesome.
  The performance of the FCM algorithm depends on the selection of the initial cluster center and/or the initial membership value.

7 Source Code

7.1 fuzzy.h

```
#ifndef FUZZY_H
#define FUZZY_H

#include <iomanip>
#include <vector>
#include <random>
#include <cmath>
#include <fstream>
#include <limits>

//*****print matrix
template <typename T>
void printM(const std::vector<std::vector<T>> &matrix, const std::string& title =
    if (!title.empty()) {
        std::cout << title << ":" << std::endl;
    }

    std::cout << std::setprecision(4) << std::fixed;
    for (const auto& row : matrix) {
        for (const auto& element : row) {
            std::cout << std::setprecision(4) << element << "  ";
        }
        std::cout << std::endl;
    }

    std::cout << std::endl;
}

//*****print vector
template <typename T>
void printV(const std::vector<T> &v){
 std::cout << std::endl;
 std::cout << std::setprecision(4) << std::fixed;
 for(const auto& element : v){
   std::cout << std::setprecision(8) << element << "  ";
  }
}

//random number with range
int r(int max){
    std::uniform_int_distribution<int> distribution(0, max);

 // Create a random number generator
 std::random_device rd;
 std::mt19937 gen(rd());
```

```
 return distribution(gen);
}

//Initialize partition matrix U_f
//fuzziness !
std::vector<std::vector<double>> partitionInit(int c, int len){

 std::vector<std::vector<double>> U(c ,std::vector<double>(len, 0));

 int n;
 int max = 10;
 int error=0;
 for(size_t i = 0; i<U[0].size(); i++){
  for(size_t j = 0; j<U.size()-1; j++){
   n = r(max);
   max -= n;
   U[j][i] = static_cast<double>(n) / 10.0;
   error += n;
  }
 U[U.size()-1][i] = static_cast<double>(10.0-error)/10.0;
 n = 0;
 error=0;
 max = 10;
}
 //printM(U);
 return U;
};

//*****clusterCenters v

std::vector<std::vector<double>> clusterCenters(int c, double m, const std::vector

 std::vector<std::vector<double>> V(X.size() ,std::vector<double>(c, 0.0));

 double n;
 double d;
 size_t row = U.size();
 size_t col = U[0].size();
 std::vector<double> v;

 for(size_t i = 0; i<X.size(); i++){
  //std::vector<double> u = U[i];
  std::vector<double> x = X[i];
  for(size_t I = 0; I<row; I++){
   for(size_t J = 0; J<col; J++){
    n += pow(U[I][J],m)*x[J];
    d += pow(U[I][J],m);
   }
   v.push_back(n/d);
   n = 0;
   d=0;
  }
```

```
 }

 //printV(v);
 //std::cout<<std::endl;
 for (size_t i = 0; i < X.size(); ++i) {
  for (size_t j = 0; j < c; ++j) {

   V[i][j] = v[i * c + j];
   //std::cout<<V[i][j]<<std::endl;
  }
 }
  //printM(V);
 return V;
}

//*****Euclidean norm
//Distance of data points to prototypes
std::vector<std::vector<double>> euclideanNorm (int c, const std::vector<std::vect

 size_t row = X.size();
 size_t col = X[0].size();
 std::vector<std::vector<double>> D(c, std::vector<double>(col,0));
 double d;

 for(size_t i = 0; i<c; i++){
  for(size_t J = 0; J<col; J++){
   for(size_t I = 0; I<row; I++){
     d+=pow((X[I][J] - V[I][i]),2);
   }
   D[i][J] = sqrt(d);
   d = 0;
  }
 }

 return D;
 }

double fcmCriterion(int m, const std::vector<std::vector<double>> U, const std::ve
 printM(U,"U");
 double criterion;
 size_t row = U.size();
 size_t col = U[0].size();
   for(size_t i = 0; i<row; i++){
    for(size_t j = 0; j<col; j++){
     criterion += pow(U[i][j],m) * pow(D[i][j],2);
    }
 }
 return criterion;
}

std::vector<std::vector<double>> updateU (int m,std::vector<std::vector<double>> I
```

```
 size_t row = D.size();
 size_t col = D[0].size();
 std::vector<std::vector<double>> U(row, std::vector<double>(col, 0.0));
 double d;

 for (size_t i = 0; i < row; i++) {
  for (size_t k = 0; k < col; k++) {
   U[i][k] = pow(D[i][k],2/(1-m));
   for (size_t j = 0; j < row; j++) {
    d+= pow(D[j][k],2/(1-m));
   }

   U[i][k] /= d;
   d = 0;
  }
 }

 return U;
}

//**************** to python

// Function to append matrix to a file
void appendToFile(const std::vector<std::vector<double>>& m, const std::string& fi
 std::ofstream file(filename, std::ios::app);// Use app flag to append existing fi
 if (file.is_open()) {
  for (const auto& row : m) {
   for (int point : row) {
    file << point << " ";
   }
   file << std::endl;
  }
  file.close();
 }
 else {
  std::cerr << "Error opening file: " << filename << std::endl;
 }
}

void makeColZero(int col, std::vector<std::vector<double>> &tempProb) {
 for (size_t i = 0; i < tempProb.size(); ++i) {
  tempProb[i][col] = 0.0;
 }
}

std::vector<std::vector<double>> normalizeU(std::vector<std::vector<double>> U){

 size_t row = U.size();
 size_t col = U[0].size();
 std::vector<std::vector<double>> N(row, std::vector<double>(col, 0.0));
 double d=0;
 int I=0;
```

```
 for (size_t j = 0; j < col; j++) {
  for (size_t i = 0; i < row; i++) {
   if(U[i][j] > d){
    d = U[i][j];
    I = i;
   }
  }
  N[I][j] = 1;
  I = 0;
 }

 return N;
}


#endif
```

7.2 FCM.cc

```
#include <iostream>
#include <vector>

#include "fuzzy.h"
#include "readFromFile.h"

double fuzzyClustering(int,double,int,std::vector<std::vector<double>>&);

int main(){

 std::vector<std::vector<double>> X = readDataFromFile("exampleSix.txt");

 double e = 10E-10;

 fuzzyClustering(3,e,3,X);

}

double fuzzyClustering(int m, double e, int C, std::vector<std::vector<double>> &X
//*****SETUP
 int len = X[0].size();
 int c  = C;
 std::vector<std::vector<double>> U = partitionInit(c,len);
 //U ={ {0.5,0.7, 0.0, 0.0, 1.0, 0.4, 0.6, 0.2},{0.5, 0.3, 1.0, 1.0, 0.0, 0.6, 0.4

 //*****1st iteration
 std::vector<std::vector<double>> V = clusterCenters(c,m,X,U);

 std::vector<std::vector<double>> D = euclideanNorm(c,X,V);

 double FCM_Criterion= fcmCriterion(m,U,D);

 double tempCriterion;
```

```
 //*****nth iteration
 do{
  U = updateU(m,D);
  V = clusterCenters(c,m,X,U);
  D = euclideanNorm(c,X,V);
  tempCriterion = FCM_Criterion;
  FCM_Criterion = fcmCriterion(m,U,D);
 }while(abs(tempCriterion) > FCM_Criterion - e);

 std::cout<<"criterion "<<FCM_Criterion<<"\n";

 return FCM_Criterion;
}
```

7.3 readDataFromFile.h

```
#ifndef READFROMFILE_H
#define READFROMFILE_H

#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>

std::vector<std::vector<double>> readDataFromFile(const std::string& filename) {
 std::vector<std::vector<double>> X;

 // Open the text file
 std::ifstream inputFile(filename);

 // Check if the file is open
 if (!inputFile.is_open()){
  std::cerr << "Error opening the file." << std::endl;
  return X;
 }

 // Read each line from the file
 std::string line;
 while(std::getline(inputFile, line)){
  std::vector<double> row;
  std::istringstream iss(line);

  // Read each comma-separated value from the line
  double value;
  char comma;

  while (iss >> value) {
   row.push_back(value);
   // Check for a comma and ignore it
   if(iss >> comma && comma != ','){
    std::cerr << "Error: Expected comma." << std::endl;
```

```
    return X;
   }
  }

  // Add the row to the vector
  X.push_back(row);
 }

 // Close the file
 inputFile.close();

 return X;
}


#endif
```