



Coding Assignment 3
Computational Intelligence
Dr. S. Hajipour

Mohammad Hossein Shafizadegan
99104781

January 5, 2024

Contents

1	Question 1	2
1.1	Data visualization	2
1.2	Train and validation set	2
1.3	Design RBF network with one output neuron	2
1.4	Optimal parameters	3
1.5	RBF network with two output neurons	4
2	Question 2	5
2.1	Part 1	5
2.2	Part 2	6
2.3	MATLAB kmeans	6
2.4	Hierarchical clustering	7
3	Question 3	8
3.1	Genetic Algorithm	8
3.2	PSO	9
3.3	ACO	10

1 Question 1

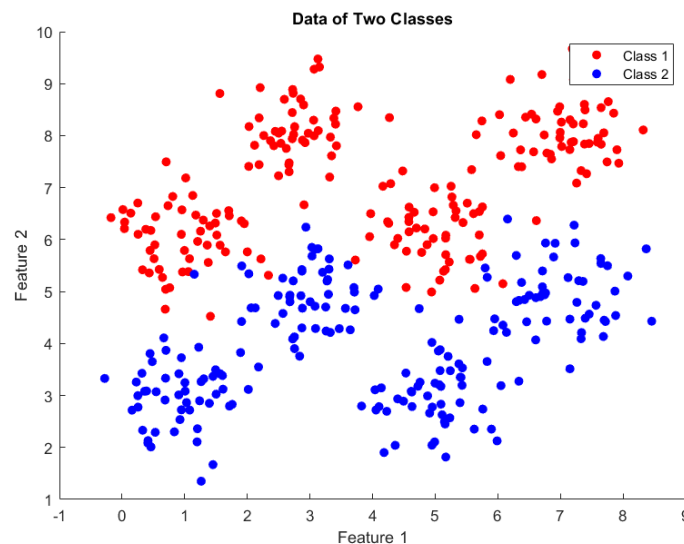
1.1 Data visualization

After loading the provided data, we separate data corresponding to each class regarding the labels as follows:

```
1 % Separate the data into two classes based on the labels
2 class1Data = TrainingData(:, TrainingLabels == 0);
3 class2Data = TrainingData(:, TrainingLabels == 1);
```

Then we visualize the data of each class using built-in function 'scatter()':

```
1 % Plot the data of two classes
2 figure;
3 scatter(class1Data(1, :), class1Data(2, :), 'ro', 'filled');
4 hold on; % Hold on to the current figure
5 scatter(class2Data(1, :), class2Data(2, :), 'bo', 'filled');
```



1.2 Train and validation set

We randomly separate our data into train and validation set with proportion of 70% train and 30% validation. Here is the code used for this section:

```
1 % Split into training and validation sets (70% training, 30% validation)
2 [trainInd, valInd, ~] = dividerand(size(TrainingData, 2), 0.7, 0.3, 0);
3 trainX = TrainingData(:, trainInd);
4 trainY = TrainingLabels(:, trainInd);
5 valX = TrainingData(:, valInd);
6 valY = TrainingLabels(:, valInd);
```

1.3 Design RBF network with one output neuron

In order to implement the RBF network, we are going to use the 'newrb()' which is a MATLAB built-in function. In this function the radius of the radial activation function used in the network can be set by calibrating the 'spread'

input argument.

```

1 % Define the network spec
2 hiddenNeurons = 10;
3 spread = 1;
4
5 % Design the RBF network
6 net_1 = newrb(trainX, trainY, 0, spread, hiddenNeurons);

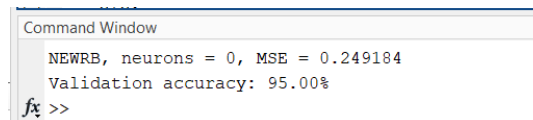
```

Then we utilize our network to predict the outputs for the validation set and then calculate the accuracy using the following code:

```

1 val_pred_1 = net_1(valX);
2
3 % Calculate the classification accuracy
4 predictedLabels = round(val_pred_1);
5 accuracy = sum(predictedLabels == valY) / numel(valY);
6 fprintf('Validation accuracy: %.2f%%\n', accuracy * 100);

```



```

Command Window
NEWRB, neurons = 0, MSE = 0.249184
Validation accuracy: 95.00%
fx >>

```

It can be seen that we have achieved a quite good performance with this value of parameters.

1.4 Optimal parameters

Here we aim to find which value for number of hidden neurons and the radius (σ) of the activation function, lead to a better outcome. We will check different values of these parameters and plot the accuracy regarding to these parameters. Here is the code:

```

1 nhidden = 1:20;
2 sigma = 0.1:0.1:10;
3 [X, Y] = meshgrid(nhidden, sigma);
4
5 acc = zeros(length(nhidden), length(sigma));
6 for i=1:length(nhidden)
7     for j=1:length(sigma)
8         acc(i,j) = pred_RBF_1out(nhidden(i), sigma(j), valX, valY, trainX, trainY);
9     end
10 end

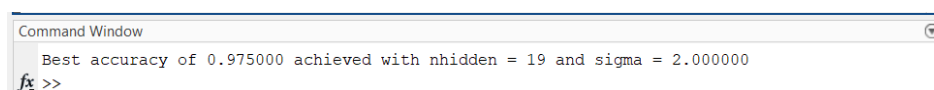
```

Now we print the spec resulted into the the best accuracy using the following code:

```

1 [~, idx] = max(acc(:));
2 [row, col] = ind2sub(size(acc), idx);
3
4 fprintf('Best accuracy of %f achieved with nhidden = %d and sigma = %f \n', ...
5         acc(row, col), nhidden(row), sigma(col));

```



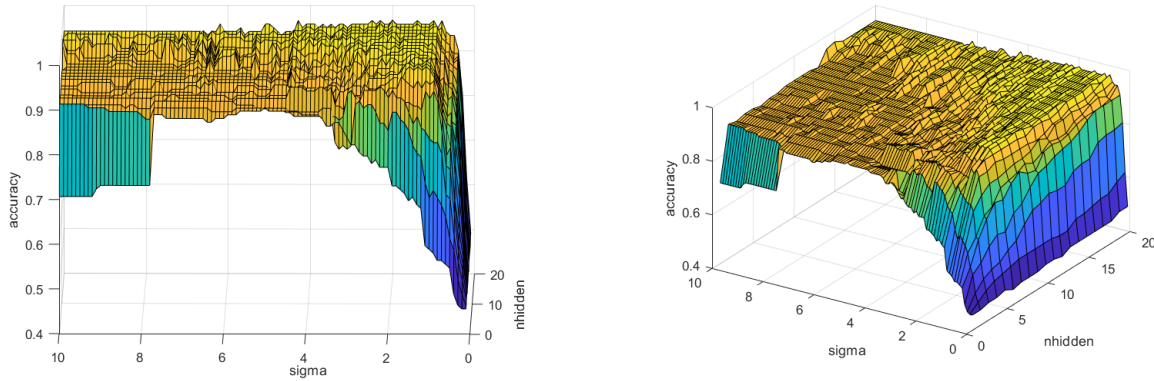
```

Command Window
Best accuracy of 0.975000 achieved with nhidden = 19 and sigma = 2.000000
fx >>

```

It can be seen that the best choice for number of hidden neurons is 19 and the best choice for radius is 2.

Now we plot the 3D curve, demonstrating the value of accuracy for different values of radius and number of hidden neurons.



1.5 RBF network with two output neurons

In order to design our network, we have to perform some modifications. The network will have two output neurons. We assume that the first output neuron indicates the probability of belonging to the first class and in a same way, the second output neuron will indicate the probability of a sample to belong to the second class (label).

Regarding this, we will modify the output for train and validation set as follows:

```
1 train_Y_2 = [1-trainY; trainY];
2 val_Y_2 = [1-valY; valY];
```

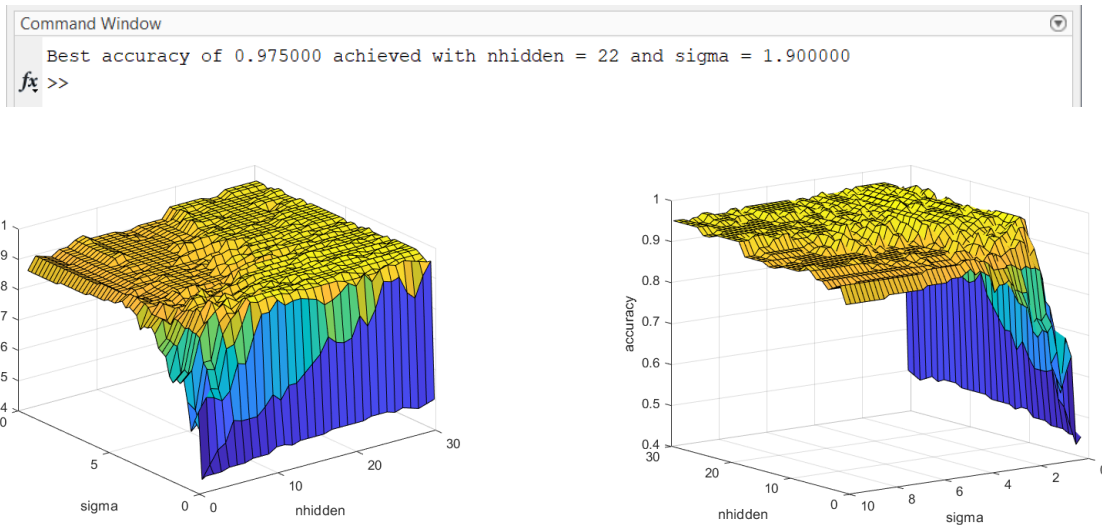
Finally in order to decide the label of a sample, we assign them the class with higher probability. This can be done as follows:

```
1 hiddenNeurons = 20;
2 % Design the RBF network
3 net_2 = newrb(trainX, train_Y_2, 0, 1, hiddenNeurons);
4
5 val_Y_2 = [1-valY; valY];
6 val_pred_2 = sim(net_2, valX);
7
8 % Calculate the classification accuracy
9 [~, predictedLabels] = max(val_pred_2);
10 [~, actualLabels] = max(val_Y_2);
```

Now we assess our model:

```
Command Window
NEWRB, neurons = 0, MSE = 0.249184
Validation accuracy: 95.83%
fx >>
```

In order to observe how number of hidden neurons and value of radius affect this network, we pursue the same approach as we did for the previous network:



2 Question 2

2.1 Part 1

We have developed a function called 'my_kmeans'. It takes input data, the number of clusters (k), and initial centroids. The code iteratively assigns data points to the nearest centroids and updates the centroid positions until convergence is reached. It does this by calculating distances between data points and centroids, assigning labels to each data point based on the nearest centroid, and updating the centroids based on the mean of the data points assigned to each cluster. The process continues until the centroids no longer change.

The function then returns the resulting centroids and labels. Overall, it's a simple implementation of k-means clustering, performing iterative assignments and updates to find cluster centers and assign data points to clusters.

```

1  % Initialize labels and previous centroids
2  labels = zeros(size(data, 2), 1);
3  prevCentroids = zeros(size(centroids));
4
5  % Iterate until convergence
6  while ~isequal(centroids, prevCentroids)
7
8      % Assign each data point to the nearest centroid
9      for i = 1:size(data, 2)
10         distances = sum((data(:, i) - centroids).^2, 1); % Adjusted indexing
11         [~, index] = min(distances);
12         labels(i) = index;
13     end
14
15     % Update the centroids
16     prevCentroids = centroids;
17     for j = 1:k
18         clusterPoints = data(:, labels == j);
19         centroids(:, j) = mean(clusterPoints, 2); % Adjusted mean calculation
20     end
21 end

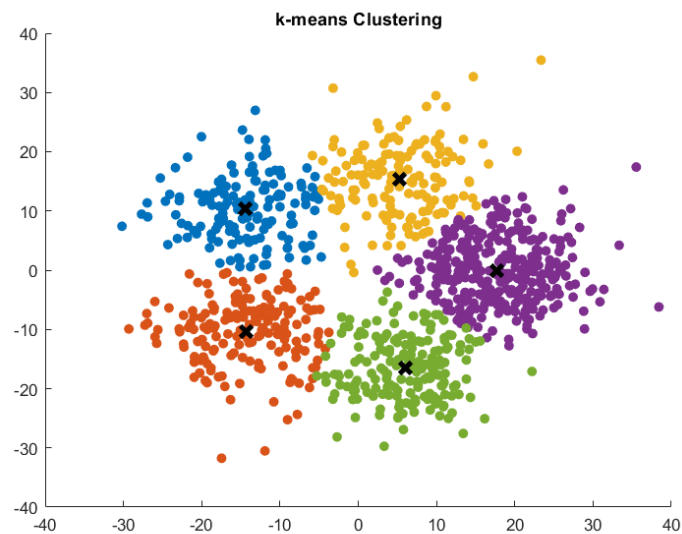
```

Now we assess the performance of our function for dividing data into 5 clusters. Using the following code we will visualize the results:

```

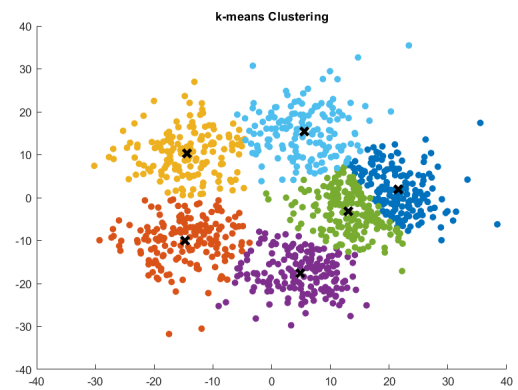
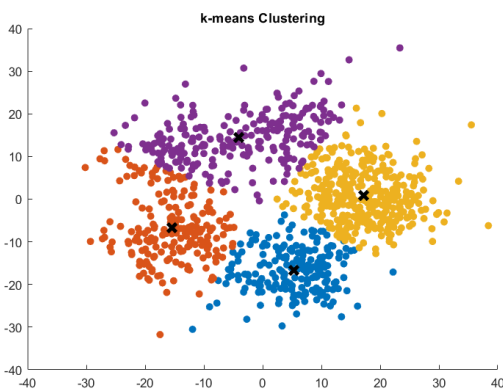
1  hold on;
2  colors = lines(k);
3  for i = 1:k
4      clusterPoints = DataNew(:, labels == i);
5      scatter(clusterPoints(1, :), clusterPoints(2, :), [], colors(i, :), 'filled');
6  end
7  scatter(centroids(1, :), centroids(2, :), 100, 'k', 'filled');
8  hold off;
9  title('k-means Clustering');

```



2.2 Part 2

Here we check the results for 4 and 6 clusters below:



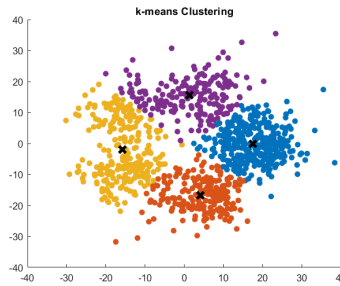
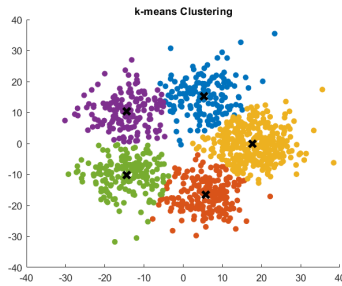
2.3 MATLAB kmeans

Now we use the MATLAB builtin function as follows:

```

1 k=6;
2 [labels, centroids] = kmeans(DataNew', k);
3 plot_clusters(k, DataNew, centroids', labels)

```



2.4 Hierarchical clustering

Hierarchical clustering is a machine learning algorithm used to group unlabeled datasets into a tree-shaped structure called a dendrogram. It is an unsupervised learning method that starts by treating each data point as a separate cluster and then iteratively combines the closest clusters until a stopping criterion is reached. The algorithm can be divided into two approaches: Agglomerative and Divisive. The Agglomerative approach is a bottom-up approach, in which the algorithm starts with taking all data points as single clusters and merging them until one cluster is left. The Divisive algorithm is the reverse of the Agglomerative algorithm as it is a top-down approach.

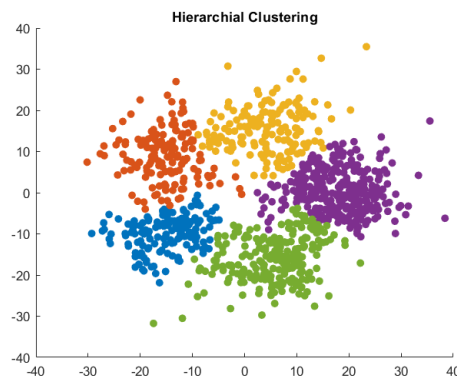
Hierarchical clustering is useful when we don't have knowledge about the predefined number of clusters. It is also helpful when we want to create clusters of different sizes. We have implemented this method in MATLAB using the following code for 5 clusters:

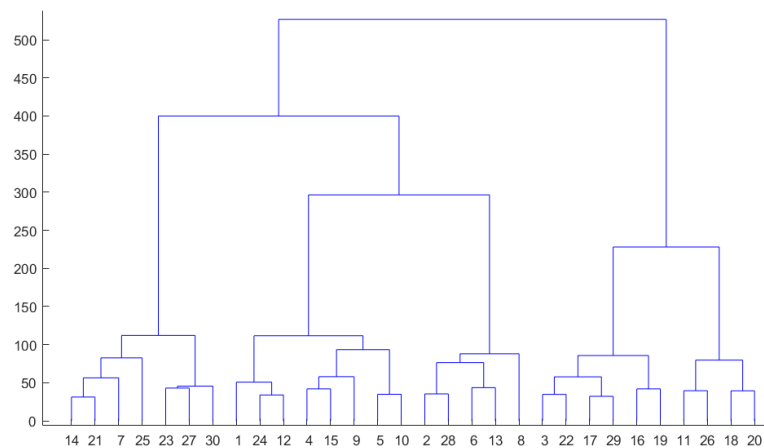
```

1 %Perform hierarchical clustering
2 linkageMatrix = linkage(data, 'ward', 'euclidean');
3
4 % Plot the dendrogram
5 dendrogram(linkageMatrix);
6
7 k = 5; % Specify the desired number of clusters
8 clusterLabels = cluster(linkageMatrix, 'maxclust', k);
9 plot_clusters_2(k, data', clusterLabels);

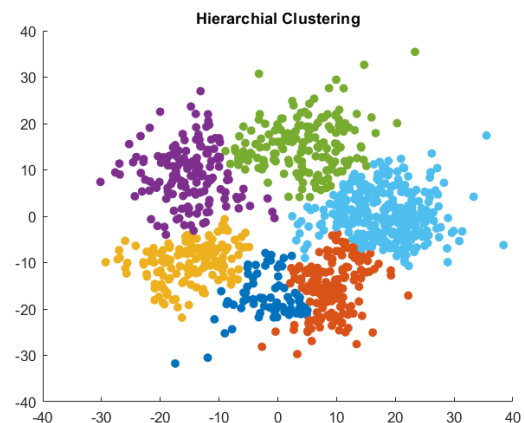
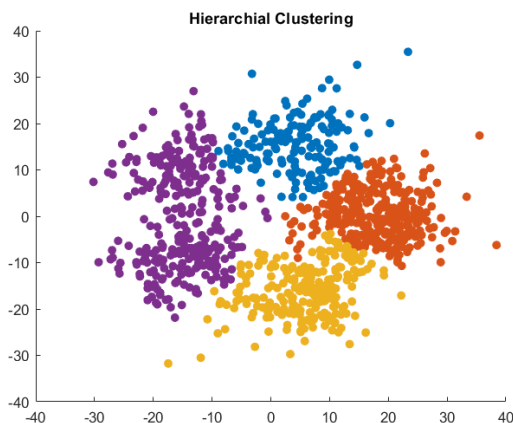
```

Here are the results:





Also we will generate the results for 4 and 6 clusters:



3 Question 3

3.1 Genetic Algorithm

First we simply load the data. Then we define the lower and upper bound of the centroid of the clusters regarding the minimum and maximum value of data. After that using MATLAB built-in function called 'optimoptions()', we define the genetic algorithm options as follows:

```
1 % Define the genetic algorithm options
2 options = optimoptions('ga', 'Display', 'iter', 'MaxGenerations', 100, 'PopulationSize', 50);
```

The next step is to define the fitness function regarding the problem of kmeans. The following function is developed to do so:

```
1 function fitness = kmeansFitness(data, centroids, K)
2 % Reshape the centroids
3 centroids = reshape(centroids, K, []);
4
5 % Compute distances from each data point to each centroid
```



```

6     distances = pdist2(data, centroids);
7
8     % Find the closest centroid for each data point
9     [~, idx] = min(distances, [], 2);
10
11    % Compute the sum of squared distances
12    fitness = sum(sum((data - centroids(idx, :)).^2));
13    end

```

Then we run the genetic algorithm using the builtin function 'ga()' as follows:

```

1    % Run the genetic algorithm
2    centroidsGA = ga(fitnessFunction, size(data, 2) * K, [], [], [], [], lb, ub, [], options);

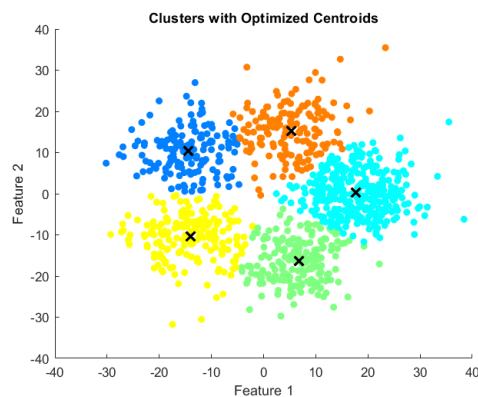
```

Now we have the final and resulting positions of the centroids. In order to visualize the results we have to find the nearest data points to these clusters and plot the results. The following code will do this:

```

1    % Compute distances from each data point to each centroid
2    distancesGA = pdist2(data, centroidsGA);
3
4    % Find the closest centroid for each data point
5    [~, idxGA] = min(distancesGA, [], 2);
6
7    % Visualize the results
8    figure;
9    scatter(data(:, 1), data(:, 2), 30, idxGA, 'filled');
10   hold on;
11   scatter(centroidsGA(:, 1), centroidsGA(:, 2), 100, 'k', 'x', 'LineWidth', 2);
12   title('Clusters with Optimized Centroids');
13   xlabel('Feature 1');
14   ylabel('Feature 2');
15   colormap(gca, jet(K));

```



3.2 PSO

In this section, we first sets up the k-means parameters, including the number of clusters and the data to be clustered.

```

1    % Set up PSO parameters
2    options = optimoptions('particleswarm', 'SwarmSize', 50, 'MaxIterations', 100);

```

In this algorithm, each particle consists of an array with $10 (2 \times k)$ elements, indicating the position of the desired 5 (k in general) centroids. Regarding this particle presentation we run the PSO algorithm using the following code:

```

1 % Run PSO to find cluster centroids
2 numVariables = k * size(data, 2);
3 lb = repelem(min(data), k);
4 ub = repelem(max(data), k);
5
6 [bestPositions, cost] = particleswarm(objectiveFunction, numVariables, lb, ub, options);

```

The final cluster centroids are obtained by reshaping the best positions, and the data points are assigned to the final centroids.

```

1 % Reshape the best positions to get the final cluster centroids
2 numElementsPerCluster = numVariables / k;
3 bestCentroids = reshape(bestPositions(1 : floor(numElementsPerCluster) * k), k, []);
4
5 % Assign data points to the final centroids
6 [~, clusterIndices] = pdist2(bestCentroids, data, 'euclidean', 'Smallest', 1);

```



3.3 ACO

First we set parameters such as the number of clusters, ants, and maximum iterations. Then we simply load our data.

We have developed the ‘antColonyClustering’ function which implements the ACO algorithm. Here are the things we have considered in developing this function:

- Initializes pheromones.
- Runs the main loop for a specified number of iterations.
- Constructs solutions for each ant using the ‘constructSolution’ function.
- Updates pheromones based on the solutions using the ‘updatePheromones’ function.
- Evaluates the solutions and finds the best solution with the lowest sum of squared distances.

Here is the code:

```

1 % Main loop
2 for iteration = 1:maxIterations
3     solutions = zeros(numAnts, size(data, 1));

```

```

4      % Ant solutions construction
5      for ant = 1:numAnts
6          solutions(ant, :) = constructSolution(pheromones);
7      end
8
9      % Pheromone update
10     pheromones = updatePheromones(pheromones, solutions, data);
11
12     % Find the best solution
13     [minDistance, minIndex] = min(evaluateSolutions(solutions, data));
14     bestSolution = solutions(minIndex, :);
15     bestCentroids = calculateCentroids(data, bestSolution, numClusters);
16
17 end
18

```

Using the ‘constructSolution’ function, we construct a solution for each data point by probabilistically assigning it to a cluster based on pheromones.

```

1      function solution = constructSolution(pheromones)
2          % Construct a solution using pheromones for k-means clustering
3
4          numDataPoints = size(pheromones, 1);
5
6          % Initialize probabilities for each data point to be assigned to each cluster
7          probabilities = pheromones;
8
9          % Normalize probabilities
10         probabilities = probabilities ./ sum(probabilities, 2);
11
12         % Assign each data point to a cluster based on probabilities
13         solution = zeros(1, numDataPoints);
14         for point = 1:numDataPoints
15             solution(point) = selectCluster(probabilities(point, :));
16         end
17     end

```

We evaluate the quality of solutions based on the sum of squared distances between data points and their assigned cluster centroids.

We have developed the ‘updatePheromones’ function to implement the updating procedure of the pheromones. This function:

- Updates pheromones based on the assigned clusters in each solution.
- Uses an evaporation rate to decay existing pheromones and a deposit rate to add new pheromones based on the inverse of the squared distance.

```

1      function newPheromones = updatePheromones(pheromones, solutions, data)
2          % Update pheromones based on the assigned clusters
3          evaporationRate = 0.05;
4          depositRate = 0.8;
5
6          % Evaporation
7          pheromones = (1 - evaporationRate) * pheromones;
8

```

```

9      for ant = 1:size(solutions, 1)
10         centroids = calculateCentroids(data, solutions(ant, :), max(solutions(ant, :)));
11
12         for point = 1:size(data, 1)
13             cluster = solutions(ant, point);
14             temp = min(norm(data(point, :) - centroids(cluster, :))^2);
15             pheromones(point, cluster) = pheromones(point, cluster) + 10 * depositRate / temp;
16         end
17     end
18
19     % Normalize pheromones
20     newPheromones = pheromones ./ sum(pheromones, 2);
21 end

```

Finally we assess our code for the provided data and visualize the results:

