CSCI-B 565 DATA MINING
Project Report for K-means Clustering algorithm
Computer Science Core
Fall 2012
Indiana University

Jayesh Kawli
jkawli@indiana.edu

09/17/2012

All the work herein is solely mine.

1. Examining Wolberg's breast cancer data

   (a) i. Data contains 699 Records

   ii. No. This is not a large dataset

   iii. If we ignore Sample code Number, (SCN) Δ has 10 attributes

   (b) There are total 16 missing values in given data.
   Missing values are as follows

```
1057013
1096800
1183246
1184840
1193683
1197510
1241232
169356
432809
563649
606140
61634
704168
733639
1238464
1057067
```

Dealing with unknown data

I preferred to remove tuples which contained missing data values. Main reason behind this is, if I go by usual way to replace unknowns by zero, it will add skew to resulting distribution There are total 699 records with 16 with missing values. Thus ratio of missing values to total values is 0.022. Although ratio is very small, it is not negligible. If I prefer to replace thus values with zeroes instead of removing those tuples, it will add skew and given that this is relatively small data set, effect of skew will be high and it will affect final clustering.

i. Although size of missing data which is 16 records is not very large. However compared to size of given data it is significant.

ii. Significance of strategy
Cluster size = 10
PPV value when unknown tuples were removed 0.952862
PPV value when unknown values were replaced by fixed unknown value i.e. Zero 0.925207

Cluster size = 2
PPV value when unknown tuples were removed 0.955220
PPV value when unknown values were replaced by fixed unknown value (0) 0.947861

I analysed the result of two program runs for different k i.e. number of clusters to form. In the first run,I did not consider records with unknown values, while in second, unknown values were replaced by constant (zero) for all the records
Result of final PPV value is populated in following table

Run	Cluster size	Eliminating missing	Replacing missing
1	2	0.955220	0.947861
2	10	0.952862	0.925207
3	15	0.919564	0.8973471
4	30	0.954679	0.806545

Data analysis observed in final data

- Final PPV value was slightly improved when tuples with unknown values were removed instead of replacing unknown values with 0 or fixed constant

- Even though above results reflect that, eliminating unknowns give better results than replacing them with fixed constant value, result depends upon K centroids chosen at the beginning of clustering. Depending upon result of random function, every time there is slight change in final PPV value for each run.

- It was found that for lesser number of classifiers (only two of them), PPV value decreases with increase in number K.In addition to it, algorithm takes more number of iteration to converge to final solution with increase in number of clusters. Result is shown below

Run	Number of clusters	Iterations before convergence	PPV value	Time(Seconds)
1	2	4	0.953722	1.09
2	10	6	0.950336	1.736
3	20	12	0.931893	2.264
4	50	14	0.883532	5.565
5	100	16	0.833298	8.764

However above numbers are specific to choice of initial centroids thus chosen. However, when experiment is repeated with multiple runs, result seemed to be following the above mentioned rule.

Time thus mentioned are average of 4 runs. However time is subjected to change each time depending upon initial choice of centroids and number of iterations to converge to solution.

(c)

(d) Let Δ^* be a cleaned Δ the tuples with the missing values are removed.

(e) K means clustering algorithm implementation

Following points explain protocol for choosing several parameter and decisions while implementing K-means clustering algorithm

i. Initializing Centroids
First we set the value of total number of final clusters required.SCN of each record is stored in an array numbered from 0 to N-1 where N is the total number of data records.
Centroids are picked up at random from this array using in built random function in C++. I also kept check for repeat random value. If the SCN is already picked up, it again goes to generate another random centroid using same random function. Random function is set to give values only between 0 to N-1.
This process is repeated at least K times until all the initial centroids In case there is collision, process is repeated ¿K times.

ii. Maintaining K centroids
K centroids thus chosen are stored in a Map data structure provided by C++. These centroids are stored in a Map where centroids themselves are stored as key and its point members belonging to cluster of that centroid are stored in a vector which is stored in a same Map as value
For each point, I calculated its euclidean distance from all the centroids. Point becomes member of the centroid to which it has minimum distance from

iii. Splitting centroids
For each point, I calculated its euclidean distance from all the centroids. Point becomes member of the centroid to which it has minimum distance from
At the end of iteration, where all the points are assigned to their respective centroids, we take average of all attributes of all points belonging to that centroid. The resulting vector containing average attribute values is now a new centroid.This process is repeated for remaining clusters.
Same process is repeated going over again until convergence occurs. That is there is negligible difference between previous and current attribute values for all the centroids

iv. Deciding ties
For each point, euclidean distance is calculated and point belongs to cluster which is having minimum distance from that point among all the other clusters.In case there is distance tie between point and more than one centroids, point goes into cluster of very first centroid
For e.g. there is point p and two centroids c_1 and c_2 such that dist(c_1,p=dist(c_2.),p) and centroid c_1 appears first, in that case, point p gets into first c_1 instead of getting into c_2 in next iteration

v. Stopping criteria
K mean algorithm continuously generates set of new centroids with modified attribute values until convergence occurs. Convergence is defined as a case where no new centroids values are generates.At each pass we calculate sum of city block distances between current and old attributes values for each centroid

This sum is then compared against pre set threshold (which is generally set as <2).When this condition is satisfied, we say that centroids are no longer changing and convergence has occurred. This is stopping criteria for K-Means algorithm where points do not change their cluster membership

After stopping criteria, we get K clusters with their respective data points and centroids. Algorithm takes more time to converge given higher number of clusters. This is because with increasing K we get more number of resulting clusters. When sum of differences of these clusters is taken, value is large initially. And hence algorithm will take more number of iterations to get converged. Where in each iteration this difference goes on decreasing.

vi. Table that takes pairs $C_{km=2}s$ PPV using the following attributes shown below

$C_{km=2(\Delta^*)}$	PPV
$A_1 \dots A_9$	0.953966
$A_1 \dots A_7$	0.954811
$A_1 \dots A_5$	0.946951
$A_1 \dots A_3$	0.938298
A_1, A_2	0.929330

Analysis

A. Thus from above analysis, it was observed that, even though we decrease value of number of attributes to form respective clusters,we still get higher value of PPV which indicates that each attribute acts as a strong classifier for given data.

B. However, it is observed that, as I reduce the number of attributes, PPV value decreases slightly but not much. It still appears to be more than 90 %

Assessing function using V -fold cross validation.

Train	Test	PPV Result
$C_{km=2(D^*-d_1^*)}$	$C_{km=2}(d_1^*)$	0
$C_{km=2(D^*-d_2^*)}$	$C_{km=2}(d_2^*)$	1
$C_{km=2(D^*-d_3^*)}$	$C_{km=2}(d_3^*)$	1
$C_{km=2(D^*-d_4^*)}$	$C_{km=2}(d_4^*)$	0
$C_{km=2(D^*-d_5^*)}$	$C_{km=2}(d_5^*)$	1
$C_{km=2(D^*-d_6^*)}$	$C_{km=2}(d_6^*)$	1
$C_{km=2(D^*-d_7^*)}$	$C_{km=2}(d_7^*)$	1
$C_{km=2(D^*-d_8^*)}$	$C_{km=2}(d_8^*)$	1
$C_{km=2(D^*-d_9^*)}$	$C_{km=2}(d_9^*)$	1
$C_{km=2(D^*-d_{10}^*)}$	$C_{km=2}(d_{10}^*)$	1

a weighted PPV, $\frac{1}{10}\sum_{i=1}^{10} PPV_i$

Thus value of weighed PPV from all iterations $= \frac{8}{10} = 0.8$

Analysis

- In order to perform V- Fold cross validation, initial data was cleaned. Cleaning involved adjusting duplicates and removing tuples with unknown values as it causes skew in final distribution.After filtering, total number of records are 630.
- Divided given data in 10 equal sized data set.
- Total 10 iterations were performed on input data. Where each iteration consisted of 1 test data and remaining 9 as training data set.
- Each data set acted as test data set once and training data set remaining 9 times.
- At the end of each iteration, label of test dataset was compared against training data set. Whenever there is match we get PPV value of 1 otherwise we took PPV as 0.
- At the end of all 10 iterations, I calculated weighed PPV value using following formula

  $\frac{1}{10}\sum_{i=1}^{10} PPV_i$

- Result of final PPV and PPV value at each of the 10 iterations is shown in table above

2. Assume one stopping criteria (often called epochs) is m. What is the complexity of $C_{km} = k$ in m epochs? Assume $|\Delta| = n$

Solution
Given there are n elements in data set.we have to run K-means for km=k. Where k is the number of final clusters. Hence in each iteration, program runs k*n times since it has to compare distance of all n points with all k centroids.

However, there is no guarantee that program might stop immediately after first iteration. If the convergence does not occur, it might go on run for m number of times(Epoch of algorithm). In that case, k*n computations will be

performed m times. Since k*n is calculated during each loop iteration.

Thus for program with input dataset size n, number of cluster k and epoch m, complexity can be given as follows

Complexity =O(k*n*m)

I analysed my program for different parameters. Result of analysis is as follows
(n=630 which is obtained after adjusting repeat values and eliminating tuples with unknown values)

Run	n	Clusters (k)	Epoch (m)	PPV	Time(Seconds)	Complexity
1	630	2	4	0.953722	1.09	5040
2	630	10	6	0.950336	1.736	37800
3	630	20	12	0.931893	2.264	151200
4	630	50	14	0.883532	5.565	441000
5	630	100	16	0.833298	8.764	1008000

Appendix

A Implementation of K-Means clustering algorithm Using C++

```
#include<iostream>
#include<conio.h>
#include<fstream.h>
#include<map>
#include<time.h>
#include<vector>
#include<string>
#include<stdlib.h>
#include<math.h>
using namespace std;
void ckm(map<long,vector<int> >,int, int,map<int,long> );
int main()
{
    map<long,vector<int> > datastore;
    int temp=0;
    ofstream missingscns;

    /*File to write SCNs having row with unknown values */
    missingscns.open ("r.txt");

    int threshold=2;
   int clustersize;
   cout<<"How many clusters you want?"<<endl;
   cin>>clustersize;

    char buffer[40];
    map<int,long> tempholder;
    int count=0;
    int uni=0;
    ifstream breastcancerdata("bcdata.txt");

    while(!breastcancerdata.eof())
    {
        bool isunknown=false;
```

```
vector<int> attributeholder;
char *indiattri;
breastcancerdata.getline(buffer,40);

/*Break input row into two parts:
First part contains SCN while other
contains remaining attributes */

char *pch = strtok (buffer,",");
long first=atol(pch);
if(!first)
{
    break;
}
char *pch1 = strtok (NULL, ",");
while (pch1 != NULL)
{
    if((string)pch1=="?")
    {
        missingscns<<first<<"\n";
        isunknown=true;
    }
    temp=atoi(pch1);
    attributeholder.push_back(temp);
    pch1 = strtok (NULL, ",");
}
if(isunknown==true)
{
    /* Don't consider records with unknown values.
    Skip to next record */

    continue;
}

if(datastore.count(first))
{
    /* Check for duplicates
    If duplicate SCN already exists, take the
    average of all attributes and
    store with same SCN in map */

    vector<int> tempforrepeat;
    vector<int> existing=datastore[first];
    for(int j=0; j<attributeholder.size()-1; j++)
    {
        int approx=(existing.at(j)+attributeholder.at(j))/2;
        tempforrepeat.push_back(approx);
    }
    tempforrepeat.push_back(attributeholder.at(9));
    datastore[first]=tempforrepeat;
}
else
{
    /* If no duplicate element found in a map,
    store row with SCN as key and corresponding
    attributes as value in a map */

    /*tempholder array stores SCN of all the records
    eleminating duplicates.
```

```
            Values stored range from 0 to M-1 where M is the
            total number of unique keys in a given data */

            datastore[first]=attributeholder;
            tempholder[count++]=first;
        }
    }

            /*Calling K-means clustering routines
            datastore - map which contains all the SCNs with their respective attributes
            clustersize - Total number of final clusters to form K=clustersize
            threshold - Parameter to assess diffrence between previous and current centroid positions
            When difference goes below threshold, it meets convergence condition
            */

    ckm(datastore,clustersize,threshold,tempholder);
    missingscns.close();
    getch();
    return 1;
}


void ckm(map<long,vector<int> > datastore,int clustersize,int threshold,map<int,long> tempholder)
{
    map<long,vector<int> >::iterator iter;
    map<int,long>::iterator iter1;
    int count123=0;
    srand((unsigned)time(0));
    vector<long> initcentroids;
    map<int,long> checkdupli;
    int prevalue=0;
    map<long,vector<int> > oldvaluecentroid;

    /* Routine to generate initial set of centroids from given data
    K centroids are chosen using C++ rand() function */

    while(count123<clustersize)
    {
        int rannumber=rand()%tempholder.size();
        if(checkdupli[rannumber]>-1)
        {
            vector<int> softvector;
            for(int i1=0; i1<datastore[tempholder[rannumber]].size(); i1++)
            {
                softvector.push_back(datastore[tempholder[rannumber]].at(i1));
            }
            initcentroids.push_back(tempholder[rannumber]);
            oldvaluecentroid[tempholder[rannumber]]=softvector;
            count123++;
            softvector.clear();

            /* Flag to avoid same centroid being generated twice */

            checkdupli[rannumber]=-1;
        }
        else
        {
            /* Same centroid generated using rand() function
            again loop back and generate other centroid using
```

```
        rand() function*/

        continue;

    }
}

/* Container to hold new centroid and its corresponding members */

map<long,vector<int> > newvaluecentroid;
vector<int> temp90;
bool flag=true;
map<long,vector<int> > indicluster;
map<long, vector<int> >::iterator it57;

/*Infinite while loop to generate the new set of centroids
Loop breaks whenever convergence occurs.
Convergence occurs when there is no change in the values of centroid attributes*/

int iterationcount=0;

while(1)
{
    iterationcount++;
    indicluster.clear();
    for(int j=0; j<datastore.size(); j++)
    {
        int cluster=0;
        double minimum=30000;
        double sum;
        vector<int> pointdata=datastore[tempholder[j]];
        for(int centro=0; centro<initcentroids.size(); centro++)
        {
            sum=0.0;
            vector<int> centroiddata=oldvaluecentroid[initcentroids.at(centro)];

            /*Calculate euclidean distance of given point from set of all centroids*/

            for(int vl=0; vl<pointdata.size()-1; vl++)
            {
                sum+=sqrt(pow((pointdata.at(vl)-centroiddata.at(vl)),2));
            }

            /*If distance thus computed is less than distance of previous centroid,
            store it in a minimum variable */

            if(sum<minimum)
            {
                cluster=initcentroids.at(centro);

                minimum=sum;
            }
        }


            /*Point is assigned to centroid with minimum euclidean distance */

            indicluster[cluster].push_back(tempholder[j]);
    }
```

```
/* If no point is assigned to any centroid, store dummy value of zero for that
centroid. This is to avoid segmentation fault due to access to non existent
variable */

for(int centro=0; centro<initcentroids.size(); centro++)
{
    if(!(indicluster.count(initcentroids.at(centro))))
    {
        indicluster[initcentroids.at(centro)].push_back(0);
    }

}

map<long,vector<int> >::iterator ite2;
map<long,vector<int> > indiclusternew;

/*For each value of current centroid, iterate over all its member records.
Generate new centroid by averaging all attribute values */

for(ite2=indicluster.begin(); ite2!=indicluster.end(); ite2++)
{
    int *temparr=(int *)calloc(10,sizeof(int));
    vector<int> sumattri;
    for(int j=0; j<(ite2->second).size(); j++)
    {
        int fed=0;
        if((ite2->second).size()==1 && (ite2->second).at(0)==0)
        {
            break;
        }
        for(int k=0; k<9; k++)
        {
            temparr[k]=temparr[k]+datastore[(ite2->second).at(j)].at(k);
        }
        fed=0;
    }

    for(int co=0; co<9; co++)
    {
        int finvalue=(temparr[co])/((ite2->second).size());
        sumattri.push_back(finvalue);
    }

    long val = ite2->first;

    /*Push new centroid along with its attributes in a
    newvaluecentroid map data structure. These attributes
    are compared against old values of centroid to check
    if convergence occurred */

    indiclusternew[val]=sumattri;
    newvaluecentroid[val]=sumattri;
    sumattri.clear();
}

map<long,vector<int> >::iterator ittnew=newvaluecentroid.begin();
map<long,vector<int> >::iterator ittold=oldvaluecentroid.begin();
```

```
    int sumall=0;

    /*Iterate over every centroid. Check for attribute differences for each attribute.
    Take the absolute sum of these differences*/

    while(ittnew!=newvaluecentroid.end() && ittold!=oldvaluecentroid.end())
    {
        long temp60new=ittnew->first;
        long temp60old=ittold->first;
        for(int count12=0; count12<newvaluecentroid[temp60new].size(); count12++)
        {
            sumall+=
       (oldvaluecentroid[temp60old].at(count12)-newvaluecentroid[temp60new].at(count12));
        }
        ittnew++;
        ittold++;
    }

    cout<<"Absolute value of centroids difference in "
    <<iterationcount<<" th iteration is "<<sumall<<endl;

    /* If diffrence between current and old centroid values is less than threshold, then
    we reached the convergence condition. Break out of infinite while loop */

    if(sumall<threshold)
    {
        break;
    }

    /*For the next iteration clear all hashmap values*/

    initcentroids.clear();
    oldvaluecentroid.clear();

    /*Change current centroids to old centroids for next iteration */

    oldvaluecentroid=newvaluecentroid;
    map<long, vector<int> >::iterator itt;

    for(itt=newvaluecentroid.begin(); itt!=newvaluecentroid.end(); itt++)
    {
        long keyval=itt->first;
        initcentroids.push_back(keyval);

    }

    newvaluecentroid.clear();
    flag=false;
}

int checkvalue=0;
int four=0;
int two=0;
int tp=0;
int fp=0;
double ppv=0.0;

for(it57=indicluster.begin(); it57!=indicluster.end(); it57++)
{
```

```
four=0;
two=0;

/*If empty cluster is found with no members assigned to it,
        don't consider it. Go to next centroid */

if((it57->second).size()==1 && (it57->second).at(0)==0)
{
  continue;
}
checkvalue=datastore[it57->first].at(9);

/* Check for majority in each cluster
if points with class=4 are more than those
with class=2 then we name that as cluster of class 4
and vice versa.

After classification points mismatching with cluster
label are considered as false positives while those
matching with label are considered as true positives  */

for(int h=0; h<(it57->second).size(); h++)
{
    if(datastore[(it57->second).at(h)].at(9)==4)
    {
        four++;
    }
    else
    {
        two++;
    }
}

if(four>=two)
{
    for(int h=0; h<(it57->second).size(); h++)
    {
        if(datastore[(it57->second).at(h)].at(9)==4)
        {
            tp++;
        }
        else
        {
            fp++;
        }
    }
}

/* Calculate number of True positives and False positives in each cluster */

else
{
    for(int h=0; h<(it57->second).size(); h++)
    {
        if(datastore[(it57->second).at(h)].at(9)==2)
        {
            tp++;
        }
        else
```

```
                {
                    fp++;
                }
            }
        }

        /* Calculate and accumulate Positive predictive value for each cluster
        PPV value is calculated as

        ppv=(tp)/(tp+fp)

        */

        ppv+=(tp)/(double)(tp+fp);
        tp=0;
        fp=0;

    }

    /* Final average PPV value over k-clusters */

    double finalppv=ppv/(double)indicluster.size();
    cout<<endl<<endl<<"Final PPV value for  "<<clustersize<<" clusters is "<<finalppv<<endl;
}
```

B Source code for V-fold cross validation of breast cancer data

```
#include<iostream>
#include<conio.h>
#include<fstream.h>
#include<map>
#include<time.h>
#include<vector>
#include<string>
#include<stdlib.h>
#include<math.h>
using namespace std;
int ckm(map<long,vector<int> >,vector<int>, int,map<int,long> );
int main()
{
    map<long,vector<int> > datastore;
    int temp=0;


    /*File to write SCNs having row with unknown values */


    int clustersize=2;
    char buffer[40];
    map<int,long> tempholder;
    int count=0;
    int uni=0;
    ifstream breastcancerdata("bcdata.txt");

    while(!breastcancerdata.eof())
    {
        bool isunknown=false;
```

```
vector<int> attributeholder;
char *indiattri;
breastcancerdata.getline(buffer,40);

/*Break input row into two parts:
First part contains SCN while other
contains remaining attributes */

char *pch = strtok (buffer,",");
long first=atol(pch);
if(!first)
{
    break;
}
char *pch1 = strtok (NULL, ",");
while (pch1 != NULL)
{
    if((string)pch1=="?")
    {

        isunknown=true;
    }
    temp=atoi(pch1);
    attributeholder.push_back(temp);
    pch1 = strtok (NULL, ",");
}
if(isunknown==true)
{
    /* Don't consider records with unknown values.
    Skip to next record */

    continue;
}
if(datastore.count(first))
{
    /* Check for duplicates
    If duplicate SCN already exists, take the
    average of all attributes and
    store with same SCN in map */

    vector<int> tempforrepeat;
    vector<int> existing=datastore[first];
    for(int j=0; j<attributeholder.size()-1; j++)
    {
        int approx=(existing.at(j)+attributeholder.at(j))/2;
        tempforrepeat.push_back(approx);
    }
    tempforrepeat.push_back(attributeholder.at(9));
    datastore[first]=tempforrepeat;
}
else
{
    /* If no duplicate element found in a map,
    store row with SCN as key and corresponding
    attributes as value in a map */

    /*tempholder array stores SCN of all the records
    eleminating duplicates.
    Values stored range from 0 to M-1 where M is the
```

```
        total number of unique keys in a given data */

        datastore[first]=attributeholder;
        tempholder[count++]=first;
    }
}
double totalppv=0;

/*Calling K-means clustering routines
datastore - map which contains all the SCNs with their respective attributes
clustersize - Total number of final clusters to form K=clustersize
threshold - Parameter to assess diffrence between previous and current centroid positions
When difference goes below threshold, it meets convergence condition
*/
/* After eliminating unknown and duplicate entrie */
int itercount=0;
for(int iter=0; iter<630; iter+=63)
{

/*Divide data in 10 equal partitions */

    map<long,vector<int> > testmap;
    map<long,vector<int> > trainingmap;
    vector<int> centroiddata;
    testmap.clear();
    trainingmap.clear();
    trainingmap=datastore;
    int count4=0;
    int count2=0;
    for(int j=iter; j<iter+63; j++)
    {
        if(datastore[tempholder[j]].at(9)==4)
        {
            count4++;
        }
        else
        {
            count2++;
        }

        /* Add one block of records into test map */

        testmap[j%63]=datastore[tempholder[j]];

        /*Add remaining blocks of records into training map */

        trainingmap.erase(tempholder[j]);
    }

    for(int j=0; j<9; j++)
    {
        int sumattri=0;
        int finalsuma=0;
        for(int k=0; k<63; k++)
        {
            sumattri+=testmap[k].at(j);
        }

        /*Get centroid valu for test map */
```

```
            finalsuma=sumattri/testmap.size();
            centroiddata.push_back(finalsuma);
        }

        int label=2;

        /*Check what label test block belongs to */

        if(count2<count4)
        {
            label=4;
        }

        /*Get PPV value for successive test blocks with remaining 9 as training blocks*/
int temp=ckm(datastore,centroiddata,label,tempholder);
        cout<<"PPV value for iteration "<<itercount++<<" is "<<temp<<endl;
        totalppv+=temp;
        //cout<<totalppv<<"per iteration"<<endl;
    }
    cout<<"\n\nThis is Final weighed PPV value\n\n"<<(totalppv/(double)10)<<endl;
    getch();
    return 1;
}

int ckm(map<long,vector<int> > datastore,vector<int> centroiddata,int label,map<int,long> tempholder)
{
    map<long,vector<int> >::iterator iter;
    map<int,long>::iterator iter1;
    int count123=0;
    srand((unsigned)time(0));
    vector<long> initcentroids;
    map<int,long> checkdupli;
    int prevalue=0;
    map<long,vector<int> > oldvaluecentroid;
    int threshold=2;
    /* Routine to generate initial set of centroids from given data
    K centroids are chosen using C++ rand() function */

    while(count123<2)
    {
        int rannumber=rand()%tempholder.size();
        if(checkdupli[rannumber]>-1)
        {
            vector<int> softvector;
            for(int i1=0; i1<datastore[tempholder[rannumber]].size(); i1++)
            {
                softvector.push_back(datastore[tempholder[rannumber]].at(i1));
            }
            initcentroids.push_back(tempholder[rannumber]);
            oldvaluecentroid[tempholder[rannumber]]=softvector;
            count123++;
            softvector.clear();

            /* Flag to avoid same centroid being generated twice */

            checkdupli[rannumber]=-1;
        }
        else
```

```
    {
        /* Same centroid generated using rand() function
        again loop back and generate other centroid using
        rand() function*/

        continue;
    }
}

/* Container to hold new centroid and its corresponding members */

map<long,vector<int> > newvaluecentroid;
vector<int> temp90;
bool flag=true;
map<long,vector<int> > indicluster;
map<long, vector<int> >::iterator it57;

/*Infinite while loop to generate the new set of centroids
Loop breaks whenever convergence occurs.
Convergence occurs when there is no change in the values of centroid attributes*/

int iterationcount=0;

while(1)
{
    iterationcount++;
    indicluster.clear();
    for(int j=0; j<datastore.size(); j++)
    {
        int cluster=0;
        double minimum=30000;
        double sum;
        vector<int> pointdata=datastore[tempholder[j]];
        for(int centro=0; centro<initcentroids.size(); centro++)
        {
            sum=0.0;
            vector<int> centroiddata=oldvaluecentroid[initcentroids.at(centro)];

            /*Calculate euclidean distance of given point from set of all centroids*/

            for(int vl=0; vl<pointdata.size()-1; vl++)
            {
                sum+=sqrt(pow((pointdata.at(vl)-centroiddata.at(vl)),2));
            }

            /*If distance thus computed is less than distance of previous centroid,
            store it in a minimum variable */

            if(sum<minimum)
            {
                cluster=initcentroids.at(centro);
                minimum=sum;
            }
        }
        /*Point is assigned to centroid with minimum euclidean distance */
        indicluster[cluster].push_back(tempholder[j]);
    }
```

```
/* If no point is assigned to any centroid, store dummy value of zero for that
centroid. This is to avoid segmentation fault due to access to non existent
variable */

for(int centro=0; centro<initcentroids.size(); centro++)
{
    if(!(indicluster.count(initcentroids.at(centro))))
    {
        indicluster[initcentroids.at(centro)].push_back(0);
    }
}

map<long,vector<int> >::iterator ite2;
map<long,vector<int> > indiclusternew;

/*For each value of current centroid, iterate over all its member records.
Generate new centroid by averaging all attribute values */

for(ite2=indicluster.begin(); ite2!=indicluster.end(); ite2++)
{
    int *temparr=(int *)calloc(10,sizeof(int));
    vector<int> sumattri;
    for(int j=0; j<(ite2->second).size(); j++)
    {
        int fed=0;
        if((ite2->second).size()==1 && (ite2->second).at(0)==0)
        {
            break;
        }
        for(int k=0; k<9; k++)
        {
            temparr[k]=temparr[k]+datastore[(ite2->second).at(j)].at(k);
        }
        fed=0;
    }

    for(int co=0; co<9; co++)
    {
        int finvalue=(temparr[co])/((ite2->second).size());
        sumattri.push_back(finvalue);
    }
    long val = ite2->first;

    /*Push new centroid along with its attributes in a
    newvaluecentroid map data structure. These attributes
    are compared against old values of centroid to check
    if convergence occurred */

    indiclusternew[val]=sumattri;
    newvaluecentroid[val]=sumattri;
    sumattri.clear();
}

map<long,vector<int> >::iterator ittnew=newvaluecentroid.begin();
map<long,vector<int> >::iterator ittold=oldvaluecentroid.begin();
int sumall=0;

/*Iterate over every centroid. Check for attribute differences for each attribute.
Take the absolute sum of these differences*/
```

```
    while(ittnew!=newvaluecentroid.end() && ittold!=oldvaluecentroid.end())
    {
        long temp60new=ittnew->first;
        long temp60old=ittold->first;
        for(int count12=0; count12<newvaluecentroid[temp60new].size(); count12++)
        {
            sumall+=abs(oldvaluecentroid[temp60old].at(count12)-newvaluecentroid[temp60new].at(count
        }
        ittnew++;
        ittold++;
    }



    /* If diffrence between current and old centroid values is less than threshold, then
    we reached the convergence condition. Break out of infinite while loop */

    if(sumall<threshold)
    {
        break;
    }

    /*For the next iteration clear all hashmap values*/

    initcentroids.clear();
    oldvaluecentroid.clear();

    /*Change current centroids to old centroids for next iteration */

    oldvaluecentroid=newvaluecentroid;
    map<long, vector<int> >::iterator itt;
    for(itt=newvaluecentroid.begin(); itt!=newvaluecentroid.end(); itt++)
    {
        long keyval=itt->first;
        initcentroids.push_back(keyval);
    }
    newvaluecentroid.clear();
    flag=false;
}

int checkvalue=0;
int four=0;
int two=0;
int tp=0;
int fp=0;
double ppv=0.0;
double min=20000;
map<long,vector<int> >::iterator ittnew;
int sum22;
long required=0;
for(ittnew=newvaluecentroid.begin(); ittnew!=newvaluecentroid.end(); ittnew++)
{
    sum22=0;
    for(int in=0; in<(ittnew->second).size(); in++)
    {
        sum22+=abs(newvaluecentroid[ittnew->first].at(in)-centroiddata.at(in));
    }
    if(sum22<min)
```

```
        {
            min=sum22;
            required=ittnew->first;
        }
    }

    for(int j=0; j<indicluster[required].size(); j++)
    {
        four=0;
        two=0;


        /* Check for majority in each cluster
        if points with class=4 are more than those
        with class=2 then we name that as cluster of class 4
        and vice versa.

        After classification points mismatching with cluster
        label are considered as false positives while those
        matching with label are considered as true positives  */

        if(datastore[indicluster[required].at(j)].at(9)==4)
        {
            four++;
        }
        else
        {
            two++;
        }
        if(four>=two)
        {
            if(label==4)
            {
                return 1;
            }
            else
            {
                return 0;
            }
        }
        /* Calculate number of True positives and False positives in each cluster */
        else
        {
            if(label==4)
            {
                return 0;
            }
            else
            {
                return 1;
            }
        }
    }
    /* Final average PPV value over k-clusters */
}
```