EXTENSION NEURAL NETWORK WITH PSO OPTIMIZATION

EEL706 PROJECT 2012-13

Submitted by:

ARPIT GUPTA(2010MT50590)

KESHAV GOYAL(2010MT50599)

DIGVIJAY TRIGHATIA(2010MT50593)

AMEYA SHENDRE(2010MT50616)

T NEERAJ KUMAR(2010MT50625)

INTRODUCTION

Neural networks as we have studied are broadly divided into two main classes- supervised learning and unsupervised learning. Supervised learning is a process that incorporates an external teacher and environmental information, so it requires an external goal output to respond to input data. A supervised learning neural network (NN) can estimate a relation function between the inputs and outputs from a learning process and also can discover mapping from feature space into a space of classes. Unsupervised learning is a process that incorporates no external teacher; it results in exposition of clusters for given input patterns. The goal of cluster analysis is to partition a set of patterns into a group of desired subsets.

However the applicability domain of neural networks is more or less restricted in terms of stability and plasticity of the memory system. Therefore a neural network topology called Extensive Neural Networks (ENN) is being used. This is because it permits classification of problems whose features are defined over an interval of values in our world, supervised learning, discrete or continuous output. Basically the term **'extension'** is used in the method because it works for negative membership function values as well. E.g. Height could only be positive but account balance can take all types of floating decimal point values. It uses a modified **extension distance** (ED) to measure the similarity between data and cluster center. This ENN has lower memory consumption and shorter learning times than traditional neural networks in applications.

Phase-I of our project focuses on applying ENN-2 to various data sets and obtaining the clustering results. ENN-2 is a more developed version of the original ENN proposed by M. H. Wang in his IEEE paper.

However this ENN does not work well on large datasets so we have come up with a different approach wherein we update the weights in clusters through particle swarm optimization(PSO). This is what we take up in **Phase-II** of our project. Typically, in a PSO algorithm each particle keeps track of the coordinates in the search space which are associated with the best solution it has found so far. The corresponding value of the objective function (fitness value) is also stored. Another"best" value that is tracked by each particle is the best value obtained so far by any particle in its topological neighborhood. When a particle takes the whole population as its neighbors, the best value is a global best. At each iteration of the PSO algorithm the velocity of each particle is changed towards the personal and global best (or neighborhood best) locations. But also some random component is incorporated into the velocity update. This approach works well on large datasets unlike the ENN-2 approach which worked accurately for small datasets only.

PHASE-I

ENN-2 (EXTENSION NEURAL NETWORKS TYPE-2)

Extension Neural Networks(ENN) does not require an initial guess of the cluster center coordinates, nor of the initial number of clusters. ENN was used in:

- Failure detection in machinery.
- Tissue classification through MRI.
- Fault recognition in automotive engine.
- State of charge estimation in lead-acid battery.
- Classification with incomplete survey data.

We use a slight variation of the ENN called ENN-2 (reference *M. H. Wang's ENN-2 and its applications*). We attempt to code this in JAVA and test it on different data sets for performance analysis. A Structure of ENN (ENN-2 used here) we used is illustrated with the help of diagram given below. It consists of an input layer and an output layer. The connections between the jth input node and the mth are w_{mj}^{u} and w_{mj}^{l}.

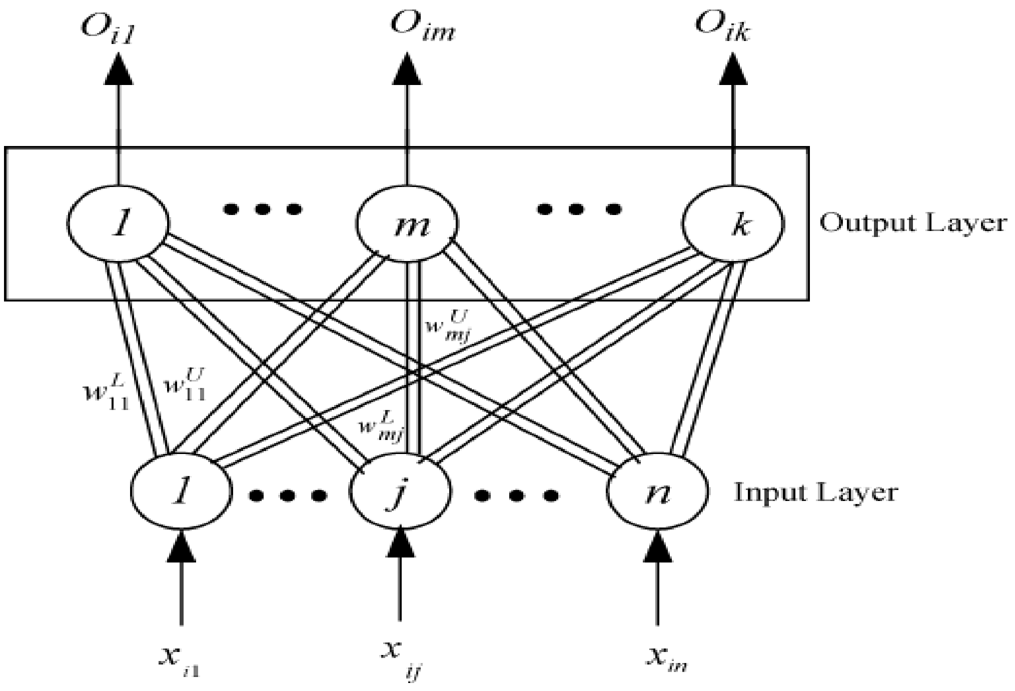


Figure 1 -Topology of the Extension Neural Network

ALGORITHM:

The algorithm for the process is given as follows:

Variables Defined:

M_k: Number of patterns belonging to cluster k
N_p: Total number of the input pattern
n: Number of features
k: Number of existing clusters
X_i: ith pattern
x_{ij}: jth feature of the ith input pattern
Z_k: Centre of the cluster k
λ: Distance parameter

1. **1.** Set the desired DP λ.
2. **2.** Produce the first pattern and M_1=1. The centre coordinates and weights of the first cluster are calculated as

$k=1$ (1)

$Z_k= X_k \Rightarrow \{z_{k1}, z_{k2}, \ldots\ldots, z_{kn}\} = \{x_{k1}, x_{k2}, \ldots\ldots, x_{kn}\}$ (2)

${}^{L}_{kj}w= z_{kj}-\lambda$; for j=1,2…n (3)

${}_{kj}{}^{U}w= z_{kj}+\lambda$; for j = 1,2…n (4)

3. **3.** Read the input pattern vectors (put i=1), and go to next step.
4. **4.** Read next input pattern Xi= $\{x_{i1}, x_{i2}\ldots\ldots x_{in}\}$ and calculate the Extension distance ED_p between X_i and the existing pth cluster center as:

$$EDp = \sum_{j=1}^{n}\left[\frac{|x_{ij} - z_{pj}| - ({}^{U}_{pj}w - {}^{L}_{pj}w)/2}{\left|\frac{{}^{U}_{pj}w - {}^{L}_{pj}w}{2}\right|} + 1\right]$$

; for p = 1,2…k; (5)

5. **5.** Find *s* for which,

$ED_s= \min \{ED_p\}$. (6)

If $ED_s> n$, then create a new cluster center. $ED_s>n$ expresses that X_i does not belong to sth cluster. Then a new cluster center will be created.

$k = k+1$ (7)

$M_k = 1$ (8)

$Z_k = X_i \Rightarrow \{z_{k1}, z_{k2}\ldots., z_{kn}\} = \{x_{i1}, x_{i2}\ldots., x_{in}\}$ (9)

${}^{L}_{kj}w = z_{kj} - \lambda$; for j=1, 2…n (10)

$^{U}_{kj}w = z_{kj} + \lambda$; for j = 1, 2….n (11)

Else, the pattern X_i belongs to the cluster *s*, and updates the weight and centre of clusters. Update them accordingly for n+1 number of points.

$$^{U(new)}_{sj}W = {}^{U(old)}_{sj}W + \frac{1}{M_s+1}(x_{ij} - {}^{old}_{sj}Z); \quad (12)$$

$$^{L(new)}_{sj}W = {}^{L(old)}_{sj}W + \frac{1}{M_s+1}(x_{ij} - {}^{old}_{sj}Z); \quad (13)$$

$$^{new}_{sj}Z = \frac{{}^{U(new)}_{qj}w + {}^{L(new)}_{qj}w}{2}; \quad \text{for } j = 1, 2…., n \quad (14)$$

$M_s = M_s + 1$; for all $p \neq S$; (15)

6. **6.** If input pattern X_i changes from the cluster “q” (old one) to “k” (new one), then the weights and center of cluster “q” (for n-1 points since the point has now jumped to the new cluster) are modified as, and "k"(for n+1 points) modified in the same way as above;

$$^{U(new)}_{qj}W = {}^{U(old)}_{qj}W - \frac{1}{M_q}(xij - {}^{old}_{qj}Z); \quad (16)$$

$$^{L(new)}_{qj}W = {}^{L(old)}_{qj}W - \frac{1}{M_q}(x_{ij} - {}^{old}_{qj}Z); \quad (17)$$

$$^{new}_{qj}Z = \frac{{}^{U(new)}_{qj}w + {}^{L(new)}_{qj}w}{2}; \quad \text{for } j = 1, 2…., n \quad (18)$$

$M_q = M_q - 1$ (19)

7. **7.** Set *i=i+1*, repeat Step 4-Step 7 until all the patterns have been compared with the existing clusters, if the clustering process has converged, end; o/w, return to Step 3.

EXPERIMENTAL RESULT

A. Wine Dataset

The wine dataset consists of 3 clusters with 48, 59 and 71 points in each. Each sample has 13 features. A wide range of distance parameters was explored and only the range of (1.2, 1.5) was found to contain the most optimal DP. DP<0.9 implied that we got a huge number of clusters (around 20) and DP>1.5 meant that we received majority points (158 points with DP=1) clustered in one. However, none of the single results is in accordance with the accuracy claimed by Wang.

RESULTS:

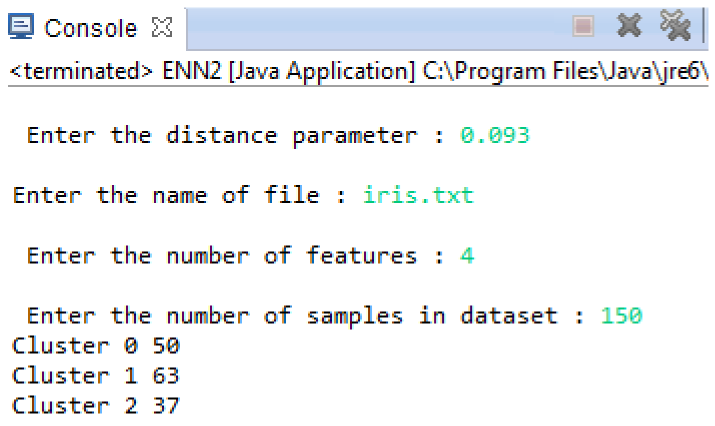
	NUMBER OF POINTS IN EACH CLUSTER										
DP	Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6	Cluster 7	Cluster 8	Cluster 9	Cluster 10
.97	47	12	92	1	6	18	1	1			
.99	47	12	18	1	60	6	32	1	1		
.9	31	12	34	18	12	1	6	12	1	1	51
.95	36	12	84	23	1	6	14	1	1		
1.1	27	8	37	72	1	11	19	1	3		
1.2	60	8	44	11	36	18	1	1			
1.3	87	8	11	26	1	46	1				
1.4	87	8	11	26	2	46					
1.5	108	19	1	50							
1.6	158	19	1								

ACCURACY: With DP=1.4 , miss rato = 36/178. Hence **80%** Accuracy

B. *Iris Dataset*

This is perhaps the best known database to be found in the pattern recognition literature. Fisher's paper is a classic in the field and is referenced frequently to this day. The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are NOT linearly separable from each other.

RESULTS SCREENSHOT:



```
<terminated> ENN2 [Java Application] C:\Program Files\Java\jre6\

 Enter the distance parameter : 0.093

Enter the name of file : iris.txt

 Enter the number of features : 4

 Enter the number of samples in dataset : 150
Cluster 0 50
Cluster 1 63
Cluster 2 37
```

ACCURACY = Miss ratio = 13/150 Hence **91.4%** accuracy.

LIMITATIONS

1. The number of clusters obtained and the elements contained in it depends on the choice of the distance parameter DP in this algorithm which is totally random in ENN-2.
2. While testing we were unable to get the results with high accuracy similar to the ones proposed by Wang. Choice of DP used by Wang is unknown to us thereby resulting in varied results.
3. The basis on which the accuracy of 96.6% in case of wine dataset is claimed is unknown and hence our performance can't be measured.
4. The accuracy claims in ENN-2 paper are made after making 10 random trials taking random DPs and then averaging the outputs and not for a single trial.

PHASE-II

ENN-2 WITH FUNCTION OPTIMIZATION USING PSO

In order to describe the PSO algorithm for function optimization we need some notation. Let f be a given objective function over a D dimensional problem space. The location of a particle i € $\{1, . . ., m\}$ is represented by a vector $xi = (x_{i1}, . . . , x_{iD})$ and the velocity of the particle by the vector $vi = (v_{i1}, . . . , v_{iD})$. Let l_d and u_d be lower and upper bounds for the particles coordinates in the dth dimension, d € $[1 : D]$. The best previous position of a particle is recorded as $pi = (p_{i1}, . . . p_{iD})$ and is called *pBest*. The index of the particle with the so far best found position in the swarm is denoted by g and pg is called *gBest*. At each iteration of a PSO algorithm after the evaluation of function f the personal best position of each particle i is updated, i.e. if $f(x_i) < f(p_i)$ then set $pi = xi$. If $f(p_i) < f(p_g)$ then i becomes the new global best solution, i.e. set $g = i$. Then the new velocity of each particle i is determined during the update of velocity in every dimension d € $[1 : D]$ as follows:
$v_{id} = w * v_{id} + c1 * r1 * (p_{id} - x_{id}) + c2 * r2 * (p_{gd} - x_{id})$ where;

- parameter w is called the *inertia weight*, it determines the influence of the old velocity; the higher the value of w the more the individuals tend to search in new areas; typical values for W are near 1.0;
- $c1$ and $c2$ are the *acceleration coefficients*, which are also called the cognitive and the social parameter respectively, because they are used to determine the influence of the local best position and the global best position respectively; typical values are $c1 = c2 = 2$;
- $r1$ and $r2$ are random values uniformly drawn from [0, 1].

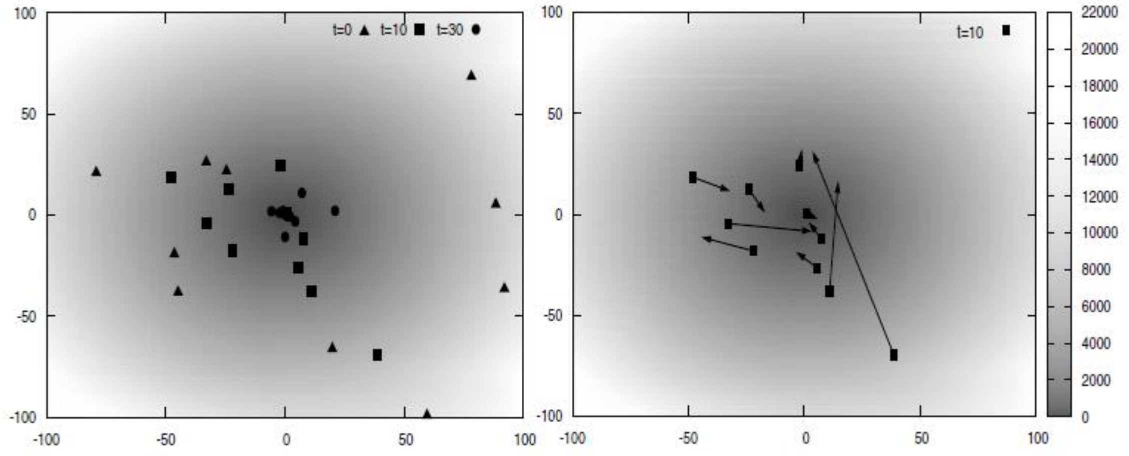


Figure 2- Illustration of Particle Swarm Optimization for a 2-D Sphere function

ALGORITHM:

M_k: Number of patterns belonging to cluster k
N_p: Total number of the input pattern
n: Number of features
k: Number of existing clusters
X_i: ith pattern
x_{ij}: jth feature of the ith input pattern
Z_k: Centre of the cluster k
λ: Distance parameter
w: Inertia weight
c1,c2: regression constants
a,b: social parameters

1. Set the desired DP λ.
2. Produce the first pattern and M_1=1. The centre coordinates and weights of the first cluster are calculated as

k=1 (1)
$Z_k = X_k \Rightarrow \{z_{k1}, z_{k2}, \ldots\ldots, z_{kn}\} = \{x_{k1}, x_{k2}, \ldots\ldots, x_{kn}\}$ (2)
${}^{L}_{kj}w = z_{kj} - \lambda$; for j=1,2…n (3)
${}_{kj}^{U}w = z_{kj} + \lambda$; for j = 1,2…n (4)

3. Read the input pattern vectors (put i=1), and go to next step.
4. Read next input pattern Xi= $\{x_{i1}, x_{i2} \ldots\ldots x_{in}\}$ and calculate the Extension distance ED_p between X_i and the existing pth cluster center as:

$$ED_p = \sum_{j=1}^{n}\left[\frac{|x_{ij} - z_{pj}| - ({}^{U}_{pj}w - {}^{L}_{pj}w)/2}{\left|\frac{{}^{U}_{pj}w - {}^{L}_{pj}w}{2}\right|} + 1\right]$$

; for p = 1,2…k; (5)

5. Find *s* for which,

   $ED_s = \min\{ED_p\}$. (6)

   If $ED_s > n$, then create a new cluster center. $ED_s > n$ expresses that X_i does not belong to sth cluster. Then a new cluster center will be created.

k = k+1 (7)
$M_k = 1$ (8)
$Z_k = X_i \Rightarrow \{z_{k1}, z_{k2}\ldots, z_{kn}\} = \{x_{i1}, x_{i2}\ldots, x_{in}\}$ (9)

${}^{L}_{kj}w = z_{kj} - \lambda$; for j=1, 2…n (10)

$^{U}_{kj}w = z_{kj} + \lambda$; for j = 1, 2….n (11)

Else, the pattern X_i belongs to the cluster *s*, and updates the weight and centre of clusters. Update them accordingly for n+1 number of points. **Create a Cost function for PSO** which will be optimized by our PSO java code giving us first the updated Lower Weight and then the updated Upper Weight.

COST FUNCTION : For a cluster C with 'n' number of elements, the Cost Function will be summation of extension distances of all points in that cluster with cluster center.

RETURN : We will get updated lower and upper weight from PSO Optimization.

$^{U(new)}_{sj}w$ and $^{L(new)}_{sj}w$

$$^{new}_{sj}z = \frac{^{U(new)}_{qj}w + ^{L(new)}_{qj}w}{2}; \quad \text{for } j = 1, 2…., n \qquad (12)$$

$M_s = M_s + 1$; for all $p \neq S$; (13)

6. If input pattern X_i changes from the cluster "q" (old one) to "k" (new one), then the weights and center of cluster "q" (for n-1 points since the point has now jumped to the new cluster) are modified as, and "k"(for n+1 points) modified in the same way as above;

$$^{new}_{qj}z = \frac{^{U(new)}_{qj}w + ^{L(new)}_{qj}w}{2}; \quad \text{for } j = 1, 2…., n \qquad (14)$$

$M_q = M_q - 1$ (15)

7. Set *i=i+1*, repeat Step 4-Step 7 until all the patterns have been compared with the existing clusters, if the clustering process has converged, end; o/w, return to Step 3.

IMPLEMENTATION

Classes:

1. Eel706: The main class within which is defined the main function implementing the algorithm. (calls neural_learn function).
2. Cluster: This is used to define each element of the cluster with its centre, lower and upper weights and the number of elements in the cluster.
3. Data: This defines each data element with its attributes and the cluster number to which it belongs after convergence
4. Fitnessfunction: Makes the cost function for PSO
5. Position: Defines the position of the particle in PSO
6. Velocity: Defines the velocity of the particle in PSO
7. Particle: Defines the Particle in PSO
8. Swarm: Defines the collection of particles(swarm) and methods which can be applied on it.
9. Init: Main class where PSO optimization algorithm is implemented.

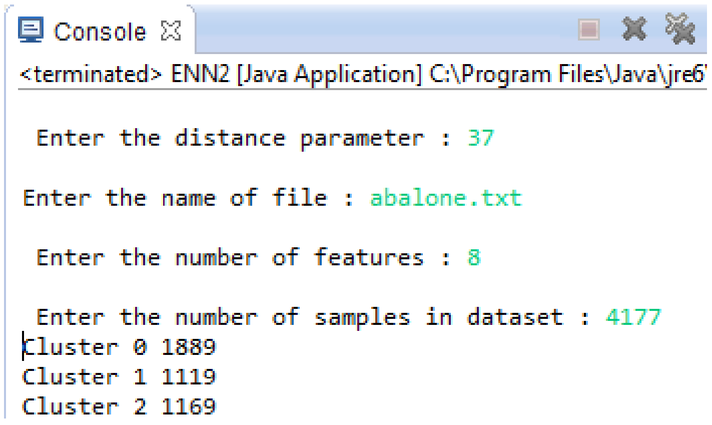
Code: See Last Page.

EXPERIMENTAL RESULT:

A. Abalone Dataset

Predicting the age of abalone from physical measurements. The age of abalone is determined by cutting the shell through the cone, staining it, and counting the number of rings through a microscope -- a boring and time-consuming task. Other measurements, which are easier to obtain, are used to predict the age. Further information, such as weather patterns and location (hence food availability) may be required to solve the problem.
No. of Data values = 4177
No. of Features = 8
Ideal Clustering = 3 cluster with 1307,1342 and 1528

RESULTS:



```
<terminated> ENN2 [Java Application] C:\Program Files\Java\jre6

 Enter the distance parameter : 37

Enter the name of file : abalone.txt

 Enter the number of features : 8

 Enter the number of samples in dataset : 4177
Cluster 0 1889
Cluster 1 1119
Cluster 2 1169
```

ACCURACY: **91.4%**

ADVANTAGES

1. PSO method can be applied to large datasets.
2. It does not require an initial guess of the cluster centre coordinates, nor of the initial number of clusters.
3. This ENN has lower memory consumption and shorter learning times than traditional neural networks in applications.

CONCLUSIONS

1. We observed that Phase 1 (i.e implementing extension neural network with normal updation) works best only for small dataset while Phase 2 (i.e implementing extension neural network with PSO optimization) works good for large datasets.
2. We still don’t achieve ideal clustering possibly due to no optimization range for value of distance parameter. The optimized range for distance parameter is not mentioned in Wangs paper also(given in reference).

REFERENCES

- *M. H. Wang and C. P. Hung, “Extension neural network and its applications”, Neural Networks, vol. 16, pp. 779–784, 2003.*
- *M. H. Wang, Extension Neural Network-Type 2 and Its Applications, IEEE Transactions on Neural Networks, Vol.16, No.6, Nov.2005.*
- *Neuro-Fuzzy Soft Computing- Jang, Sun and Mizutani*
- *Daniel Merkle and Martin Middendorf, "Swarm Intelligence", Department of Computer Science, University of Leipzig, Germany.*

CODE (phase 2): Printing out some important functions implementation in JAVA. The rest code is attached in zip file.

#NEURAL LEARN- main algorithm of Phase 2 learning implemented

```
public static void neural_learn(data[] sample,double DP,int param,int samples)
  {

     int ecc=1;
     double parameter[]= new double[param];
                    for(int r=0;r<=param-1;r++)
                    {
                              parameter[r]=DP;
                    }
     cluster clusters[]=new cluster[50];
                    for(int cl=0;cl<50;cl++)
                    {
                              clusters[cl]=new cluster(param);
     }
     for(int h=0;h<=param-1;h++)
     {
        clusters[0].centre[h]=sample[0].feature[h];
                              clusters[0].uweight[h]=clusters[0].centre[h]+parameter[h];
        clusters[0].lweight[h]=clusters[0].centre[h]-parameter[h];
     }
     clusters[0].pattern=1;
     sample[0].clusterid=0;
     for(int iter=0;iter<=5;iter++)    //assuming 1 iterations
     {
        System.out.println("ITERATION NUMBER"+iter);
     for(int i=1;i<sample.length;i++)   //Read next input pattern
     {
        double
extension_distance[]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0};
                              for(int j=1;j<=ecc;j++)        //for p=1,2,...pth cluster
        {
           for(int k=0;k<=param-1;k++)       //number of feature..for summation over j
              extension_distance[j-1]+=((Math.abs(sample[i].feature[k]-clusters[j-1].centre[k])-
(Math.abs(clusters[j-1].uweight[k]-clusters[j-1].lweight[k])/2))/Math.abs((clusters[j-1].uweight[k]-
clusters[j-1].lweight[k])/2))+1;

           // System.out.print("--p"+extension_distance[j-1]);
                 }
           //       System.out.println();


        int index=min(extension_distance,ecc);
```

```
          // System.out.println("this is it--"+extension_distance[index]+"--");
           if(extension_distance[index]>param)        //remember to check this out...EDs>n or EDs>DP
           {
              //System.out.println("joker");
              ecc++;
                                  clusters[ecc-1].pattern=1;
                                  for(int h=0;h<=param-1;h++)
              {
                 clusters[ecc-1].centre[h]=sample[i].feature[h];
                                                      clusters[ecc-1].uweight[h]=clusters[ecc-
1].centre[h]+parameter[h];
                 clusters[ecc-1].lweight[h]=clusters[ecc-1].centre[h]-parameter[h];
              }
              sample[i].clusterid=ecc-1;
                                }
                                else
           {
              int prev_clusterid=sample[i].clusterid;
              if(sample[i].clusterid==-1 || sample[i].clusterid!=index)
                       {
                  //System.out.println("entered");
                 clusters[index].pattern+=1;
                 sample[i].clusterid=index;


                 init pso= new init();
                 clusters[index].lweight=pso.execute(clusters[index], index,sample,0);
                 pso= new init();
                 clusters[index].uweight=pso.execute(clusters[index], index,sample,1);


                         for(int h=0;h<=param-1;h++)
                 {

                    clusters[index].centre[h]=(clusters[index].uweight[h]+clusters[index].lweight[h])/2;

                 }
                 sample[i].clusterid=prev_clusterid;
              }
               if(sample[i].clusterid!=-1 && sample[i].clusterid!=index)
              {
                 clusters[sample[i].clusterid].pattern-=1;
                 sample[i].clusterid=index;
                 init pso= new init();


                 clusters[prev_clusterid].lweight=pso.execute(clusters[prev_clusterid],
prev_clusterid,sample,0);
                 pso= new init();
                 clusters[prev_clusterid].uweight=pso.execute(clusters[prev_clusterid],
prev_clusterid,sample,1);
```

```
                for(int h=0;h<=param-1;h++)
                {
clusters[prev_clusterid].centre[h]=(clusters[prev_clusterid].uweight[h]+clusters[prev_clusterid].lweight[h]
)/2;
                }

            }
            sample[i].clusterid=index;
            }
                                }
        }
        for(int y=0;y<ecc;y++)
        {
                                System.out.println("Cluster " + y +" "+ clusters[y].pattern);
        }
                    for(int z=0;z<samples;z++)
        {
          System.out.println("Point "+ (z+1)+" "+sample[z].clusterid);
        }
            }
```

#SWARM- class of a collection of particles in PSO

```
public  class swarm {

  // public static ArrayList swarm_particles;
  public static double[] pBest;
   public static double fitnessList[];
   public static Position[] pBestLoc;
   public static double gBest;
   public static Position gBestLoc;
   public static Particle[] swarmparticles;
   public static int counter;

   swarm(int size,int dimension)
   {
     //swarm_particles = new ArrayList(size);
      swarmparticles = new Particle[size];
      pBest = new double[size];
      fitnessList = new double[size];
      pBestLoc =  new Position[size];
      gBest=0;
      counter=0;
   }

   public static void add(Particle p)
   {

      swarmparticles[counter]=p;
```

```
        counter++;

    }

    public static Particle get(int i)
    {

        return swarmparticles[i];
    }
    public void set(Particle p,int i)
    {
        swarmparticles[i]=p;
    }
    public static void calculateAllFitness()
    {
        int i=10;
        //int i= swarm_particles.size();
        //System.out.println(i);
        for(int j=0;j<i;j++)
        {


            swarmparticles[j].calculateFitness();

            fitnessList[j]=swarmparticles[j].fitness;

        }
    }
    public static int getBestParticle()
    {
        int best_index=0;
        double min=fitnessList[0];
        for(int i=1;i<fitnessList.length;i++)
        {
            if(fitnessList[i]<min)
            {
                min=fitnessList[i];
                best_index=i;
            }
        }
        return best_index;
    }

}
```

#init- main class implementing the PSO optimization for a given Cost Function
----

```
public class init {

int SWARM_SIZE = 10;
   int DIMENSION = 8;
   int MAX_ITERATION = 10;
   double C1 = 2.0;
   double C2 = 2.0;
   double W_UP = 1.0;
   double W_LO = 0.0;

   swarm swarm1 = new swarm(SWARM_SIZE,DIMENSION);

private void initializeSwarm(cluster C,int clusterid, data[] input,int switc) {




 for (int i = 0; i < SWARM_SIZE; i++) {
Particle p = new Particle(DIMENSION);
 p.C=C;
 p.clusterid = clusterid;
 p.input=input;
 p.switc=switc;
 Random generator = new Random();

 double posX[]=new double[DIMENSION];

 for(int feature=0; feature<DIMENSION;feature++)
 {
    posX[feature]= generator.nextInt(5);

 }

 double velX[]=new double[DIMENSION];

 for(int feature=0; feature<DIMENSION;feature++)
 {
    velX[feature]= 0;


 }
 Position newpos = new Position(DIMENSION);
 Velocity newvel = new Velocity(DIMENSION);
 newpos.setX(posX);
```

```
newvel.setX(velX);
p.setLocation((newpos));
p.setVelocity((newvel));

 swarm1.add(p);


public double[] execute(cluster C,int clusterid, data[] input,int switc) {
Random generator = new Random();
initializeSwarm(C,clusterid,input,switc);

//evolutionaryStateEstimation();

int t = 0;
double w;

while (t < MAX_ITERATION) {
// calculate corresponding f(i,t) corresponding to location x(i,t)
swarm1.calculateAllFitness();


// update pBest
if (t == 0) {
for (int i = 0; i < SWARM_SIZE; i++) {
swarm1.pBest[i] = swarm1.fitnessList[i];
//System.out.println(swarm1.pBestLoc.length);
swarm1.pBestLoc[i]=swarm1.get(i).getLocation();
}
} else {
for (int i = 0; i < SWARM_SIZE; i++)
{

   if (swarm1.fitnessList[i] < swarm1.pBest[i])
   {
      swarm1.pBest[i] = swarm1.fitnessList[i];
      swarm1.pBestLoc[i]= swarm.get(i).getLocation();
   }
}
}

int bestIndex = swarm1.getBestParticle();
// update gBest
if (t == 0 || swarm1.fitnessList[bestIndex] < swarm1.gBest) {
swarm1.gBest = swarm1.fitnessList[bestIndex];
swarm1.gBestLoc = swarm1.get(bestIndex).getLocation();
}

w = W_UP - (((double) t) / MAX_ITERATION) * (W_UP - W_LO);

for (int i = 0; i < SWARM_SIZE; i++) {
// update particle Velocity
   Particle p= swarm1.get(i);
double r1 = generator.nextDouble();
```

```
 double r2 = generator.nextDouble();
 Position lx = swarm1.get(i).getLocation();
 Velocity vx = swarm1.get(i).getVelocity();
 Position[] pBestX = swarm1.pBestLoc;
 Position gBestXLoc = swarm1.gBestLoc;

 Velocity newVelX = new Velocity(DIMENSION);
 Position newPosX = new Position(DIMENSION);
 double[] newvel = new double[DIMENSION];
 for(int feature=0;feature<DIMENSION;feature++)
 {
   newvel[feature]=w * (lx.get_feature(feature)) + (r1 * C1) * (swarm1.pBestLoc[i].get_feature(feature) -
lx.get_feature(feature)) + (r2 * C2) * (gBestXLoc.get_feature(feature) - lx.get_feature(feature));
 }
 newVelX.setX(newvel);
 p.setVelocity(newVelX);

 // update particle Location
 double newpos[]= new double[DIMENSION];
 for(int feature=0;feature<DIMENSION;feature++)
 {
     newpos[feature]  = lx.get_feature(feature) + newvel[feature];
 }
 newPosX.setX(newpos);
 p.setLocation(newPosX);

 swarm1.set(p, i);




 }

 t++;


 }
 return swarm1.gBestLoc.getX();

 }

}
```