UNIVERSIDADE DE AVEIRO
DEPARTAMENTO DE ELECTRÓNICA TELECOMUNICAÇÕES E INFORMÀTICA

Machine Learning (2016/17) – Lab work 7

Objectives: Implementation of K-means clustering algorithm and its application for image compression.

Files included in this lab work
ex7.m – the main Octave/MATLAB script
ex7data2.mat - Example Dataset for K-means
bird small.png - Example Image
drawLine.m - Draws a line over an existing figure
plotDataPoints.m - Initialization for K-means centroids
plotProgresskMeans.m - Plots each step of K-means as it proceeds
runkMeans.m - Runs the K-means algorithm
findClosestCentroids.m - Find closest centroids **(complete this function)**
computeCentroids.m - Compute centroid means **(complete this function)**
kMeansInitCentroids.m - Initialization for K-means centroids (**(complete this function)**

1. Implementing K-means Clustering
You will first start on an example 2D dataset that will help you gain an intuition of how the K-means algorithm works. After that, you will use the K-means algorithm for image compression by reducing the number of colors that occur in an image to only those that are most common in that image.

The K-means algorithm is a method to automatically cluster similar data examples together. Concretely, you are given a training set $\{x^{(1)}, ..., x^{(m)}\}$ (where $x^{(i)} \in R^n$), and want to group the data into a few cohesive "clusters". The intuition behind K-means is an iterative procedure that starts by guessing the initial centroids, and then refines this guess by repeatedly assigning examples to their closest centroids and then re-computing the centroids based on the assignments. The K-means algorithm is as follows:

```
% Initialize centroids
centroids = kMeansInitCentroids(X, K);
for iter = 1:iterations
    % Cluster assignment step: Assign each data point to the
    % closest centroid. idx(i) corresponds to c^(i), the index
    % of the centroid assigned to example i
    idx = findClosestCentroids(X, centroids);

    % Move centroid step: Compute means based on centroid
    % assignments
    centroids = computeMeans(X, idx, K);
end
```

The algorithm repeatedly carries out two steps: (i) Assigning each training example $x^{(i)}$ to its closest centroid, and (ii) Re-computing the mean of each centroid using the points assigned to it. The K-means algorithm will always converge to some final set of means for the centroids. Note that the converged solution may not always be ideal and depends on the initial setting of the centroids. Therefore, in practice the K-means algorithm is usually run a few times with different random initializations. One way to choose

between these different solutions from different random initializations is to choose the one with the lowest cost function value (distortion).

1.1 Finding closest centroids

In the "cluster assignment" phase of the K-means, the algorithm assigns every training example $x^{(i)}$ to its closest centroid, given the current positions of centroids. Specifically, for every example i we set

$$c^{(i)} := j \quad \text{that minimizes} \quad ||x^{(i)} - \mu_j||^2$$

where $c^{(i)}$ is the index of the centroid that is closest to $x^{(i)}$, and μ_j is the position of the j'th centroid. Note that $c^{(i)}$ corresponds to *idx(i)* in the code.

Complete the code in *findClosestCentroids.m*. This function takes the data matrix X and the locations of all centroids and should output a one-dimensional array idx that holds the index (a value in {1, ..., K}, where K is total number of centroids) of the closest centroid to every training example. You can implement this using a loop over every training example and every centroid. After you complete the code in *findClosestCentroids.m*, the script will run the code and you should see the output [1 3 2] corresponding to the centroid assignments for the first 3 examples.

1.2 Computing centroid means

Given assignments of every point to a centroid, the second phase of the algorithm re-computes, for each centroid, the mean of the points that were assigned to it. Specifically, for every centroid k we set

$$\mu_k := \frac{1}{|C_k|} \sum_{i \in C_k} x^{(i)}$$

where C_k is the set of examples assigned to centroid k. Concretely, if two examples say $x^{(3)}$ and $x^{(5)}$ are assigned to centroid k=2, then you should update $\mu_2=0.5(x^{(3)} + x^{(5)})$. Complete the code in *computeCentroids.m*. You can implement this function using a loop over the centroids. You can also use a loop over the examples; but if you can use a vectorized implementation that does not use such a loop, your code may run faster. Once you have completed the code in *computeCentroids.m*, the script will run the code and output the centroids after the first step of K-means.

1.3 K-means on example dataset

After you have completed the two functions (*findClosestCentroids* and *computeCentroids*), the next step is to run the K-means algorithm on a toy 2D dataset. Your functions are called from inside the *runKmeans.m* function. The code calls the two functions you implemented in a loop. When you run the next step, the K-means code will produce a visualization that steps you through the progress of the algorithm at each iteration. Press enter multiple times to see how each step of the K-means algorithm changes the centroids and cluster assignments. At the end, your figure should look as the one displayed in Fig. 1.

1.4 Random initialization

In practice, a good strategy for initializing the centroids is to select random examples from the training set. Now, you should complete the function *kMeansInitCentroids.m* with the following code:

```
% Initialize the centroids to be random examples

% Randomly reorder the indices of examples
randidx = randperm(size(X, 1));
% Take the first K examples as centroids
centroids = X(randidx(1:K), :);
```

The code above first randomly permutes the indices of the examples (using *randperm*). Then, it selects the first K examples based on the random permutation of the indices. This allows the examples to be selected randomly without the risk of selecting the same example twice.

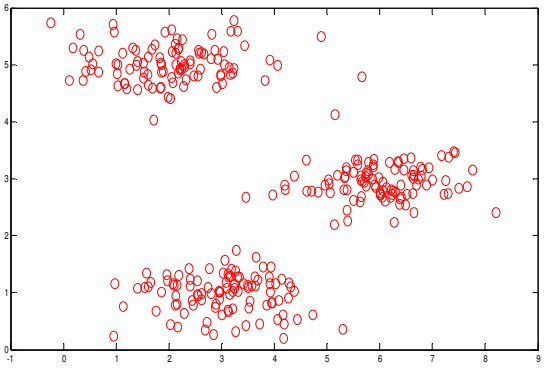

Fig.1a Data set

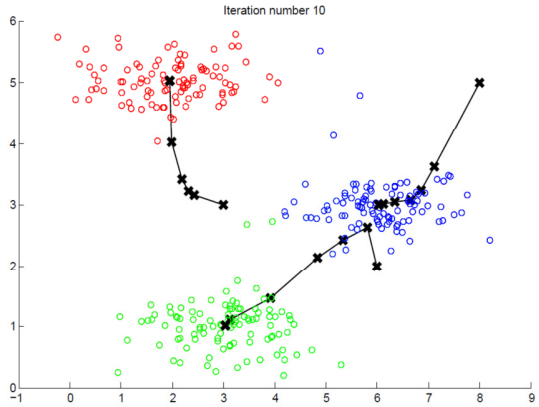

Fig.1b Clustered data after 10 iterations

2. Image compression with K-means

In this exercise, you will apply K-means to image compression. In a 24-bit color representation of an image, each pixel is represented as three 8-bit unsigned integers (ranging from 0 to 255) that specify the red, green and blue intensity values (RGB encoding). Our image contains thousands of colors, and now, you will reduce the number of colors to 16 colors. By making this reduction, it is possible to represent (compress) the photo in an efficient way. Specifically, you only need to store the RGB values of the 16 selected colors, and for each pixel in the image you now need to only store the index of the color at that location (where only 4 bits are necessary to represent 16 possibilities). In this exercise, you will use the K-means algorithm to select the 16 colors that will be used to represent the compressed image. Concretely, you will treat every pixel in the original image as a data example and use the K-means algorithm to find the 16 colors that best group (cluster) the pixels in the 3-dimensional RGB space. Once you have computed the cluster centroids on the image, you will then use the 16 colors to replace the pixels in the original image.

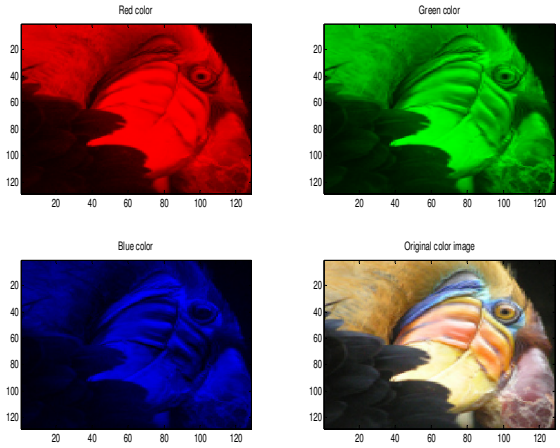

Fig.2 RGB image

2.1 K-means on pixels

Load the image in Matlab/Octave with function *imread.* This creates a 3-dimensional matrix A whose first two indices identify a pixel position and whose last index represents red, green, or blue. For example, A(50,33,3) gives the blue intensity of the pixel at row 50 and column 33. Extract each color dimension as shown in Fig.2 by making the other two dimensions =0. For example, the following code will extract the red color:

I=A; I(:,:,2)=0; I(:,:,3)=0;
subplot(221), imagesc(I), title('Red color')

The code in the script first loads the image, and then reshapes it to create an m×3 2-dimensional matrix of pixel colors (where m=16384= 128×128), and calls the K-means function on it. After finding the top K=16 colors to represent the image, you can now assign each pixel position to its closest centroid using the *findClosestCentroids* function. This allows you to represent the original image using the centroid assignments of each pixel. Notice that you have significantly reduced the number of bits that are required to describe the image. The original image required 24 bits (3*8) for each one of the 128×128 pixel locations, resulting in total size of 128 × 128 × 24 = 393216 bits. The new representation requires 16 colors (i.e. 4 bits per pixel). The final number of bits used is therefore 128×128×4=65920 bits, which corresponds to compressing the original image by about a factor of 6.

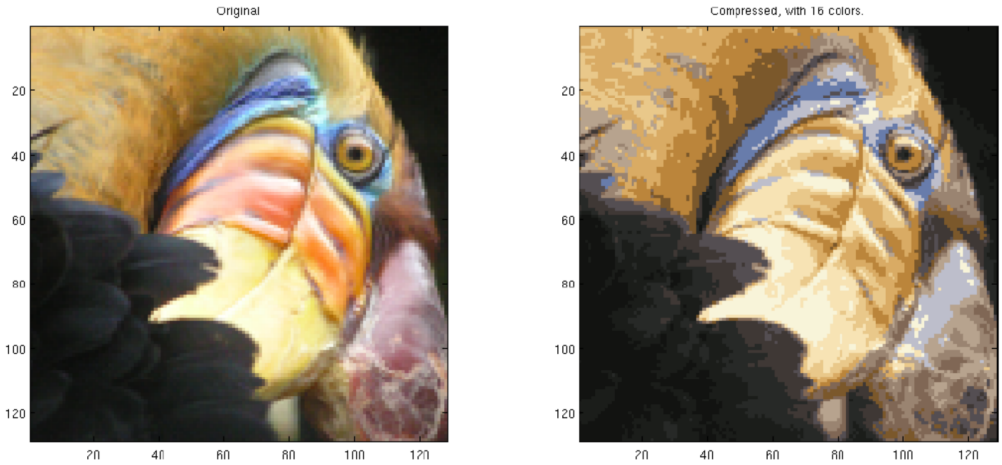

Fig.3: Original and reconstructed image (using K-means to compress the image)

Finally, you can view the effects of the compression by reconstructing the image based only on the centroid assignments. Specifically, you can replace each pixel location with the mean of the centroid assigned to it. Fig. 3 shows the reconstruction we obtained. Even though the resulting image retains most of the characteristics of the original, we also see some compression artifacts.

2.2 Use your own image (optional)

You may modify the code to run on your own image. If your image is very large, then K-means can take a long time to run. We recommend that you resize your images to manageable sizes before running the code. You can also try to vary K to see the effects on the compression.