Practical 3: k-means Clustering

Instructor: Shailesh B. Pandey (Everest Engineering College) Course: Intelligent System (BEIT)
Course webpage: sites.google.com/view/eec-is

1. Overview

In this practical you will implement a simplified k-means algorithm. k-means is a clustering algorithm most commonly used for unsupervised learning. In unsupervised learning the data do not have labels. For example, you are given images of cats and dogs and asked to learn to differentiate between them. In supervised learning you are told which images are cats and which are dogs. In k-means (unsupervised learning) data forms groups using their properties and characteristics. The idea is that, objects within the same cluster are as similar as possible (i.e., high intra-class similarity), whereas objects from different clusters are as dissimilar as possible. Distance measures (such as Euclidean) captures the notion of similarity and dissimilarity.

2. k-means Algorithm

The minimal implementation of k-means we discuss in the lab can be implemented in a few simple steps. The idea is given as below:

```
kmeans(k, data):
       Randomly initialize centre of k clusters
       do
       {
           for(each point p in the data) {
                Calculate distance between p and each of the k centres
                Assign point p to the closest centre
           }
           Re-compute centre from the new point assignments
       } until(centres do not change)
```

k-means starts with k random centres (centroids) that define the clusters. A point (data instance) is considered to be in a particular cluster if it is closer to that cluster's centre than to other clusters. k-means finds these centres by alternately assigning data points to clusters based on a current value of the centres, and then re-computing the centres based on the current assignment of data points to clusters. The algorithm stops when the centres do not change.

3. Data and Implementation

We will keep the data simple to focus on the algorithm. The data in our case is a set of points each with x and y coordinate values. These points will be randomly generated. For example, the data may look like: { (18,84), (83,95), (4,28), (33,53), (5,9) }. To make the implementation even simple, we will store the x and y values in two different arrays.

Once we have the data randomly generated we get to the k-means proper. The value of k is supplied by the user. We will then need to randomly initialize k cluster centres. Let us assume that we chose k=2. Let us also assume, the randomly generated centre of the first cluster (C1) is (1,5) and second (C2) is (10,10). Now we will assign each data point to either C1 or C2. To decide this we need to calculate distance between the data point and the two centres. We will use the standard Euclidean distance which we learned in school.

$$d_{euc}(P,Q) = \sqrt{\sum_{i=1}^{n}(P_i - Q_i)^2}$$

In our data n=2 (i.e. x and y coordinates). For example, let us calculate the distance between the first point P = (18,84) and C1 = (1,5).

$d_{euc}(P,C1) = \sqrt{(18-1)^2 + (84-5)^2}$ = 80.81

We also calculate distance between P and C2 which is 74.43. Since this is closer that with C2, we assign this point P to the C2 cluster. We do this for all the data points.

Once all the points are assigned to either C1 or C2, we have to update the centres. Cluster centre is nothing but average (mean) of all point in that cluster. Staying with the data { (18,84), (83,95), (4,28), (33,53), (5,9) }, let us assume that the first, third and fifth points were assigned to the first cluster. The centre of this cluster is (meanX,meanY):

meanX = (18+4+5)/3 and meanY = (84+28+9)/3

We do this for C2 as well. The only thing remaining is to repeat this process: calculate distance using new centres and assign point to C1 or C2; and update the centres again.

Now lets get to writing the class for this algorithm. First let us look at the data we need to store.

```
public class kmeans {
    int x[], y[];              // data points
    int num;                   // number of data points (supplied by the user)
    int k;                     // number of clusters (supplied by the user)
    double meanX[], meanY[]; // cluster centres
    double oldX[], oldY[];   // backup old cluster centres
    int cAssign[];             // cluster assignment
}
```

oldX and oldY are cluster centres from previous iteration. We need to store this because the algorithm terminates only when oldX=meanX and oldY=meanY for all the clusters. Here, meanX and meanY are cluster centres from the present iteration.

We also need to keep tract of which data point is assigned to which cluster. cAssign is created to do just that. For the first point, cAssign[0]=2 if it is assigned to the second cluster.

Let us now look at the methods we will need.

```
void randomMean() {
//Initialize meanX and meanY with random values between 0 and 1000 for all k centres
//Use the nextInt() method in the java.util.Random class
}

void randomData() {
//Initialize 'num' data points with random values between 0 and 1000
//Use the nextInt() method
}

void assignCluster() {
// Calculate the distance between the point and the cluster centre
// The Euclidean distance between the jth data point and ith cluster centre is:

    distance  = Math.sqrt(Math.pow(x[j]-meanX[i],2) + Math.pow(y[j]-meanY[i],2))

// Calculate distance for all k clusters and assign j to whichever i has the smallest
distance
// Assign this i to cAssign
}

void updateMeans() {
// Before updating the centres, backup meanX and meanY (copy to oldX and oldY)
// Calculate meanX and meanY for each cluster
}

void doClustering() {
//This is where you implement the clustering algorithm. Simple isn't it.
      randomMean();
      randomData();
      do {
            assignCluster();
            updateMeans();
         } while(isDifferent());
}

boolean isDifferent() {
//return true if meanX≠oldX or meanY≠oldY for one or more clusters
//Otherwise return false
}
```

Task A:

a. Complete the code by implementing the methods.
b. Change the condition to terminate the `do…while` loop in `doClustering()`. It should terminate either when `isDifferent()` returns false or if the iteration exceeds `MAXITER`. Set `MAXITER` as 20.
c. Run the program with values k=3 (three clusters) and num=100 (100 random data points). Explain the result: How many iterations did it take? How many points are in each cluster? What is the centroid of each cluster?

Task B:

a. Implement a function that calculates the Residual Sum of Squares (RSS).
b. Instead of Euclidean distance implement the Manhattan distance and redo the experiment. Which gives better results?
c. Assume that the data point has 5 values (size of the vector) and not just 2. Define a 2D array to store these values and make changes to the code wherever necessary.

Notes:

1. Array initialization

You declare and initialize arrays like:

```
int x[], points[][];     // Tells the compiler that x holds an array of Ints
x = new int[10];         // Actually assigns memory to x to store 10 Ints
points = new int[3][10]; // 2D array is similar
```

2. Random number generation

   To generate random numbers, we have to create an object of type `Random`. We then call the `nextInt()` method to generate a random integer. For example:

```
import java.util.Random;

Random rand = new Random();
int first = rand.nextInt(100) + 1; //Generate a random number between [1,100]
int second = rand.nextInt(100) + 1; //We add 1 in case we want to exclude zero
```

3. Math functions

   You will need to calculate square and square-root of a number. They are available as part of the class `Math`. We can use it as:

```
int x=9;
int square = Math.pow(x,2);
int sqroot = Math.sqrt(x);
```