NEURAL NETWORKS

ASSIGNMENT

UNSUPERVISED LEARNING

Non-Parametric CLustering

Keya Desai

201501012

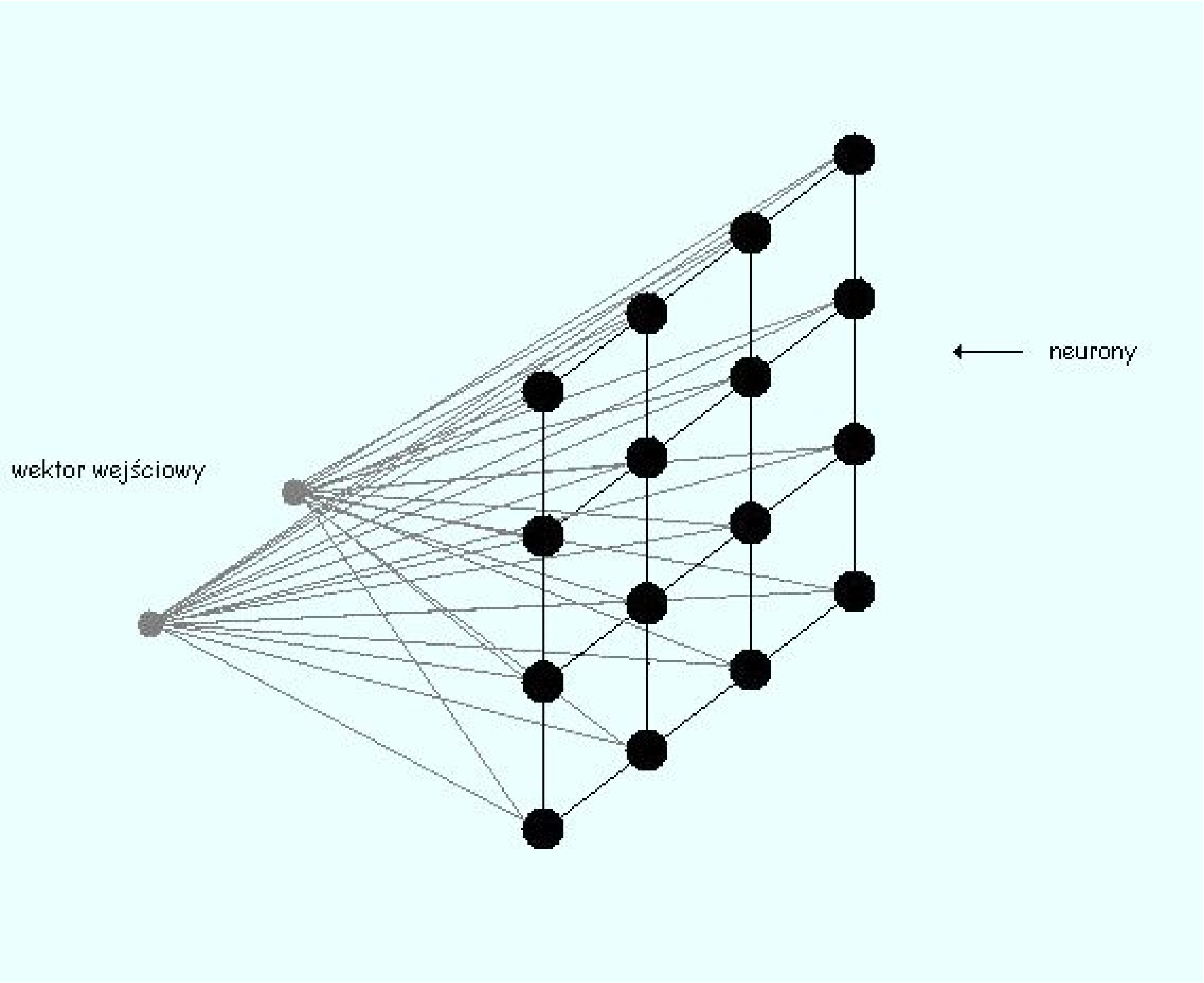
Self organizing Maps

Concept:

- SOM is an Unsupervised Neural Network learning.
- A self-organizing map consists of components called nodes or neurons. Associated with each node are a weight vector of the same dimension as the input data vectors, and a position in the map space.
- The neighboring cells in the neural network compete in their activities by means of mutual lateral interactions.

Functioning of self-organizing neural network is divided into three stages:

- construction
- learning
- identification



2-D map of neurons

Steps:

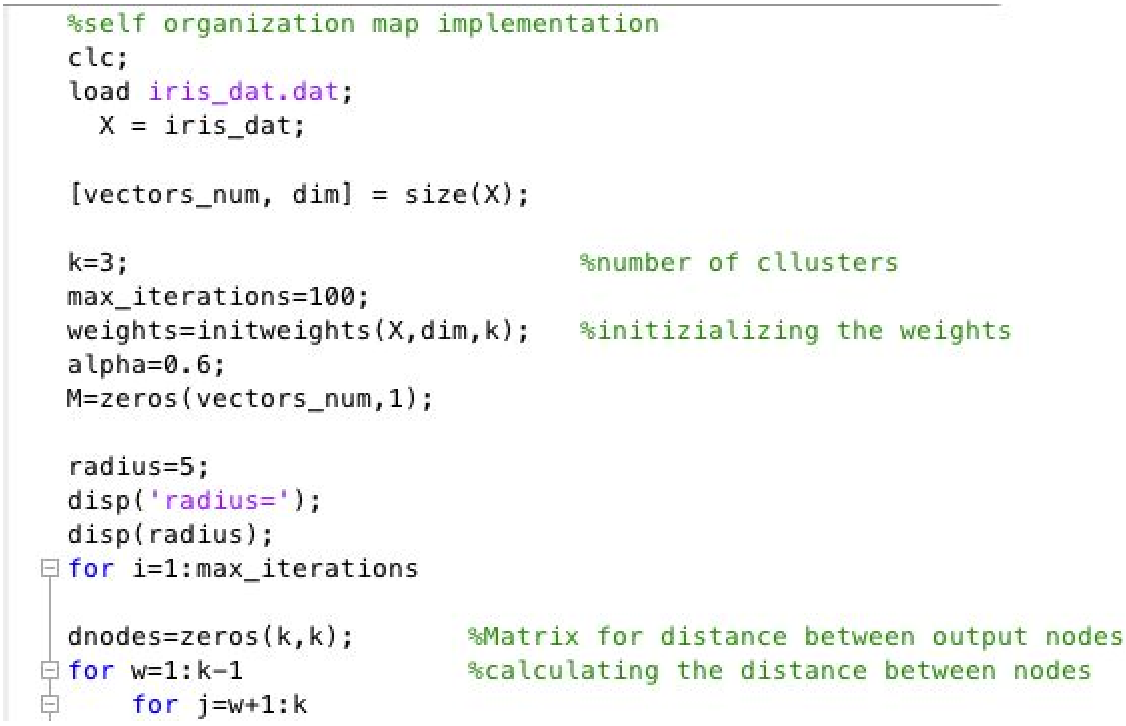
Training occurs in several steps and over many iterations:

1. Each node's weights are initialized.
2. A vector is chosen at random from the set of training data and presented to the lattice.
3. Every node is examined to calculate which one's weights are most like the input vector. The winning node is commonly known as the Best Matching Unit (BMU).
4. The radius of the neighbourhood of the BMU is now calculated. This is a value that starts large, typically set to the 'radius' of the lattice, but diminishes each time-step. Any nodes found within this radius are deemed to be inside the BMU's neighbourhood.
5. Each neighbouring node's (the nodes found in step 4) weights are adjusted to make them more like the input vector. The closer a node is to the BMU, the more its weights get altered.
6. Repeat step 2 for N iterations.

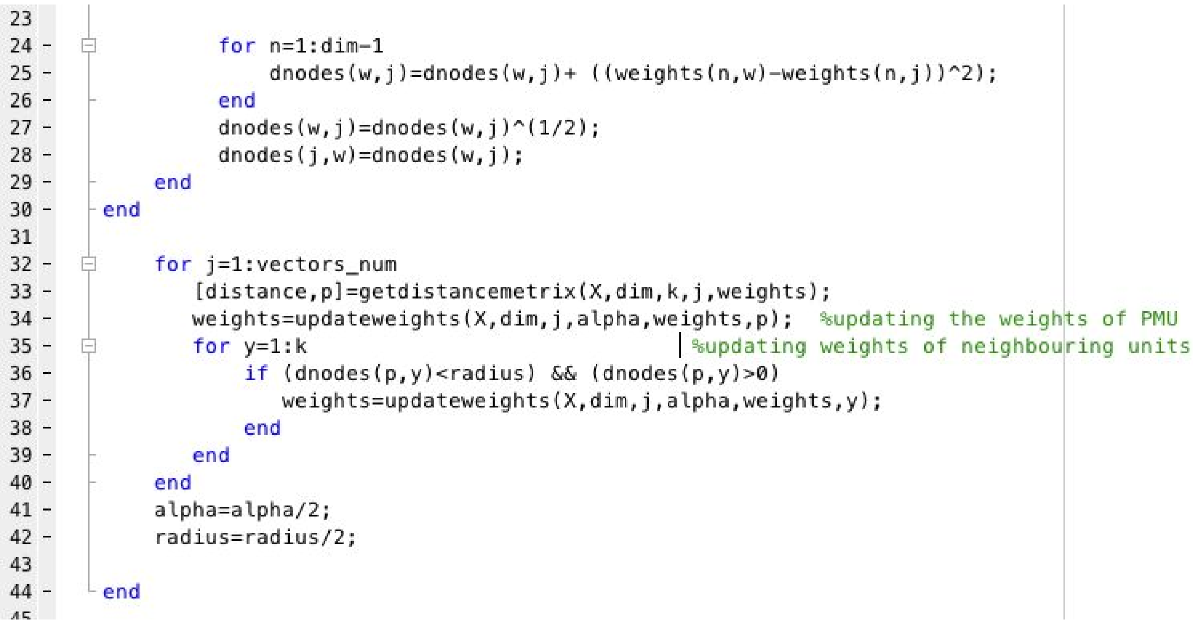
IRIS Dataset:

For Iris dataset, the number of neurons in output layer is 3 and in input layer(i.e number of features) is 4.

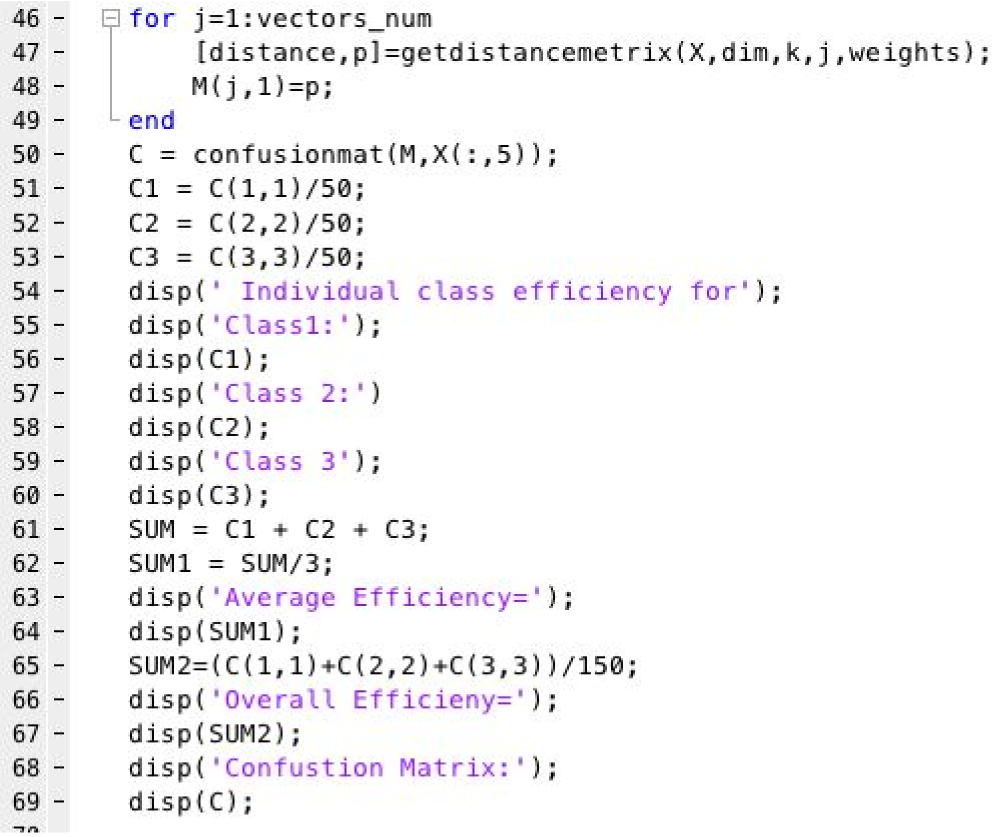
Code:



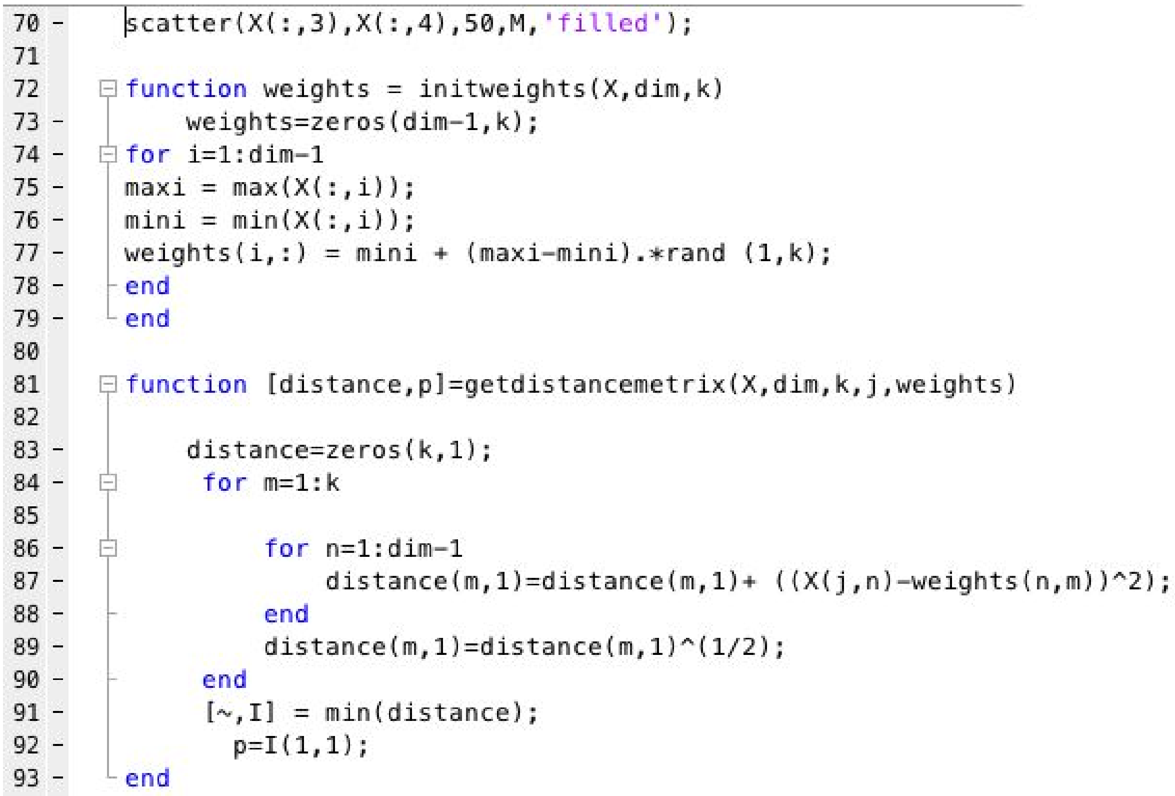
```
%self organization map implementation
clc;
load iris_dat.dat;
  X = iris_dat;

[vectors_num, dim] = size(X);

k=3;                               %number of cllusters
max_iterations=100;
weights=initweights(X,dim,k);      %initizializing the weights
alpha=0.6;
M=zeros(vectors_num,1);

radius=5;
disp('radius=');
disp(radius);
for i=1:max_iterations

dnodes=zeros(k,k);          %Matrix for distance between output nodes
for w=1:k-1                 %calculating the distance between nodes
    for j=w+1:k
```



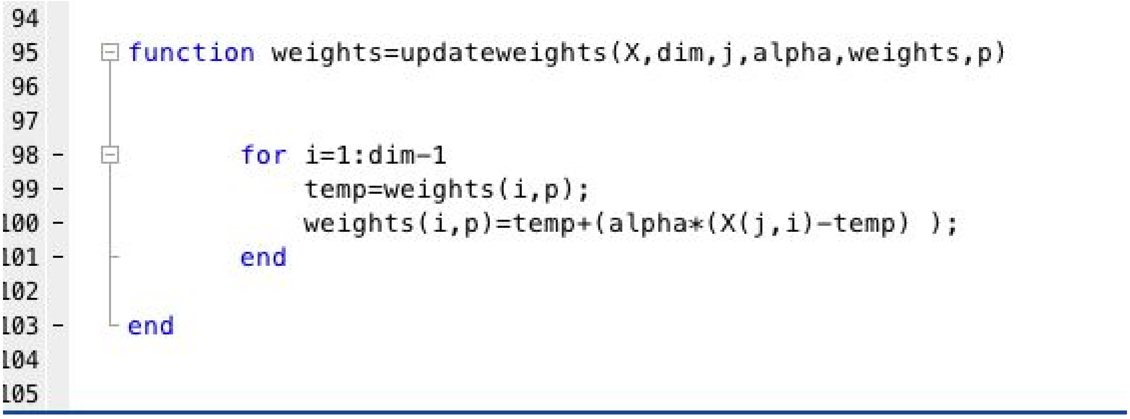
```
         for n=1:dim-1
             dnodes(w,j)=dnodes(w,j)+ ((weights(n,w)-weights(n,j))^2);
         end
         dnodes(w,j)=dnodes(w,j)^(1/2);
         dnodes(j,w)=dnodes(w,j);
    end
end

    for j=1:vectors_num
       [distance,p]=getdistancemetrix(X,dim,k,j,weights);
       weights=updateweights(X,dim,j,alpha,weights,p);  %updating the weights of PMU
       for y=1:k                              %updating weights of neighbouring units
           if (dnodes(p,y)<radius) && (dnodes(p,y)>0)
              weights=updateweights(X,dim,j,alpha,weights,y);
           end
       end
    end
    alpha=alpha/2;
    radius=radius/2;

end
```



```
for j=1:vectors_num
    [distance,p]=getdistancemetrix(X,dim,k,j,weights);
    M(j,1)=p;
end
C = confusionmat(M,X(:,5));
C1 = C(1,1)/50;
C2 = C(2,2)/50;
C3 = C(3,3)/50;
disp(' Individual class efficiency for');
disp('Class1:');
disp(C1);
disp('Class 2:')
disp(C2);
disp('Class 3');
disp(C3);
SUM = C1 + C2 + C3;
SUM1 = SUM/3;
disp('Average Efficiency=');
disp(SUM1);
SUM2=(C(1,1)+C(2,2)+C(3,3))/150;
disp('Overall Efficieny=');
disp(SUM2);
disp('Confustion Matrix:');
disp(C);
```



```
scatter(X(:,3),X(:,4),50,M,'filled');

function weights = initweights(X,dim,k)
    weights=zeros(dim-1,k);
for i=1:dim-1
maxi = max(X(:,i));
mini = min(X(:,i));
weights(i,:) = mini + (maxi-mini).*rand (1,k);
end
end

function [distance,p]=getdistancemetrix(X,dim,k,j,weights)

    distance=zeros(k,1);
     for m=1:k

         for n=1:dim-1
             distance(m,1)=distance(m,1)+ ((X(j,n)-weights(n,m))^2);
         end
         distance(m,1)=distance(m,1)^(1/2);
     end
     [~,I] = min(distance);
       p=I(1,1);
end
```



```

function weights=updateweights(X,dim,j,alpha,weights,p)


       for i=1:dim-1
           temp=weights(i,p);
           weights(i,p)=temp+(alpha*(X(j,i)-temp) );
       end

end


```

Output:

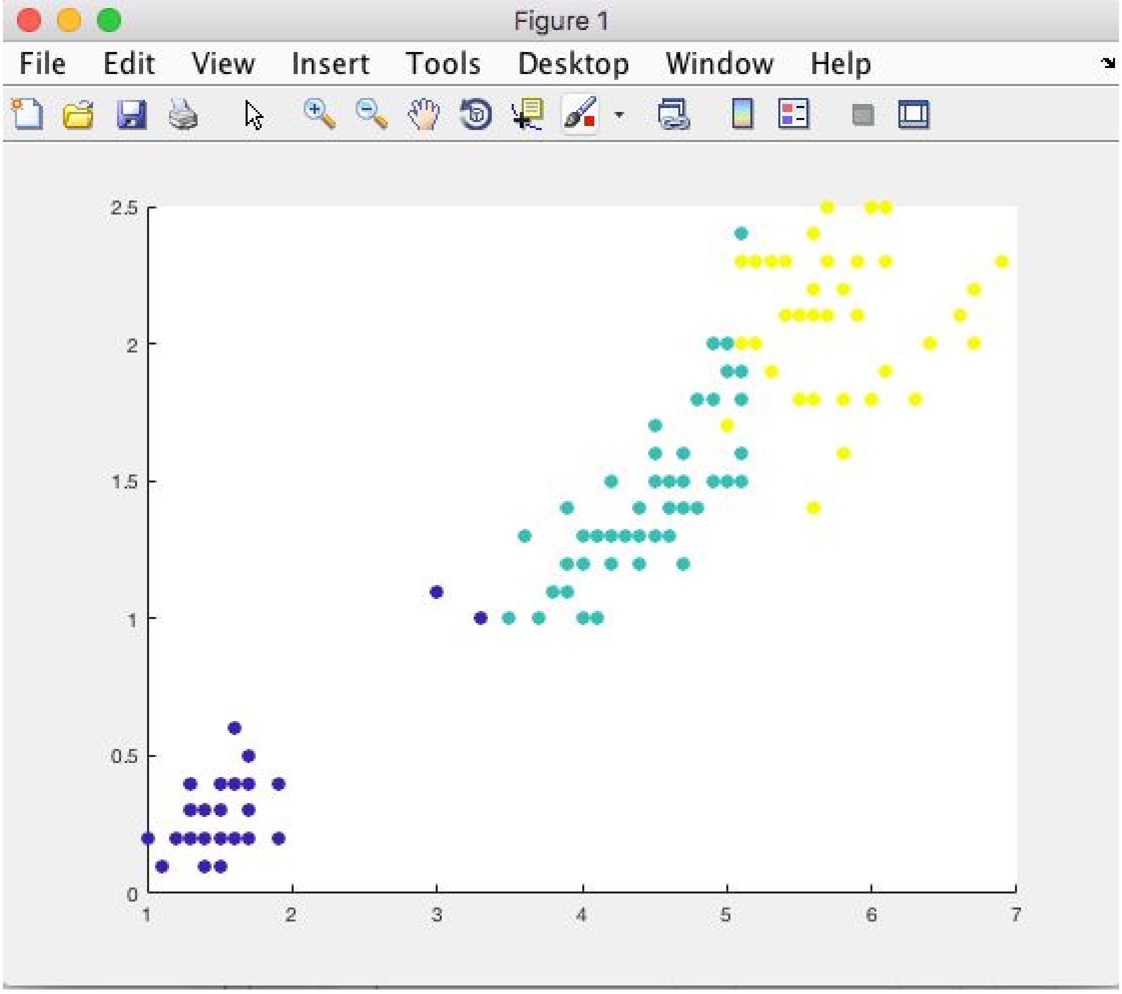


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7

Radius Analysis:

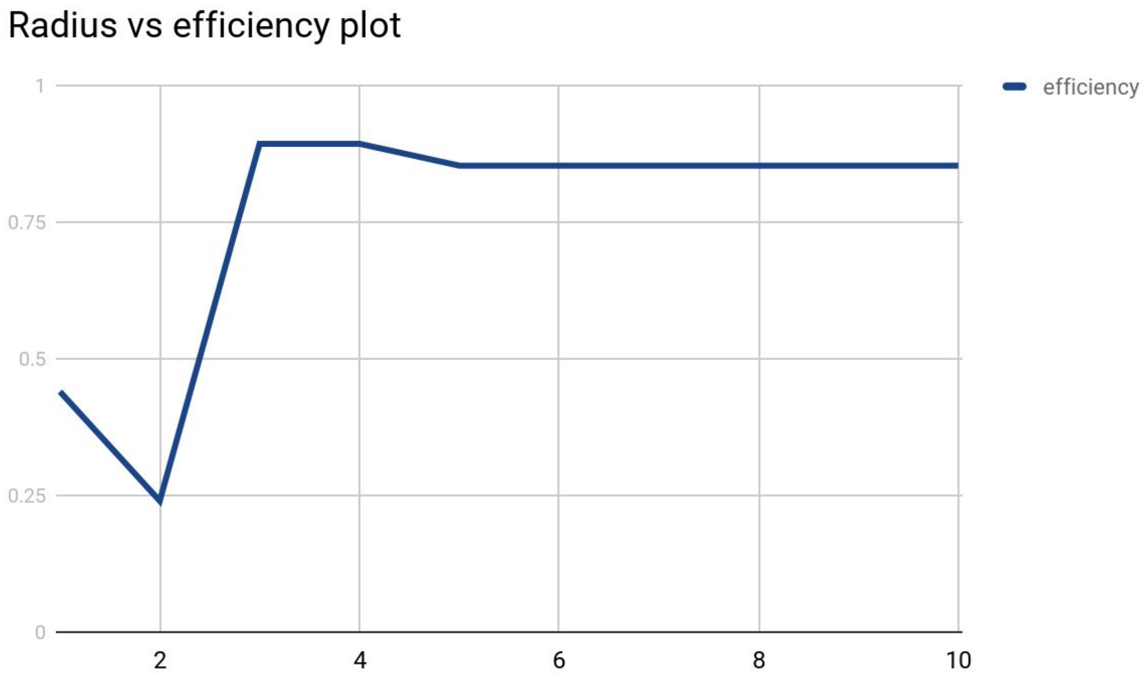
Concept

Determining the Best Matching Unit's Local Neighbourhood

- After for each iteration,the BMU(Best Matching Unit) has been determined, the next step is to calculate which of the other nodes are within the BMU's neighbourhood. All these nodes have their weight vectors altered in the next step.
- To get the neighbours, the distance between each neuron is calculated and if it below a specific threshold(given by variable radius) the weights of that neuron is also updated.

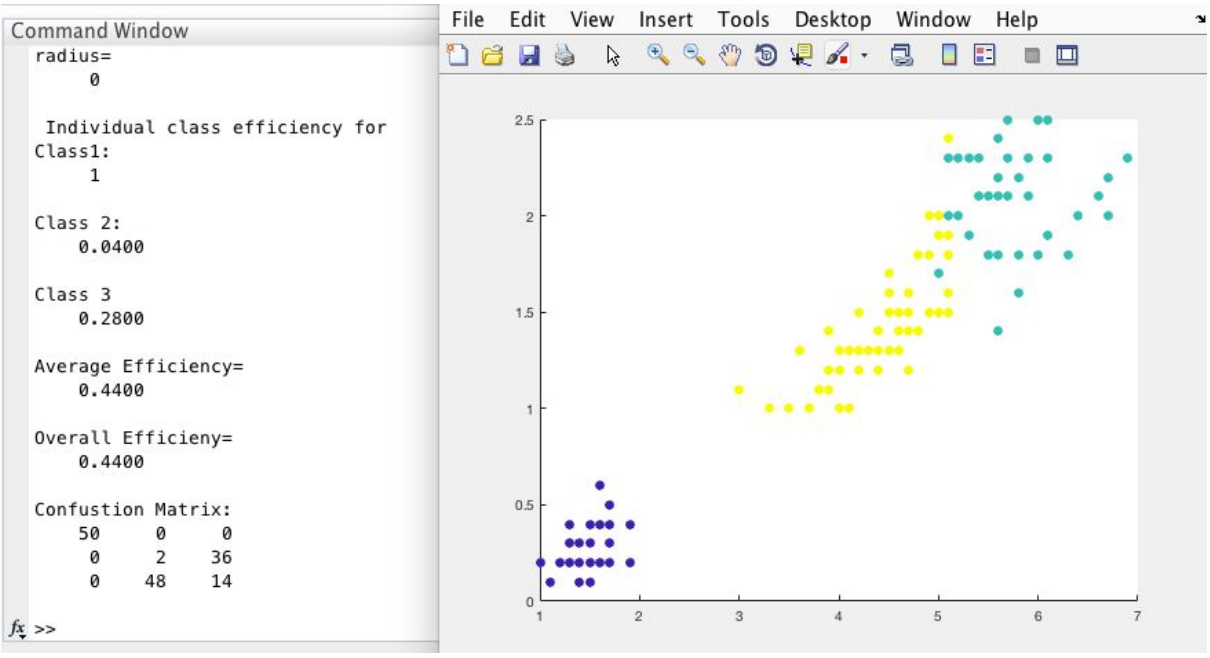
Observation

- For different values of radius,we calculated the average efficiency.We found that by initializing the radius with 3 and 4 ,the efficiency was maximum.As the efficiency was increased the efficiency decreased by 0.01 and remained constant thereafter.
- For radius=0 i.e not using the concept of neighbourhood neurons, we got the least efficiency.
- Also, by keeping radius to be a finite value, we got a consistent result on running the code multiple times which is not the case when radius is kept 0.

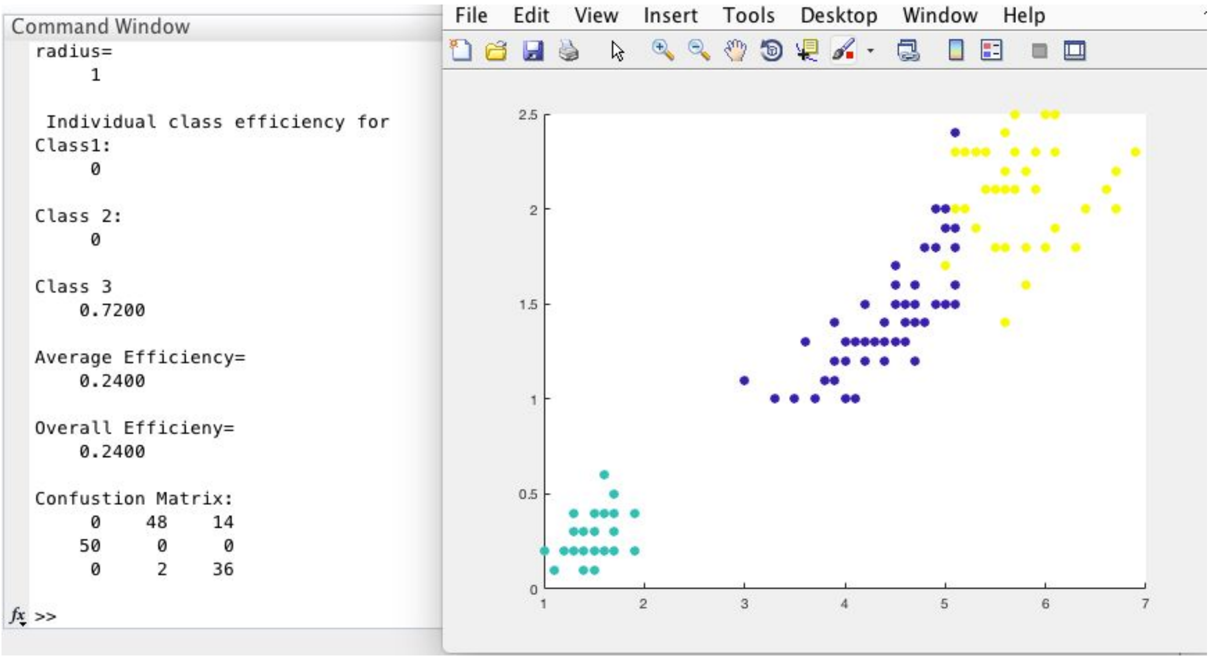


- The following output was taken by keeping the number of iterations constant at 10.

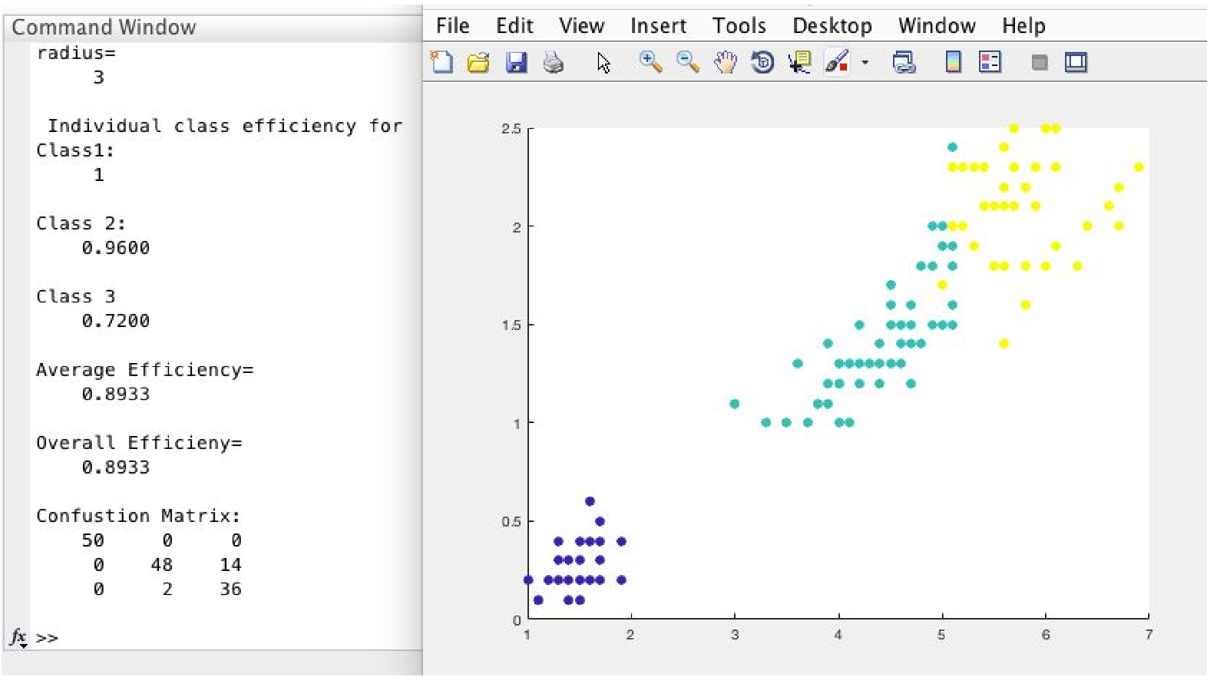
Radius	Average Efficiency
0	0.4400
1	0.2400
2	0.2400
3	0.8933
4	0.8933
5	0.8533
10	0.8533



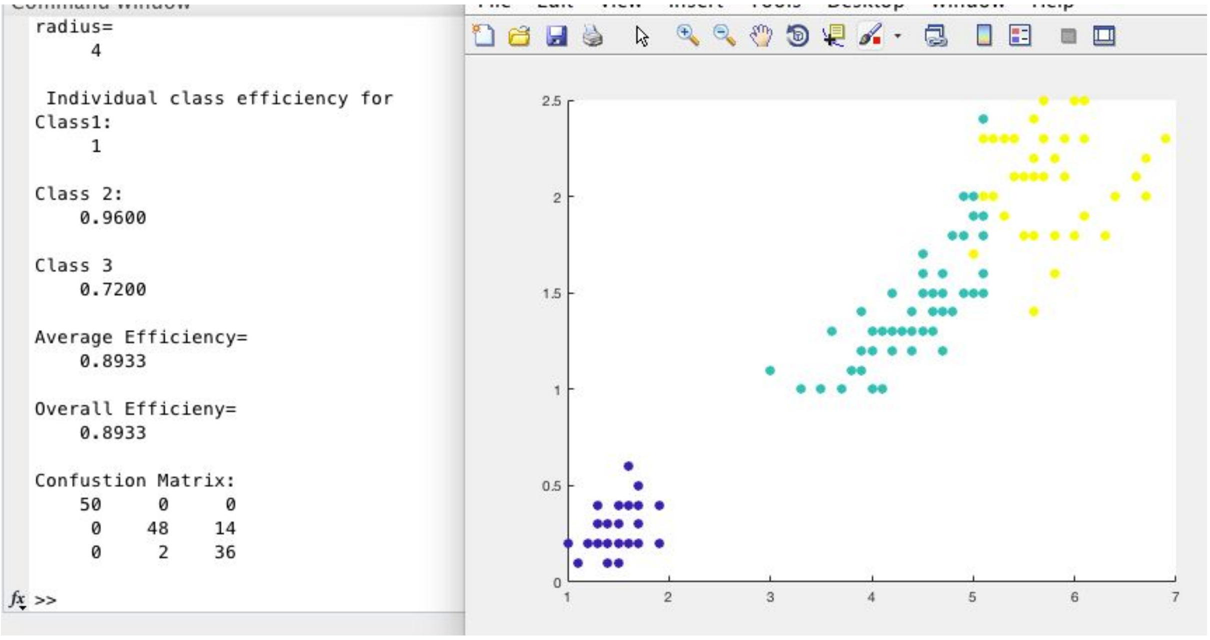
```
radius=
     0

 Individual class efficiency for
Class1:
     1

Class 2:
    0.0400

Class 3
    0.2800

Average Efficiency=
    0.4400

Overall Efficieny=
    0.4400

Confustion Matrix:
    50     0     0
     0     2    36
     0    48    14

fx >>
```



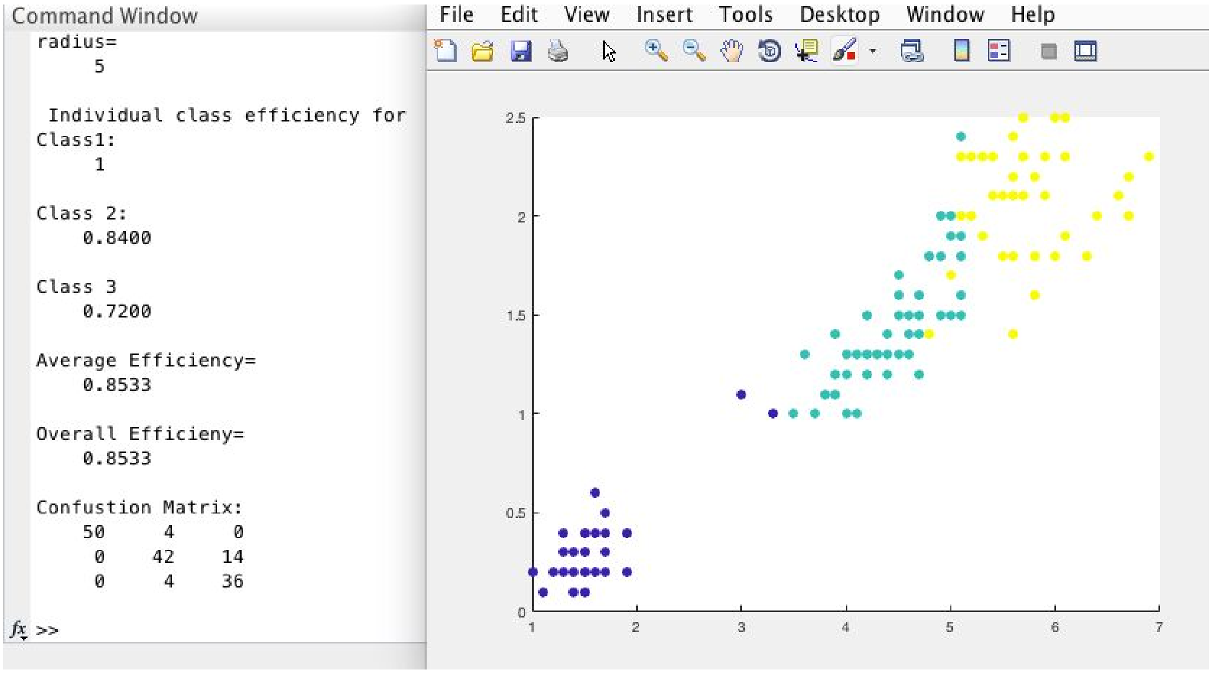
```
radius=
     1

 Individual class efficiency for
Class1:
     0

Class 2:
     0

Class 3
    0.7200

Average Efficiency=
    0.2400

Overall Efficieny=
    0.2400

Confustion Matrix:
     0    48    14
    50     0     0
     0     2    36

fx >>
```



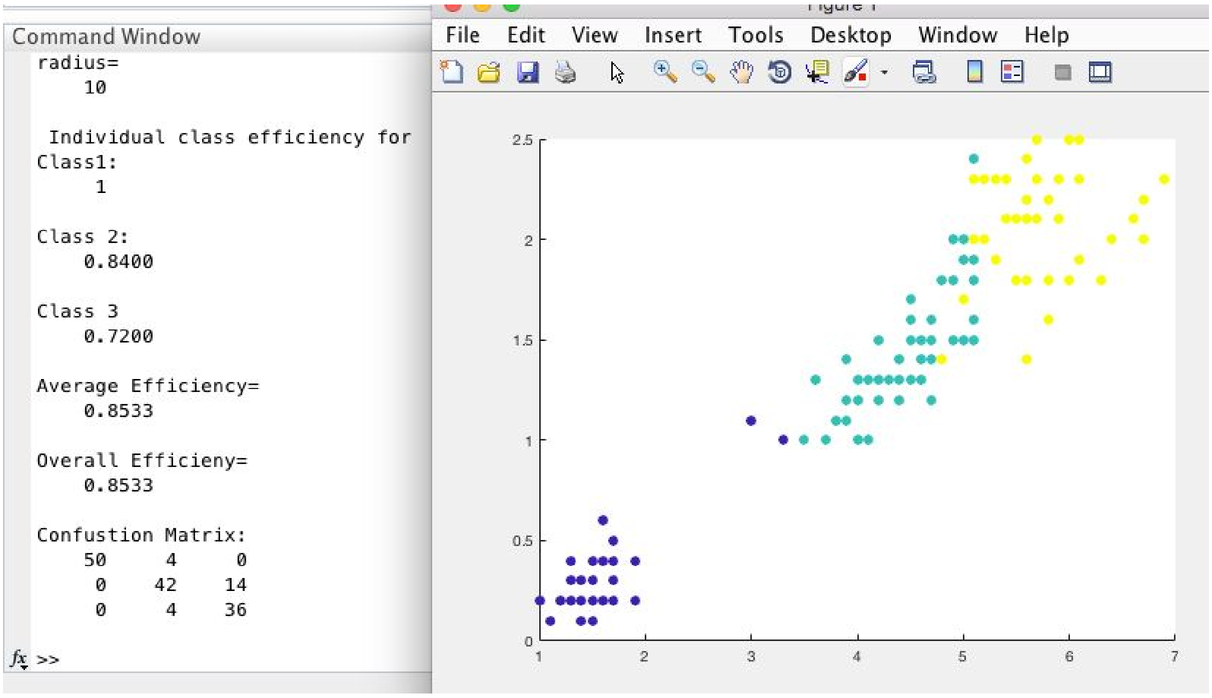
```
radius=
     3

 Individual class efficiency for
Class1:
     1

Class 2:
    0.9600

Class 3
    0.7200

Average Efficiency=
    0.8933

Overall Efficieny=
    0.8933

Confustion Matrix:
    50     0     0
     0    48    14
     0     2    36

>>
```



```
radius=
     4

 Individual class efficiency for
Class1:
     1

Class 2:
    0.9600

Class 3
    0.7200

Average Efficiency=
    0.8933

Overall Efficieny=
    0.8933

Confustion Matrix:
    50     0     0
     0    48    14
     0     2    36

>>
```



```
radius=
     5

 Individual class efficiency for
Class1:
     1

Class 2:
    0.8400

Class 3
    0.7200

Average Efficiency=
    0.8533

Overall Efficieny=
    0.8533

Confustion Matrix:
    50     4     0
     0    42    14
     0     4    36

fx >>
```



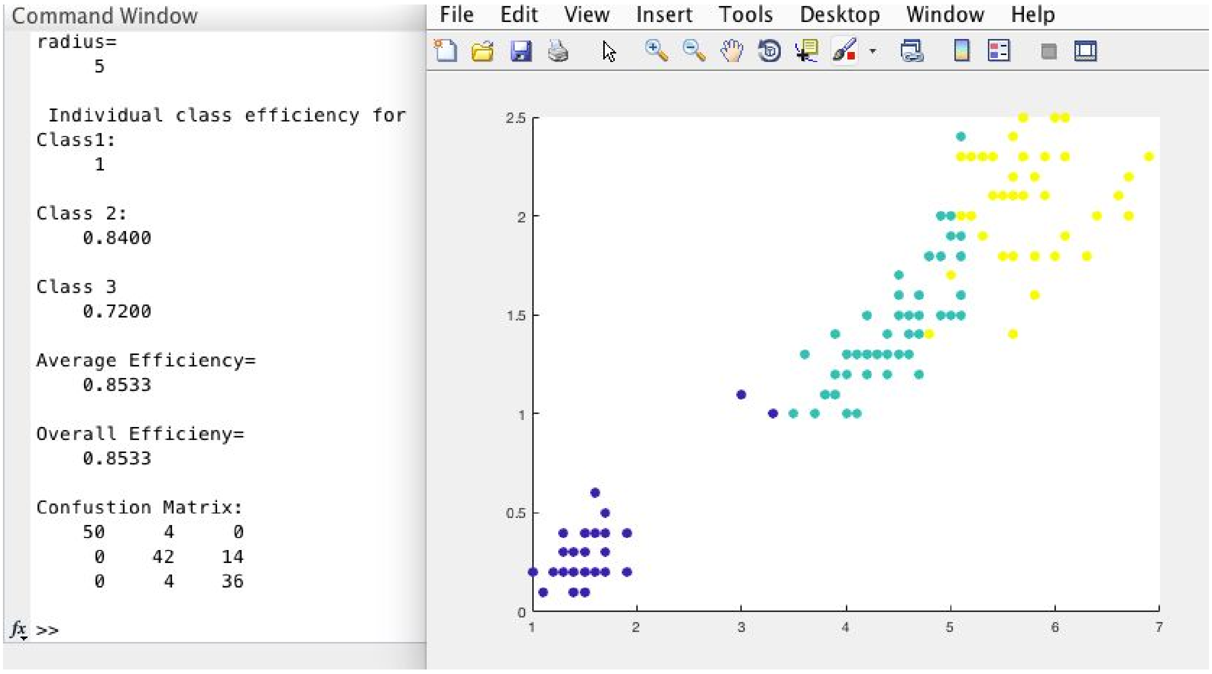
```
radius=
    10

 Individual class efficiency for
Class1:
     1

Class 2:
    0.8400

Class 3
    0.7200

Average Efficiency=
    0.8533

Overall Efficieny=
    0.8533

Confustion Matrix:
    50     4     0
     0    42    14
     0     4    36

fx >>
```

Number Of Iterations

Observation:

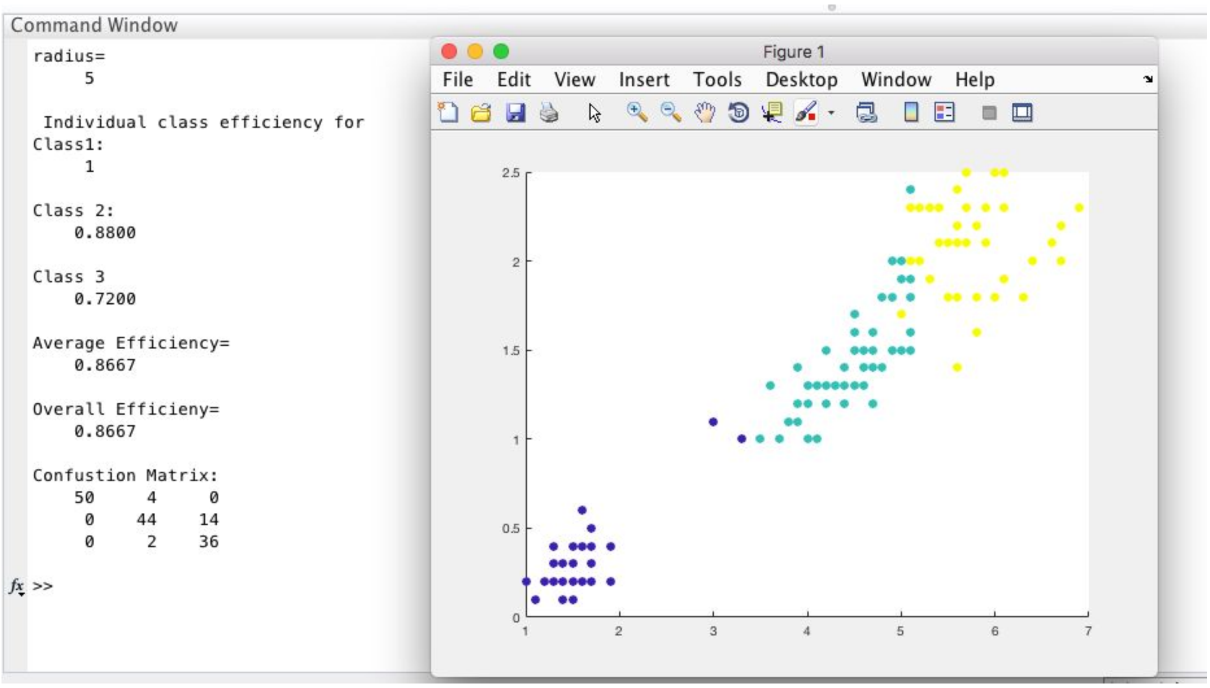
- We observed that by keeping the radius constant at 5 we get the maximum efficiency by iterating 100 times.On increasing the number of iterations further the efficiency remains the same.

Number of iterations=10



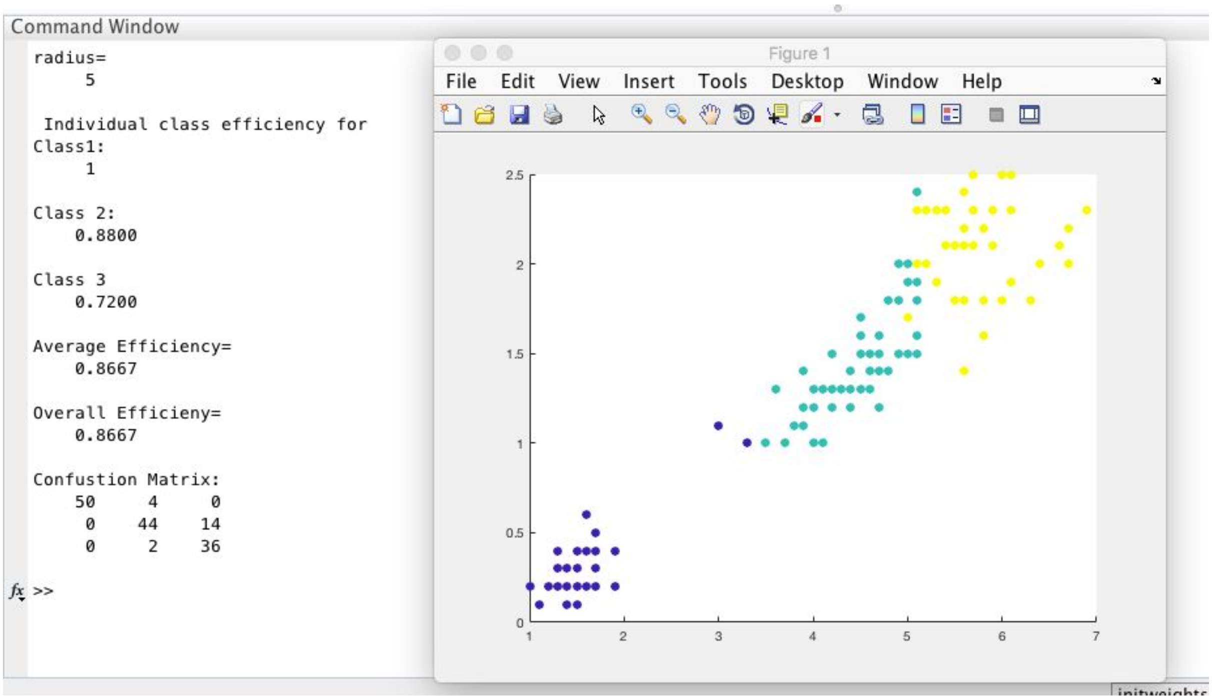
```
radius=
     5

 Individual class efficiency for
Class1:
     1

Class 2:
    0.8400

Class 3
    0.7200

Average Efficiency=
    0.8533

Overall Efficieny=
    0.8533

Confustion Matrix:
    50     4     0
     0    42    14
     0     4    36

>>
```

Number of iterations=100



```
radius=
     5

 Individual class efficiency for
Class1:
     1

Class 2:
    0.8800

Class 3
    0.7200

Average Efficiency=
    0.8667

Overall Efficieny=
    0.8667

Confustion Matrix:
    50     4     0
     0    44    14
     0     2    36

fx >>
```

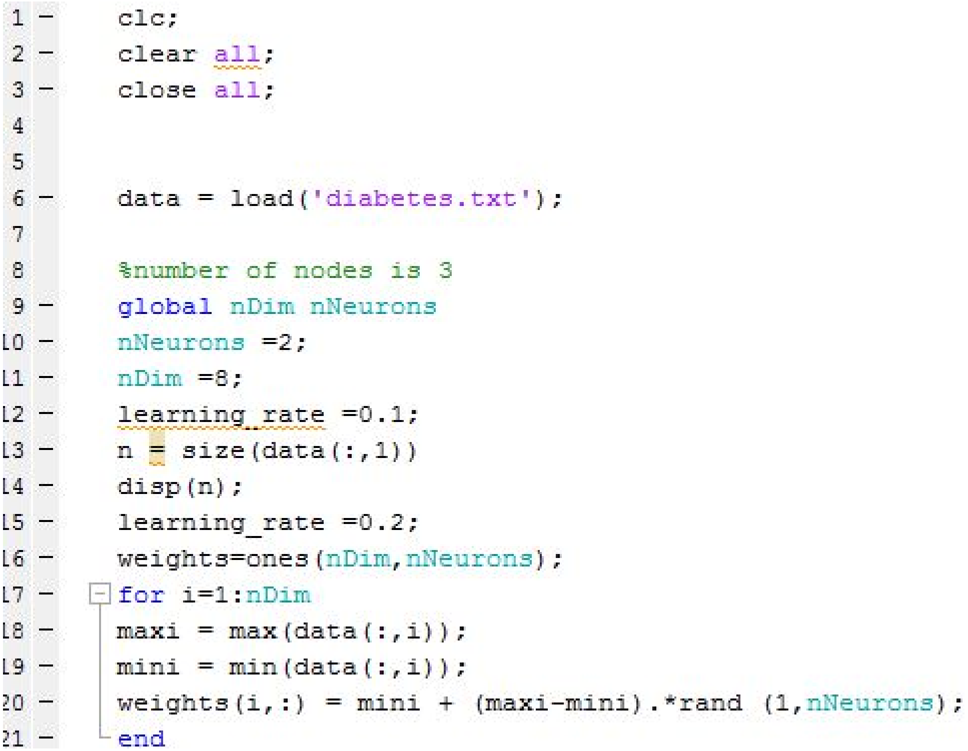
Number of iterations=500



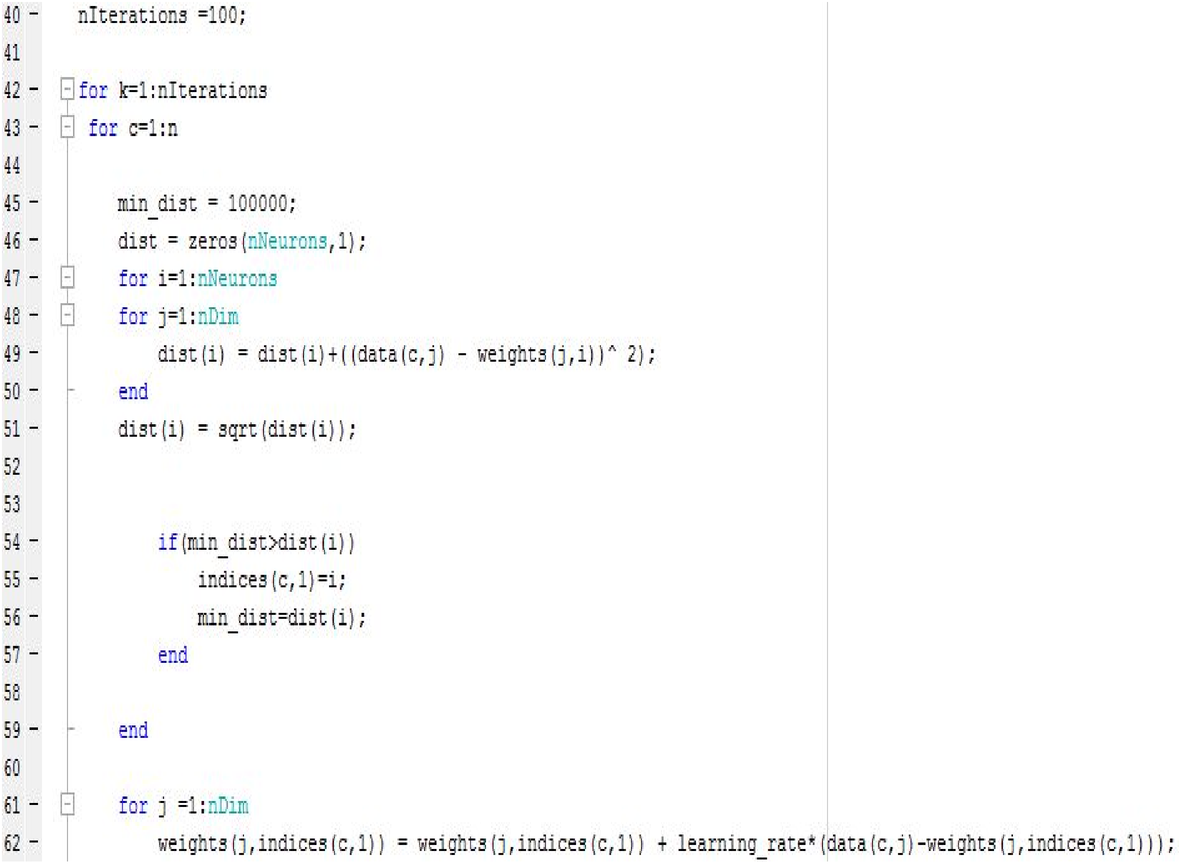
```
radius=
     5

 Individual class efficiency for
Class1:
     1

Class 2:
    0.8800

Class 3
    0.7200

Average Efficiency=
    0.8667

Overall Efficieny=
    0.8667

Confustion Matrix:
    50     4     0
     0    44    14
     0     2    36

fx >>
```

PIMA Diabetes data set

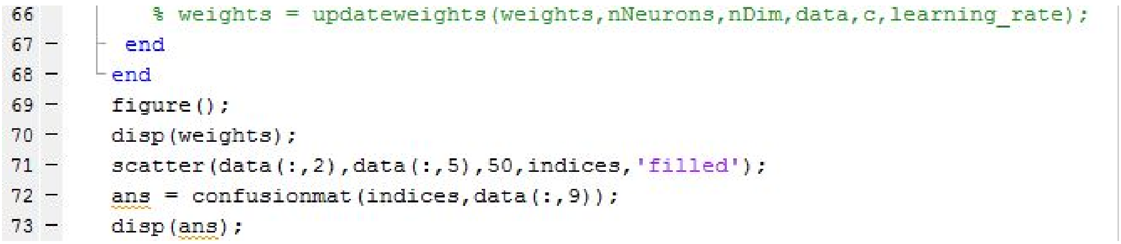
Code:



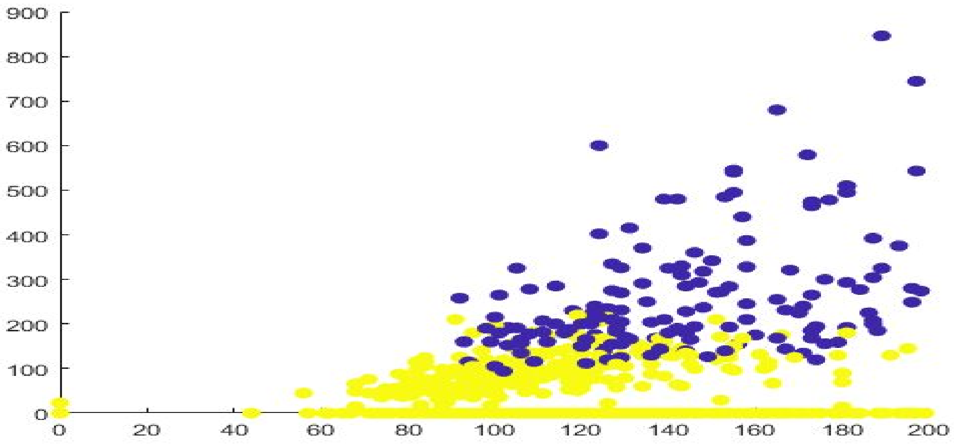
```matlab
clc;
clear all;
close all;


data = load('diabetes.txt');

%number of nodes is 3
global nDim nNeurons
nNeurons =2;
nDim =8;
learning_rate =0.1;
n = size(data(:,1))
disp(n);
learning_rate =0.2;
weights=ones(nDim,nNeurons);
for i=1:nDim
maxi = max(data(:,i));
mini = min(data(:,i));
weights(i,:) = mini + (maxi-mini).*rand (1,nNeurons);
end
```



```
nIterations =100;

for k=1:nIterations
 for c=1:n

    min_dist = 100000;
    dist = zeros(nNeurons,1);
    for i=1:nNeurons
    for j=1:nDim
        dist(i) = dist(i)+((data(c,j) - weights(j,i))^ 2);
    end
    dist(i) = sqrt(dist(i));


        if(min_dist>dist(i))
            indices(c,1)=i;
            min_dist=dist(i);
        end

    end

    for j =1:nDim
        weights(j,indices(c,1)) = weights(j,indices(c,1)) + learning_rate*(data(c,j)-weights(j,indices(c,1)));
```



```
      % weights = updateweights(weights,nNeurons,nDim,data,c,learning_rate);
  end
end
figure();
disp(weights);
scatter(data(:,2),data(:,5),50,indices,'filled');
ans = confusionmat(indices,data(:,9));
disp(ans);
```



Confusion matrix

426	198
74	70

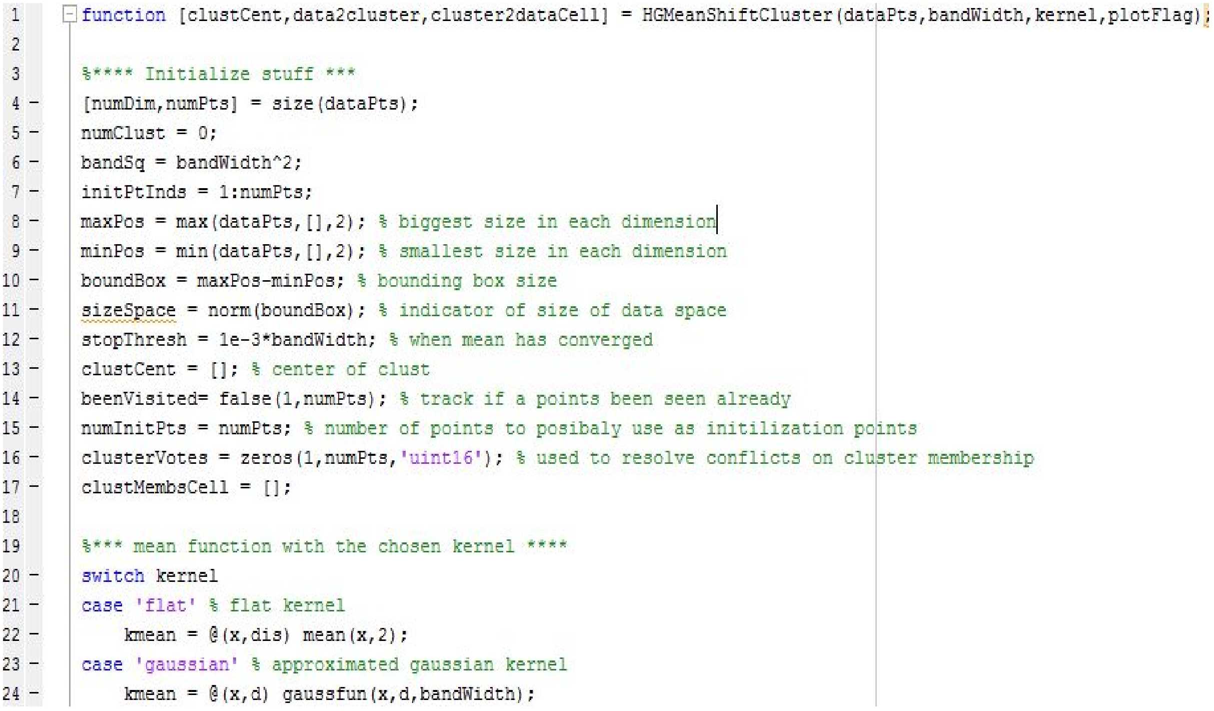
Mean-Shift Clustering

Concept

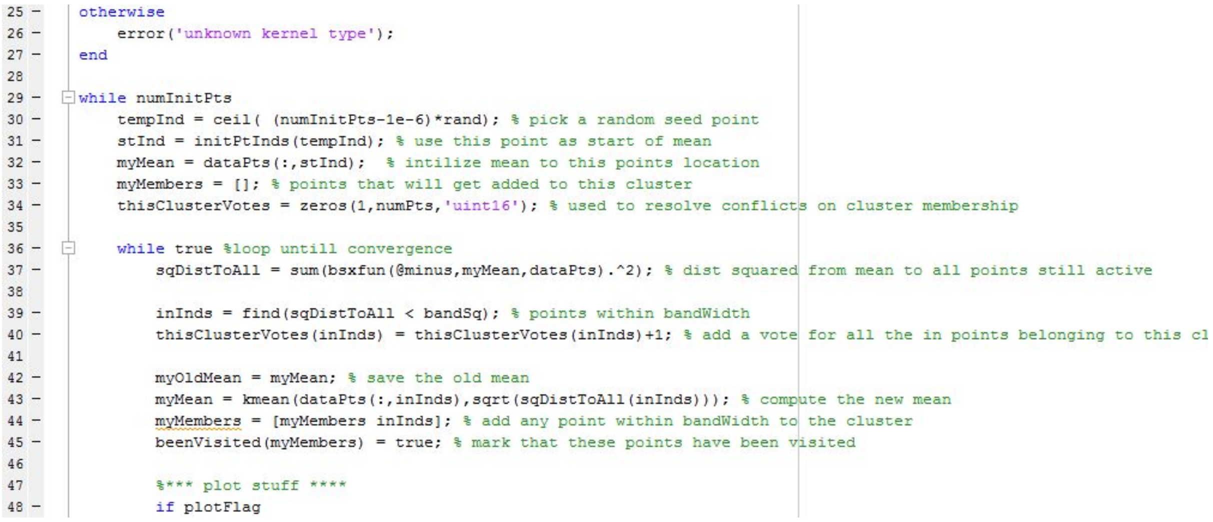
- Mean Shift is a powerful and versatile non parametric iterative algorithm that can be used for lot of purposes like finding modes, clustering etc.
- Mean shift considers feature space as an empirical probability density function. If the input is a set of points, then Mean shift considers them as sampled from the underlying probability density function. If dense regions (or clusters) are present in the feature space, then they correspond to the mode (or local maxima) of the probability density function. We can also identify clusters associated with the given mode using Mean Shift.
- To estimate the density function of a random variable, we will be using the kernel density function. We have used the Gaussian kernel function here.

Code:

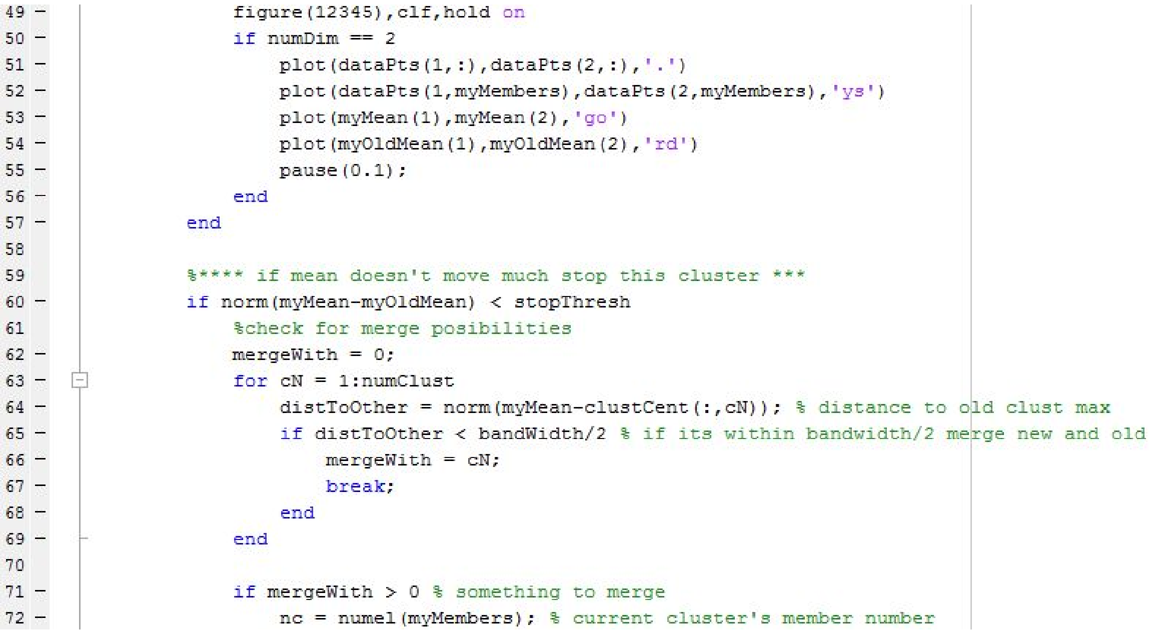
- First we create a function for Mean shift clustering called the HGMeanShiftClustering.m whose parameters are the data matrix, the bandwidth size ,the kernel and the plotflag.
- The data matrix the input data that we have to cluster.
- The bandwidth size is the bandwidth we need to set for the mean shift clustering.
- There are several kinds of kernel that we can give as argument like the ‘gaussian’ and the ‘flag’ etc.
- The plot flag is the flag for whether we want to get a plot of the output or not.For our example the plot flag is 0.



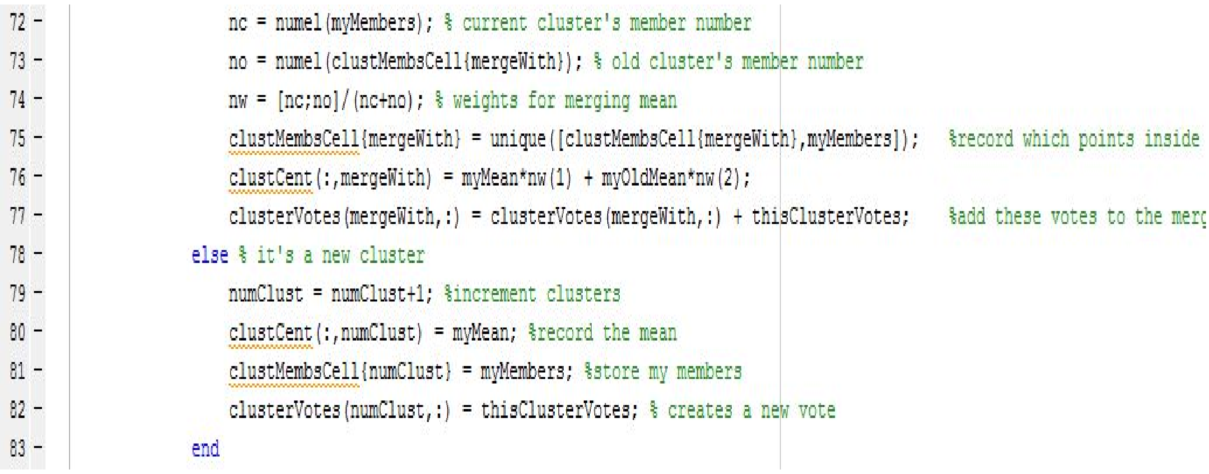
```matlab
function [clustCent,data2cluster,cluster2dataCell] = HGMeanShiftCluster(dataPts,bandWidth,kernel,plotFlag);

%**** Initialize stuff ***
[numDim,numPts] = size(dataPts);
numClust = 0;
bandSq = bandWidth^2;
initPtInds = 1:numPts;
maxPos = max(dataPts,[],2); % biggest size in each dimension
minPos = min(dataPts,[],2); % smallest size in each dimension
boundBox = maxPos-minPos; % bounding box size
sizeSpace = norm(boundBox); % indicator of size of data space
stopThresh = 1e-3*bandWidth; % when mean has converged
clustCent = []; % center of clust
beenVisited= false(1,numPts); % track if a points been seen already
numInitPts = numPts; % number of points to posibaly use as initilization points
clusterVotes = zeros(1,numPts,'uint16'); % used to resolve conflicts on cluster membership
clustMembsCell = [];

%*** mean function with the chosen kernel ****
switch kernel
case 'flat' % flat kernel
    kmean = @(x,dis) mean(x,2);
case 'gaussian' % approximated gaussian kernel
    kmean = @(x,d) gaussfun(x,d,bandWidth);
```



```matlab
otherwise
    error('unknown kernel type');
end

while numInitPts
    tempInd = ceil( (numInitPts-1e-6)*rand); % pick a random seed point
    stInd = initPtInds(tempInd); % use this point as start of mean
    myMean = dataPts(:,stInd);  % intilize mean to this points location
    myMembers = []; % points that will get added to this cluster
    thisClusterVotes = zeros(1,numPts,'uint16'); % used to resolve conflicts on cluster membership

    while true %loop untill convergence
        sqDistToAll = sum(bsxfun(@minus,myMean,dataPts).^2); % dist squared from mean to all points still active

        inInds = find(sqDistToAll < bandSq); % points within bandWidth
        thisClusterVotes(inInds) = thisClusterVotes(inInds)+1; % add a vote for all the in points belonging to this cl

        myOldMean = myMean; % save the old mean
        myMean = kmean(dataPts(:,inInds),sqrt(sqDistToAll(inInds))); % compute the new mean
        myMembers = [myMembers inInds]; % add any point within bandWidth to the cluster
        beenVisited(myMembers) = true; % mark that these points have been visited

        %*** plot stuff ****
        if plotFlag
```



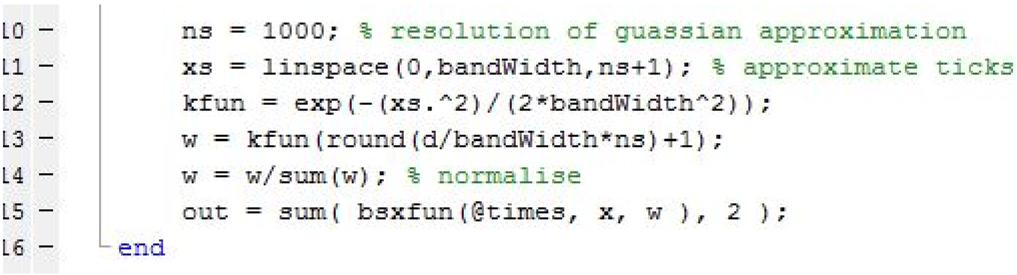
```
            figure(12345),clf,hold on
            if numDim == 2
                plot(dataPts(1,:),dataPts(2,:),'.')
                plot(dataPts(1,myMembers),dataPts(2,myMembers),'ys')
                plot(myMean(1),myMean(2),'go')
                plot(myOldMean(1),myOldMean(2),'rd')
                pause(0.1);
            end
        end

        %**** if mean doesn't move much stop this cluster ***
        if norm(myMean-myOldMean) < stopThresh
            %check for merge posibilities
            mergeWith = 0;
            for cN = 1:numClust
                distToOther = norm(myMean-clustCent(:,cN)); % distance to old clust max
                if distToOther < bandWidth/2 % if its within bandwidth/2 merge new and old
                    mergeWith = cN;
                    break;
                end
            end

            if mergeWith > 0 % something to merge
                nc = numel(myMembers); % current cluster's member number
```



```
                nc = numel(myMembers); % current cluster's member number
                no = numel(clustMembsCell{mergeWith}); % old cluster's member number
                nw = [nc;no]/(nc+no); % weights for merging mean
                clustMembsCell{mergeWith} = unique([clustMembsCell{mergeWith},myMembers]);   %record which points inside
                clustCent(:,mergeWith) = myMean*nw(1) + myOldMean*nw(2);
                clusterVotes(mergeWith,:) = clusterVotes(mergeWith,:) + thisClusterVotes;   %add these votes to the merg
            else % it's a new cluster
                numClust = numClust+1; %increment clusters
                clustCent(:,numClust) = myMean; %record the mean
                clustMembsCell{numClust} = myMembers; %store my members
                clusterVotes(numClust,:) = thisClusterVotes; % creates a new vote
            end
```

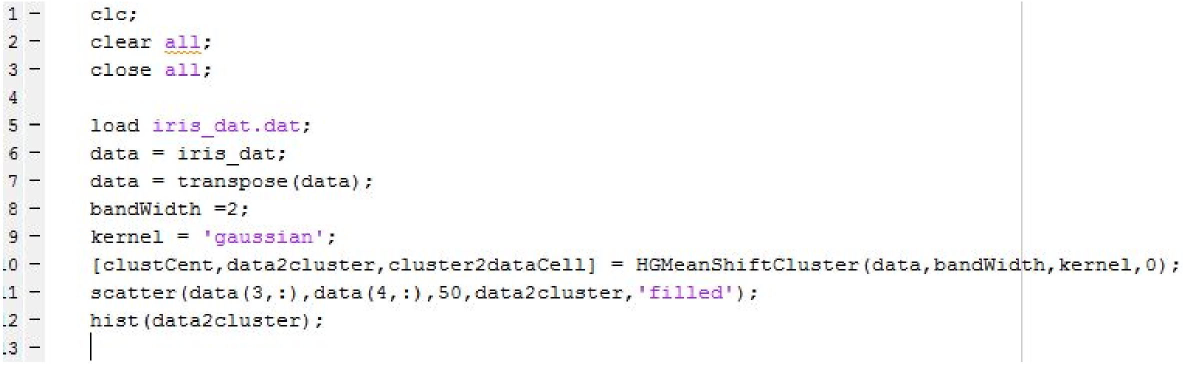
CODE FOR THE KERNEL-

For calculation purpose we use Gaussian function.



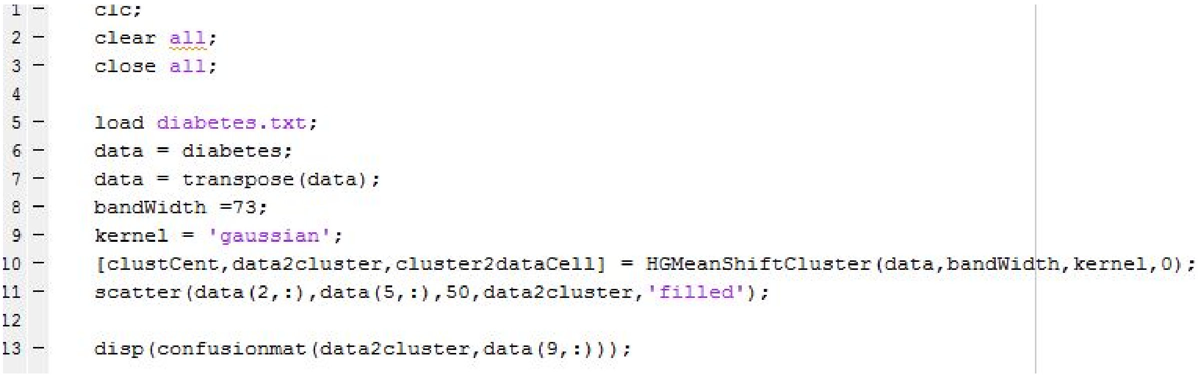
```
    ns = 1000; % resolution of guassian approximation
    xs = linspace(0,bandWidth,ns+1); % approximate ticks
    kfun = exp(-(xs.^2)/(2*bandWidth^2));
    w = kfun(round(d/bandWidth*ns)+1);
    w = w/sum(w); % normalise
    out = sum( bsxfun(@times, x, w ), 2 );
end
```

IRIS DATA



```
clc;
clear all;
close all;

load iris_dat.dat;
data = iris_dat;
data = transpose(data);
bandWidth =2;
kernel = 'gaussian';
[clustCent,data2cluster,cluster2dataCell] = HGMeanShiftCluster(data,bandWidth,kernel,0);
scatter(data(3,:),data(4,:),50,data2cluster,'filled');
hist(data2cluster);
```

PIMA DIABETES DATASET

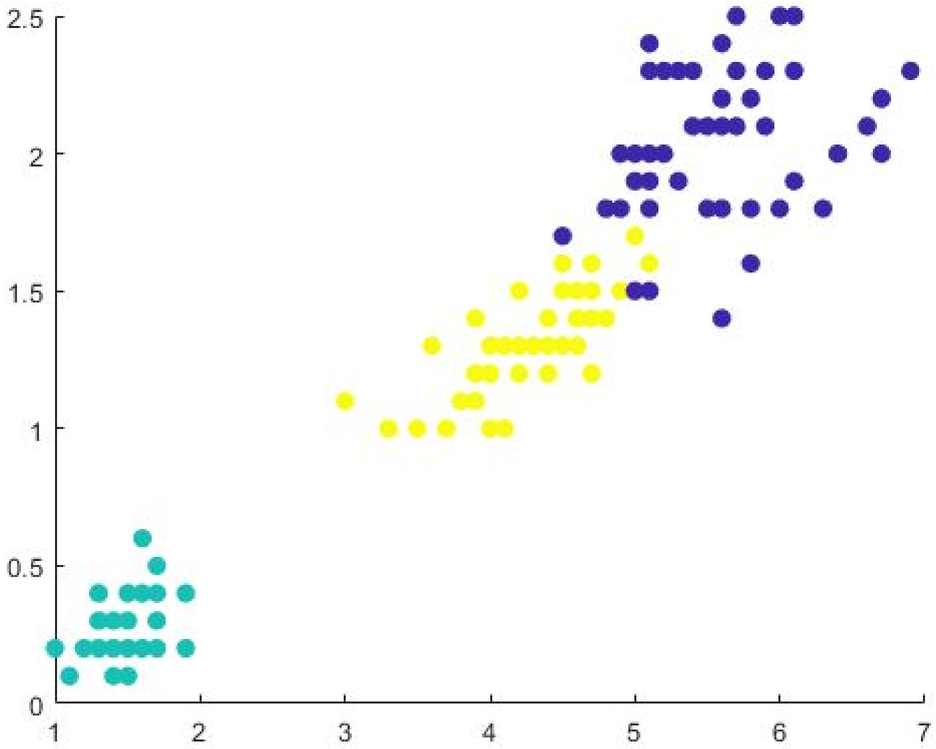


```
clc;
clear all;
close all;

load diabetes.txt;
data = diabetes;
data = transpose(data);
bandWidth =73;
kernel = 'gaussian';
[clustCent,data2cluster,cluster2dataCell] = HGMeanShiftCluster(data,bandWidth,kernel,0);
scatter(data(2,:),data(5,:),50,data2cluster,'filled');

disp(confusionmat(data2cluster,data(9,:)));
```

Output

1. **For iris data:**

a. **Bandwidth(h parameter=1)**



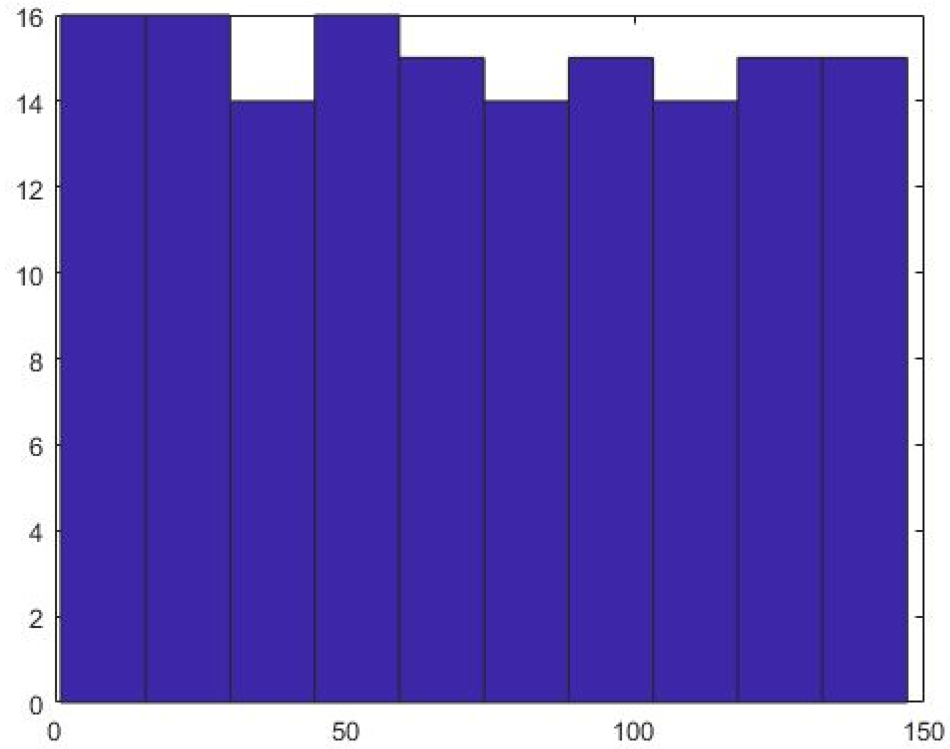
Histograms for Iris data:

X- axis:num of clusters

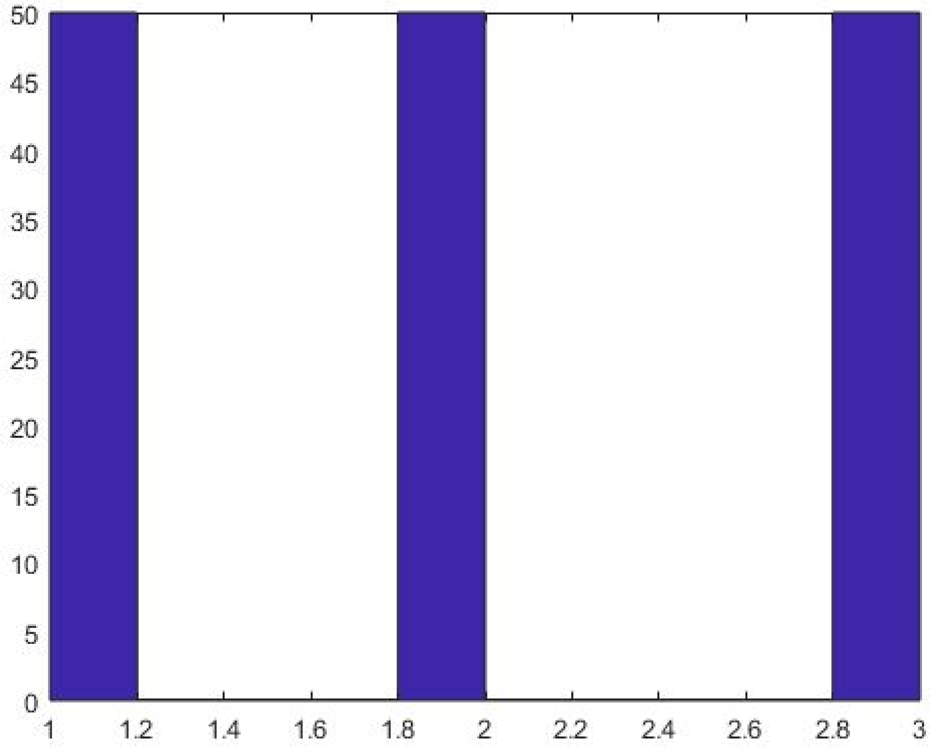
Y-axis: num of data-sets per cluster

h= Bandwidth Parameter

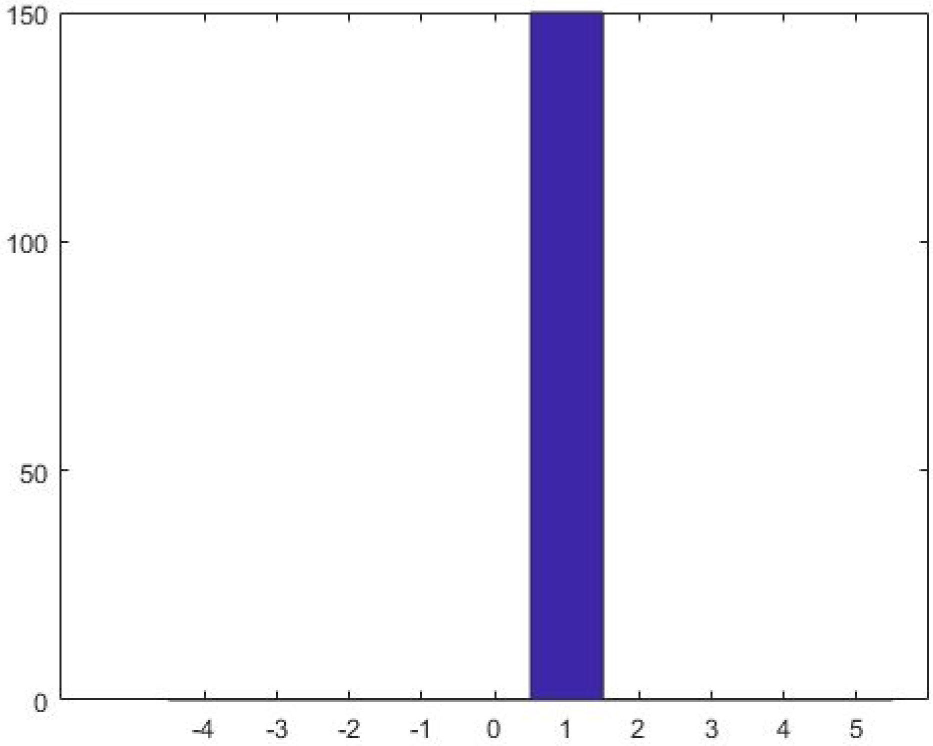
1. Iris_overclustering h=0.1
   (No of Clusters=150)



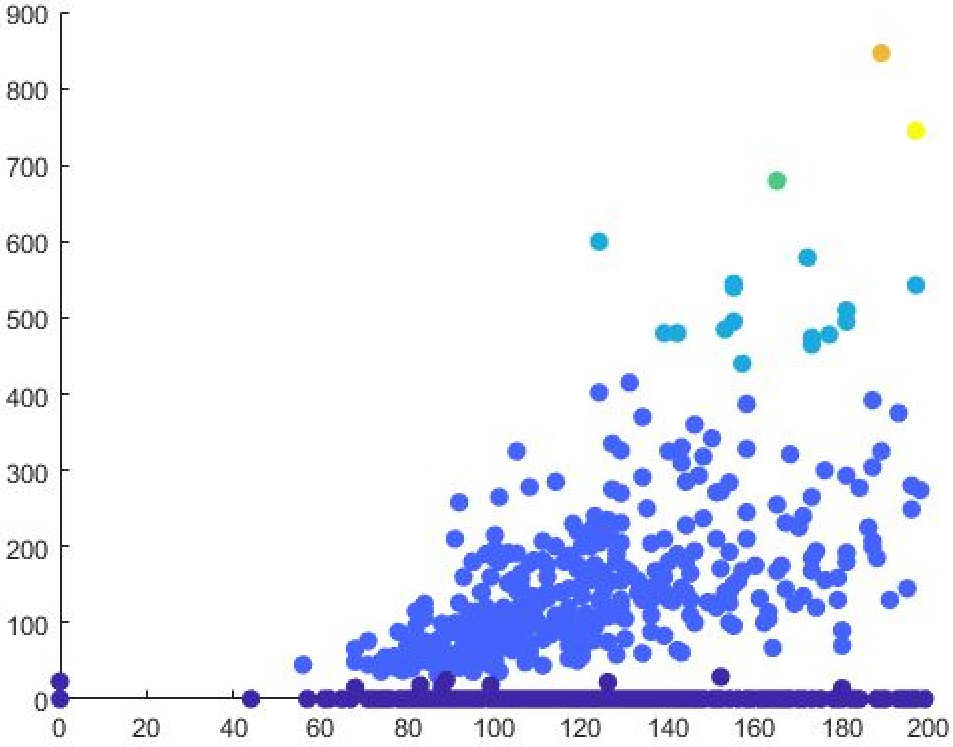
Normal Cluster -Iris Plot h=1
(No of Clusters= 3)



Under Clustering h = 5
(No of clusters=1)

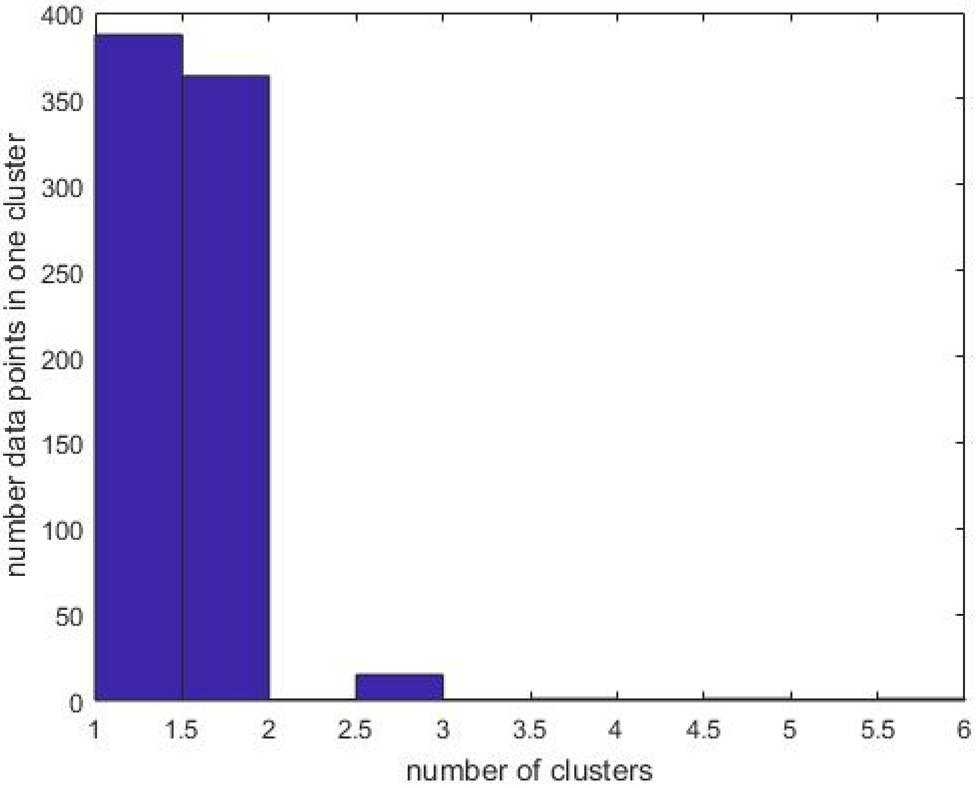


2. Diabetes Data-set

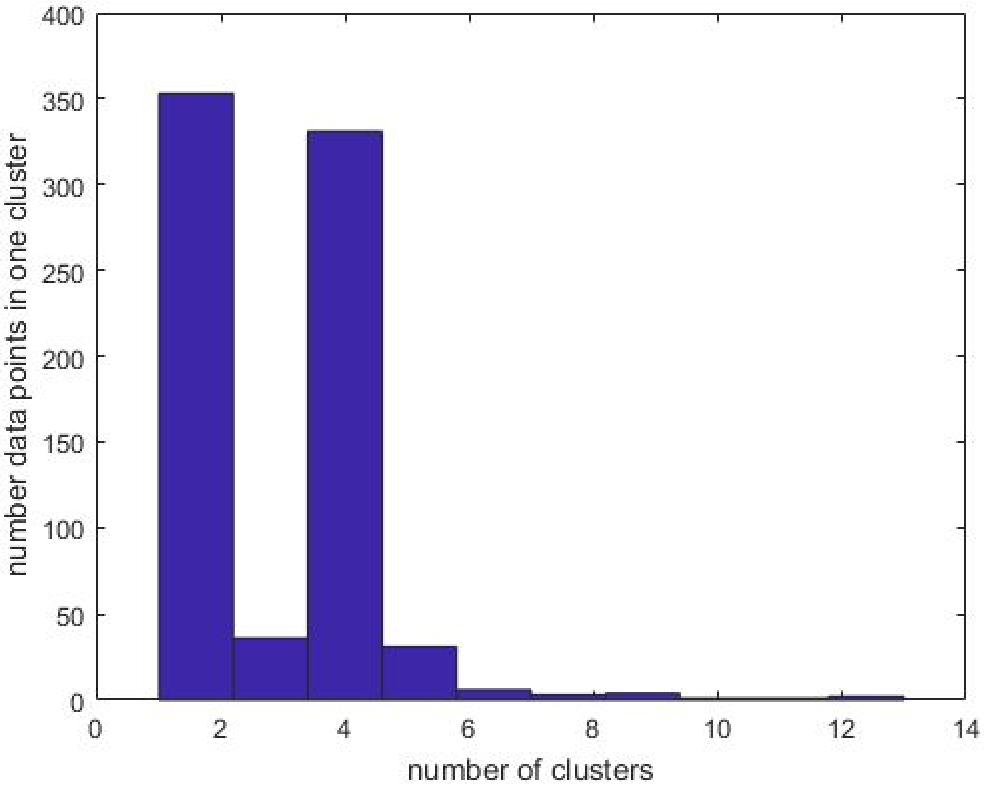


Histogram for Diabetes Data-set:

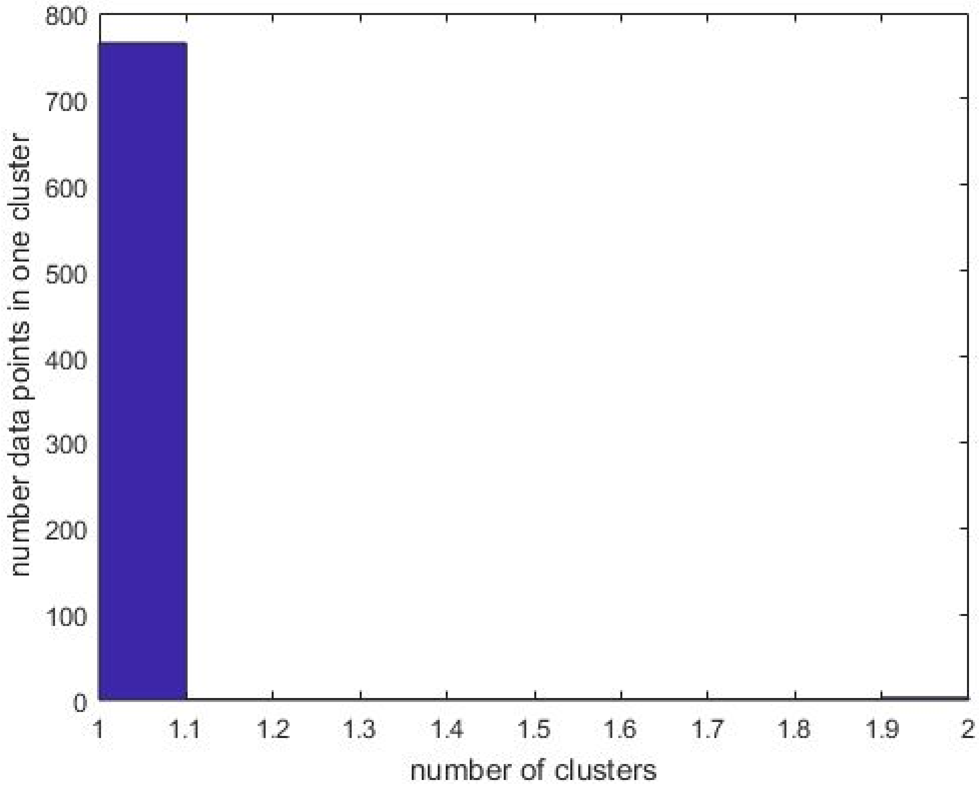
Normal Clustering h=72
(No of Clusters=3)



Over-clustering h=40
(No of Clusters=12)



Under-clustering h=100
(No of Clusters=1)



Observations:

- By tuning proper bandwidth parameter, we get:
- 3 clusters for Iris Data-set for bandwidth(h)=1
- 2 clusters(+1 misclassified) for Diabetes data-set for bandwidth(h)=72.

Changing the bandwidth leads to under-clustering and over-clustering.

Analysis:

- We could see that the mean-shift algorithm is time intensive.The time complexity of mean-shift algorithm is O(T*n^2) where
  T= number of iterations
  n= number of data points in the data set
- As we increase the data points, time complexity increases.

As we saw in Iris-data set which has 150 data points and Diabetes data-set which has 768 data points, time taken to reach convergence is more in Diabetes data-set.

Solution: We searched some methods to make the mean shift algorithm to converge faster.

1. The adaptive Mean Shift method is where you let the bandwidth parameter vary for each data point. The h parameter is calculated using Knn (k-nearest neighbour method) algorithm. Different distance metric methods are used in the Knn algorithm.

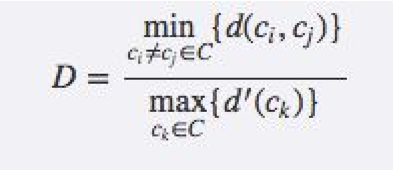
2. An alternate way to speed up convergence is to alter the data points during the course of Mean Shift.

II. Even though mean shift is a non-parametric algorithm, it does require the bandwidth parameter h to be tuned. We can use kNN to find out the bandwidth. The choice of bandwidth in influences convergence rate and the number of clusters. The ideal bandwidth which results in perfect number of clusters in case of Iris Data is 1 and in case of Diabetes data is 72.

A large h results in incorrect clustering and merges distinct clusters. A very small h results in too many clusters.

- **Dunn Index**

The Dunn Index assesses the goodness of a clustering, by measuring the maximal diameter of clusters and relating it to the minimal distance between clusters. This measure is quite conservative and prone to outliers, since it bases its calculation only on minimal and maximal distances.



$$D = \frac{\min\limits_{c_i \neq c_j \in C} \{d(c_i, c_j)\}}{\max\limits_{c_k \in C} \{d'(c_k)\}}$$

- **Davies–Bouldin index**

The **Davies–Bouldin index (DBI)** is a metric for evaluating clustering algorithms. This is an internal evaluation scheme, where the validation of how well the clustering has been done is made using quantities and features inherent to the dataset. This has a drawback that a good value reported by this method does not imply the best information retrieval.

Observation:

- The very notion of "good clustering" is relative. The same applies for the quality measures. It depends on what you want to evaluate.
- For instance, the Davies-Bouldin Index evaluates intra-cluster similarity and inter-cluster differences. If you consider these to be good criteria, go for the Davies-Bouldin.

- Dunn index does not give precise output if there are many points.
- When there are many distinct or different groups output of DB index is better.
- DB index gives better output when there is noisy data
- Dunn index takes more time

Applications of Mean-shift Algorithm:

i. The most important application is using Mean Shift for clustering. The fact that Mean Shift does not make assumptions about the number of clusters or the shape of the cluster makes it ideal for handling clusters of arbitrary shape and number.

ii. Mean Shift is used in multiple tasks in Computer Vision like segmentation, tracking, discontinuity preserving smoothing etc.

iii. Although, Mean Shift is primarily a mode finding algorithm, we can find clusters using it. The stationary points obtained via gradient ascent represent the modes of the density function. All points associated with the same stationary point belong to the same cluster.

Comparison with K-means algorithm:

We studied K-means algorithm in the parametric clustering and based on this algorithm is ISO-DATA algorithm for non-parametric clustering.
We thus give a basic comparison between K-means algorithm and Mean-Shift
One of the most important difference is that K-means makes two broad assumptions – the number of clusters is already known and the clusters are shaped spherically (or elliptically). Mean shift, being a non-parametric algorithm, does not assume anything about number of clusters. The number of modes give the number of clusters. Also, since it is based on density estimation, it can handle arbitrarily shaped clusters. K-means is very sensitive to initializations. A wrong initialization can delay convergence or sometimes even result in wrong clusters. Mean shift is fairly robust to initializations. Typically, mean shift is run for each point or sometimes points are selected uniformly from the feature space. Similarly, K-means is sensitive to outliers but Mean Shift is not very sensitive. K-means is fast and has a time complexity O(k*n*T), where k is the number of clusters, n is the number of points and T is the number of iterations. Classic mean shift is computationally expensive with a time complexity O(T*n*n). Mean shift is sensitive to the selection of bandwidth, h. A small h can slow down the convergence. A large h can speed up convergence but might merge two modes. But still, there are many techniques to determine h reasonably well.