COMPM012 - Coursework 3

Esben A. Sørig

December 1, 2014

1 Practical

1.1 k-means implementation

```
function [clusterings, centers] = mykmeans(X, k)
    % Randomly initialize centers of clusters
    c = datasample(X, k, 'Replace', false);
    r = repmat(0, size(X, 1), k);
    oldr = 1; % something that is not equal to r initially

    dist = @(x, y) norm(x-y);

    % Loop as long as the clustering is changing
    while ~isequal(r, oldr)
        oldr = r;

        % Assign points to clusters
        for i = 1:size(X,1)
            cluster = 1;
            for j = 1:k
                if dist(X(i, :), c(j, :)) <  dist(X(i, :), c(
    cluster, :))
                    cluster = j;
                end
            end

            r(i, :) = [repmat(0, 1, cluster-1) 1 repmat(0, 1, k-
    cluster)];
        end

        % Update center positions
        for i = 1:k
            npoints = 0;
            c(i, :) = 0;
            for j = 1:size(r, 1)
```

```
                c(i, :) = c(i, :) + r(j, i)*X(j, :);
                npoints = npoints + r(j, i);
            end
            c(i, :) = c(i, :)/npoints;
        end
    end

    centers = c;
    % Output formatting (vector with cluster index for each row
    in X)
    clustering = repmat(0, size(X,1), 1);
    for i = 1:size(X,1)
        for j = 1:k
            if r(i, j) == 1
                clustering(i) = j;
                break;
            end
        end
    end

    clusterings = clustering;
end
```

1.2 k-means test on data generated from three gaussians

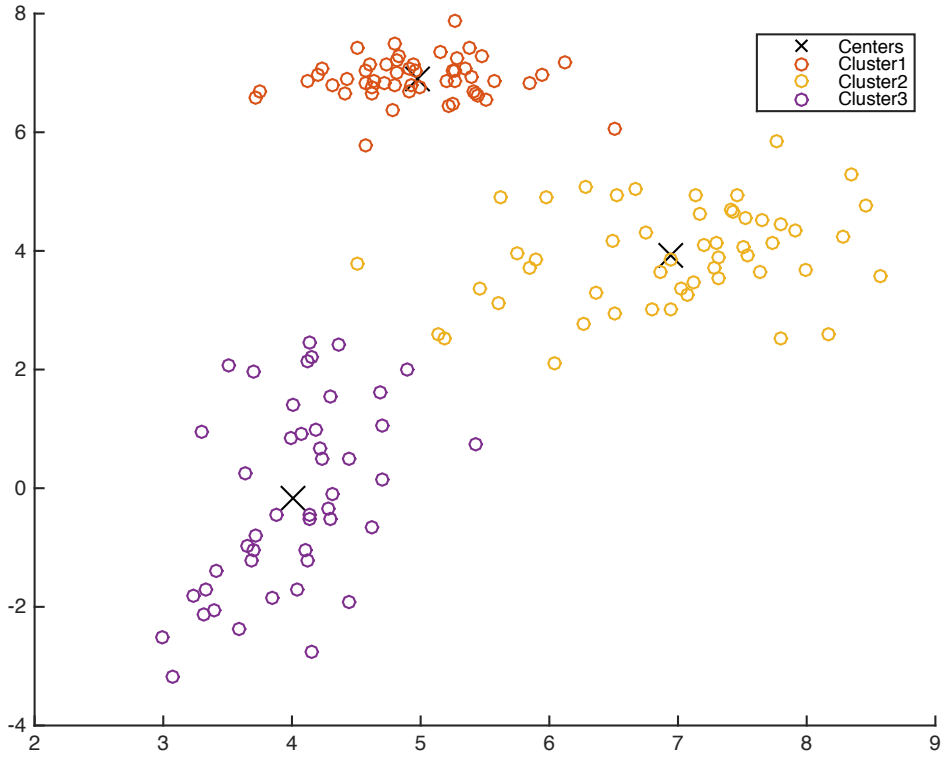
Firstly, we generate the data and run the k-means algorithm on it.

```
% Generate data
data = genData2;

% Put data for each cluster in a seperate list
cluster1 = [];
cluster2 = [];
cluster3 = [];
[clusterings, centers] = mykmeans(data, 3);

for i = 1:size(data, 1)
    if clusterings(i) == 1
        cluster1 = [cluster1; data(i, :)];
    end
    if clusterings(i) == 2
        cluster2 = [cluster2; data(i, :)];
    end
    if clusterings(i) == 3
        cluster3 = [cluster3; data(i, :)];
    end
end
```

We can now plot the clusters the algorithm has found

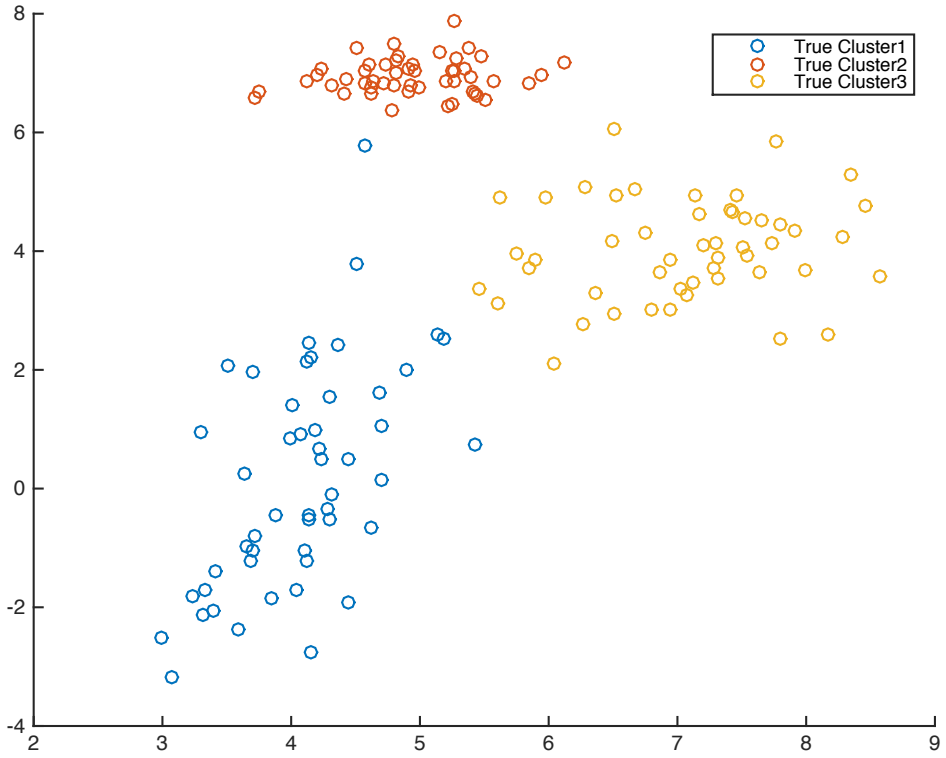
```
figure
```

```
hold on

scatter(centers(:, 1), centers(:, 2), 200, 'xblack');
scatter(cluster1(:, 1), cluster1(:, 2));
scatter(cluster2(:, 1), cluster2(:, 2));
scatter(cluster3(:, 1), cluster3(:, 2));
legend('Centers', 'Cluster1', 'Cluster2', 'Cluster3');
hold off
```



Comparing this to the true classifications

```
figure
hold on
scatter(data(1:50, 1), data(1:50, 2));
scatter(data(51:100, 1), data(51:100, 2));
scatter(data(101:150, 1), data(101:150, 2));
legend('True Cluster1', 'True Cluster2', 'True Cluster3');
hold off
```



We clearly get an intuition of how the k-means algorithm is performing on the data. Please see appendix 4.1 for the code that creates the animation of the convergence to a solution for this dataset.

We measure the mean error and standard deviation of the error of the clustering over a 100 runs of the algorithm.

```
%% Error measurements
errors = [];
for j = 1:100
    [clusterings, centers] = mykmeans(data, 3);

    % keep track of the classifications of each true cluster
    firstcluster = [0,0,0];
    secondcluster = [0,0,0];
    thirdcluster = [0,0,0];
    for i = 1:50
        firstcluster(clusterings(i)) = 1 + firstcluster(
    clusterings(i));
    end
    for i = 51:100
```

```
        secondcluster(clusterings(i)) = 1 + secondcluster(
    clusterings(i));
    end
    for i = 101:150
        thirdcluster(clusterings(i)) = 1 + thirdcluster(
    clusterings(i));
    end
    % We assume that the mode of the classifications of a true
    cluster is the class of the true cluster. Therefore, the
    error of a cluster is the frequency of other classifications
    appearing for that cluster.
    firstcluster = sort(firstcluster);
    secondcluster = sort(secondcluster);
    thirdcluster = sort(thirdcluster);
    misclassifications = firstcluster(1) + firstcluster(2) +
    secondcluster(1) + secondcluster(2) + thirdcluster(1) +
    thirdcluster(2);
    errors = [errors misclassifications/150];
end

meanerror = mean(errors)
stddeviationerror = std(errors)
```

Which returns:

```
meanerror =
    0.0333
stddeviationerror =
   2.0922e-17
```

1.3 Iris data

Performing a similar experiment and error measure on the iris dataset is done in a similar way

```
iris = readtable('iris.dat');
params = table2array(iris(:, 1:4));
classifications = table2array(iris(:, 5));
classes = unique(classifications);
class1 = find(strcmp(classifications, classes(1)));
class2 = find(strcmp(classifications, classes(2)));
class3 = find(strcmp(classifications, classes(3)));

errors = [];
for n = 1:100
    clusters = kmeans(params, 3);
    clusters1 = clusters(class1);
    clusters2 = clusters(class2);
    clusters3 = clusters(class3);
```

```
15
16      label1 = mode(clusters1);
17      label2 = mode(clusters2);
18      label3 = mode(clusters3);
19
20      errcount = 0;
21      for i = 1:50
22          if clusters1(i) ~= label1
23              errcount = errcount + 1;
24          end
25          if clusters2(i) ~= label2
26              errcount = errcount + 1;
27          end
28          if clusters3(i) ~= label3
29              errcount = errcount + 1;
30          end
31      end
32
33      e = errcount/size(params, 1);
34      errors = [errors e];
35  end
36
37  mean = mean(errors)
38  stddeviation = std(errors)
```

Which returns

```
1  mean =
2      0.1096
3  stddeviation =
4      0.0117
```

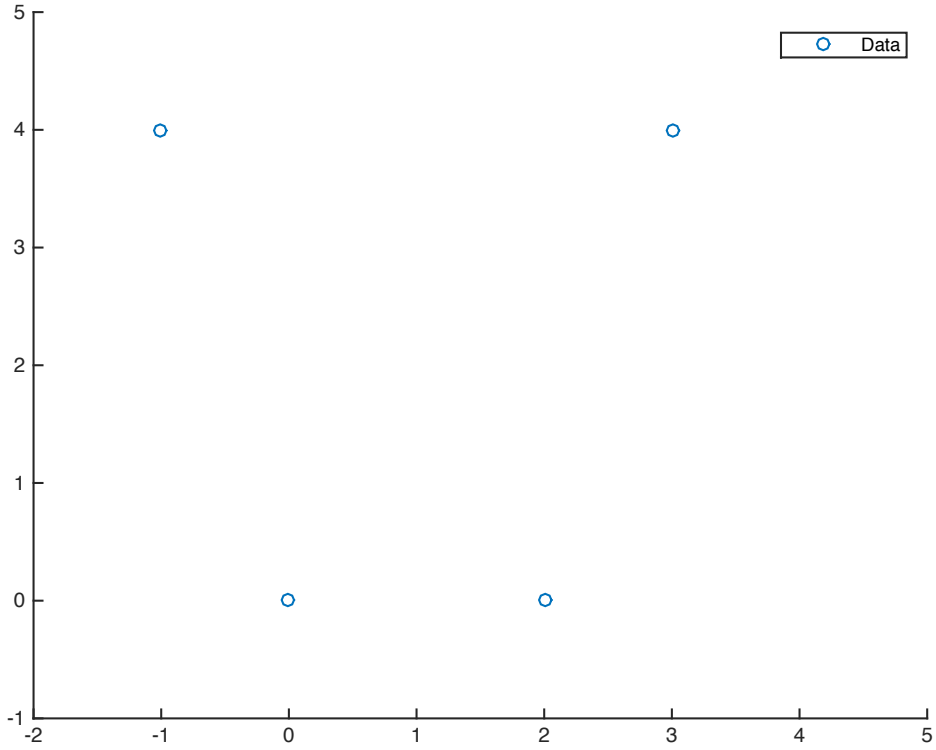
2 Questions

2.1 Dataset with local minima

We can easily construct a dataset on which our k-means algorithm has local minima. An example of such a dataset is the set $\{(0,0),(2,0),(-1,4),(3,4)\}$. This is clear if we plot the data.

```
1  X = [0, 0; 2,0; -1 4; 3 4];
2
3  hold on
4  axis([-2 5 -1 5]);
5  scatter(X(:, 1), X(:, 2));
```

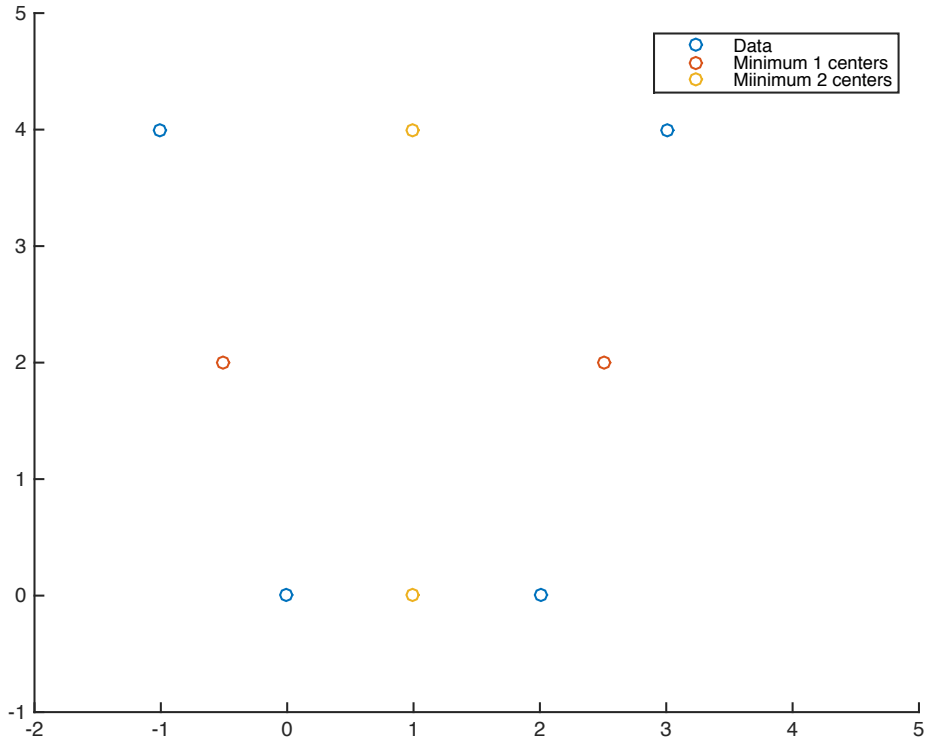
Which gives the plot:



Running k-means on this dataset will converge in one of two local minima. We run the algorithm twice and plot the centers of the clusters: (in order to get two different results, this code may have to be run several times)

```
[clustering1, centers1] = mykmeans(X, 2);
[clustering2, centers2] = mykmeans(X, 2);
legend('Data');

scatter(centers1(:, 1), centers1(:, 2));
scatter(centers2(:, 1), centers2(:, 2));

legend('Data', 'Minimum 1 centers', 'Miinimum 2 centers');
hold off
```

Which produces:



Here we clearly see the two local minima of the convergence on the four data points.

2.2 Argument that the centroid is the minimizer of the sum of squared distances

We can minimize the summed squared error

$$SSE = \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \mathbf{c}_i||^2 = \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} (\mathbf{x} - \mathbf{c}_i)^2 \tag{1}$$

for the k^{th} centroid by setting the derivative with respect to the k^{th} centroid equal to zero.

$$\begin{aligned}
\frac{\delta}{\delta \mathbf{c}_k} \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} (\mathbf{x} - \mathbf{c}_i)^2 &= \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} \frac{\delta}{\delta \mathbf{c}_k} (\mathbf{x} - \mathbf{c}_i)^2 \\
&= \sum_{\mathbf{x} \in C_k} 2(\mathbf{x} - \mathbf{c}_k) = 0 \\
\implies \sum_{\mathbf{x} \in C_k} \mathbf{c}_k &= \sum_{\mathbf{x} \in C_k} \mathbf{x} \\
\implies \mathbf{c}_k &= \frac{1}{\sum_{\mathbf{x} \in C_k} 1} \sum_{\mathbf{x} \in C_k} \mathbf{x}
\end{aligned} \tag{2}$$

We see that the minimizer of $\mathbf{c}_k$ is equivalent to the centroid of the cluster.

2.3 Proof of convergence in finite amount of steps

We know that the k-means algorithm converges but we do not know if it does so in finitely or infinitely many steps. Here we show that the k-means algorithm does indeed converge in finitely many steps.

Proof: The k-means algorithm stops once there is no longer any improvement in the clustering. It will therefore never reach the same clustering twice. Each possible k-clustering of the input consists of k subsets. The number of subsets of the input is a finite number and the number of k subsets of the input is bounded above by this. Consequently, there is a finite number of possible clusterings. Hence the k-means algorithm will terminate after a finite number of iterations.

3 Extension

3.1 k-means segmentation

The objective of the algorithm is to find the segmentation $\{i_1, ..., i_{k-1}\}$ of a sequence of points $\{\mathbf{x}_1, ..., \mathbf{x}_l\} \in \mathbb{R}^n$ that is the global minimum of the following optimisation problem:

$$\underset{i_1,...,i_{k-1};\mathbf{c}_1,...,\mathbf{c}_k}{\operatorname{argmin}} \sum_{j=1}^{k} \sum_{p=i_{j-1}+1}^{i_j} ||\mathbf{x}_p - \mathbf{c}_j||^2 \tag{3}$$

The algorithm must run in polynomial time in l, n, and k. We start by defining the error function as it will be useful when searching for minima of in the error.

```
function e = error(delimiters, X)
    e = 0;
    bounddelimiters = [0 delimiters size(X, 1)];
    for j = 2:size(bounddelimiters, 2)
        indeces = (1+bounddelimiters(j-1)):bounddelimiters(j);
        segment = X(indeces, :);
        centroid = sum(segment)/size(segment, 1);
        for k = 1:size(segment, 1)
            e = e + norm(segment(k, :)-centroid)^2;
        end
    end
end
```

We can now proceed to designing the segmentation algorithm. Let $\mathbf{i}^*_{n,k}$ be the set of segment delimiters $\{i^*_1, ..., i^*_{k-1}\}$ that minimises the error of segmenting the points $x_1, ..., x_n$ into k segments. The key observation for construction of a polynomial time algorithm is that $\mathbf{i}^*_{n,k}$ can be described as

$$\begin{aligned} \mathbf{i}^*_{n,k} &= \mathbf{i}^*_{j,k-1} \cup \{j\} \text{ where} \\ j &= \underset{1 \leq j < n}{\operatorname{argmin}}(E(\mathbf{i}^*_{j,k-1} \cup \{j\})) \end{aligned} \tag{4}$$

Here E denotes the error function of the segmentation produced a set of delimiters. From this observation we can construct a dynamic programming algorithm that builds a matrix of the optimal segmentations and a matrix of their errors. When the algorithm has reached the optimal segmentation $\mathbf{i}^*_{n,k}$ we have found the solution. The Matlab implementation of the algorithm is listed here:

```
function delimiters = segmentation(X, k)
    % E(i, k) represents the minimum error of segmenting
    % X(1:i, :) into k segments.
    E = [];

    % D(i, j) stores the jth delimiter that segments X(1:i, :)
    % into the optimal j+1 segments.
    D = [];

    % We initialise E's first column (error of a single
    % segment over the points X(1:i, :))
    for i = 1:size(X, 1)
        E(i,1) = error([], X(1:i, :));
    end
```

```
15
16      % We then proceed to filling out each entry in E until
17      % we have reached our objective (the minimum k
18      % segmentation of all points in X).
19      for i = 1:size(X, 1)
20          for l = 2:min(i, k)
21              % Find min error of segmenting x_1, ..., x_i
22              % into l segments
23              minerror = Inf;
24              for j = 1:(i-1)
25                  if j >= l
26                      e = E(j, l-1) + error([], X((j+1):i, :));
27                      if e < minerror
28                          minerror = e;
29                          D(i, l) = j; % Store delimiter
30                      end
31                  end
32              end
33              E(i, l) = minerror;
34          end
35      end
36
37      % We retrieve the optimal delimiters from D
38      delimiters = [];
39      delimiter = size(X, 1);
40      for i = k:-1:2
41         delimiter = D(delimiter, i);
42         delimiters = [delimiters delimiter];
43      end
44      delimiters = sort(delimiters)
45  end
```

Informal argument of time complexity: The running time is polynomial in l, n, and k since the algorithm simply consists of nested loops running over l and k. In the inner most loop, the error function is called, which is similarly polynomial in complexity in k and l but also n (since we sum over the points). The complexity of the algorithm can therefore be bounded above by a constant multiple of some polynomial in n, l, and k.

3.2 (p, k)-means

We derive the minimiser of the objective for each C_i in the same way as with the squared distance. We let the derivative with respect to $\mathbf{c}_k$ be equal to zero and solve for $\mathbf{c}_k$. We start by finding the derivative with respect to $\mathbf{c}_k$.

$$\frac{\delta}{\delta \mathbf{c}_k} \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} d_p(\mathbf{c}_i, \mathbf{x}) = \sum_{\mathbf{x} \in C_k} \frac{\delta}{\delta \mathbf{c}_k} d_p(\mathbf{c}_k, \mathbf{x}) \tag{5}$$

We assume that the components of the inputs are nonnegative and obtain

$$\begin{aligned} \frac{\delta}{\delta \mathbf{c}_k} d_p(\mathbf{c}_k, \mathbf{x}) &= \frac{\delta}{\delta \mathbf{c}_k} (\sum_{i=1}^{n} \mathbf{c}_{k,i}^p - \mathbf{x}_i^p - p(\mathbf{c}_{k,i} - \mathbf{x}_i)\mathbf{x}_i^{p-1}) \\ &= p\mathbf{c}_k^{p-1} - p\mathbf{x}^{p-1} \end{aligned} \tag{6}$$

Where $\mathbf{x}^n$ denotes that each component of $\mathbf{x}$ is raised to the nth power. Inserting back in (5) and setting this equal to zero, we can now solve for $\mathbf{c}_k$

$$\begin{aligned} &\sum_{\mathbf{x} \in C_k} (p\mathbf{c}_k^{p-1} - p\mathbf{x}^{p-1}) = 0 \\ &\implies \sum_{\mathbf{x} \in C_k} \mathbf{c}_k^{p-1} = \sum_{\mathbf{x} \in C_k} \mathbf{x}^{p-1} \\ &\implies \mathbf{c}_k = \sqrt[p-1]{\frac{\sum_{\mathbf{x} \in C_k} \mathbf{x}^{p-1}}{|C_k|}} \end{aligned} \tag{7}$$

Observe that this result is consistent with our result for the euclidean distance. We can now design the (p, k)-means algorithm.

1. **for** $i = 1, ..., k$ **do**

$$C_i = \{\mathbf{x} \in x \mid i = \underset{1 \le j \le k}{\operatorname{argmin}} d_p(\mathbf{c}_j, \mathbf{x})\}$$

2. **for** $j = 1, ..., k$ **do**

$$\mathbf{c}_j = \sqrt[p-1]{\frac{\sum_{i=1}^{l} r_{ij} \mathbf{x}_i^{p-1}}{\sum_{i=1}^{l} r_{ij}}}$$

3. **While** not converged **go to** step 1.

Again, $\mathbf{x}^n$ denotes that each component of $\mathbf{x}$ is raised to the nth power and $\sqrt[n]{\mathbf{x}}$ denotes that we take the nth root of each component of x. The argument of convergence for this algorithm is similar to the argument for the standard k-means algorithm. That is, the objective decreases in step 1 and 2 and the objective is bounded below. Hence the algorithm converges.

4 Appendix

4.1 Movie generation code

```
%% Movie
X = data;
k = 3;
% Randomly initialize centers of clusters
c = datasample(X, k, 'Replace', false);
r = repmat(0, size(X, 1), k);
oldr = 1; % something that is not equal to r initially
movie = [];

dist = @(x, y) norm(x-y);

% Loop as long as the clustering is changing
while ~isequal(r, oldr)
    oldr = r;

    % Assign points to clusters
    for i = 1:size(X,1)
        cluster = 1;
        for j = 1:k
            if dist(X(i, :), c(j, :)) <  dist(X(i, :), c(cluster
    , :))
                cluster = j;
            end
        end

        r(i, :) = [repmat(0, 1, cluster-1) 1 repmat(0, 1, k-
    cluster)];
    end

    %MOVIE GENERATION
    % Split the data into the three clusters
    cluster1 = [];
    cluster2 = [];
    cluster3 = [];

    for i = 1:size(X, 1)
        if r(i,1) == 1
            cluster1 = [cluster1; X(i, :)];
        end
        if r(i,2) == 1
            cluster2 = [cluster2; X(i, :)];
        end
        if r(i,3) == 1
            cluster3 = [cluster3; X(i, :)];
```

```
        end
    end

    % Plot the centers and the three clusters
    hold on
    scatter(c(:, 1), c(:, 2), 200, 'xblack');
    if size(cluster1, 1) ~= 0
        scatter(cluster1(:, 1), cluster1(:, 2));
    end
    if size(cluster2, 1) ~= 0
        scatter(cluster2(:, 1), cluster2(:, 2));
    end
    if size(cluster3, 1) ~= 0
        scatter(cluster3(:, 1), cluster3(:, 2));
    end
    legend('Centers', 'Cluster1', 'Cluster2', 'Cluster3');
    hold off
    movie = [movie getframe];
    clf;
    % MOVIE GENERATION OVER

    % Update center positions
    for i = 1:k
        npoints = 0;
        c(i, :) = 0;
        for j = 1:size(r, 1)
            c(i, :) = c(i, :) + r(j, i)*X(j, :);
            npoints = npoints + r(j, i);
        end
        c(i, :) = c(i, :)/npoints;
    end
end
movie2avi(movie, 'k-means.avi', 'fps', 1);
```