Lab#9: *K*-means and Principle Component Analysis[1]

Prepared by: Teeradaj Racharak (r.teeradaj@gmail.com)

K-means Clustering

The *K*-means algorithm is a method to automatically cluster similar data examples together. Concretely, you are given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$ (where $x^{(i)} \in \mathbb{R}^n$) 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 closet centroids. Finally, the procedure recomputes the centroids based on the assignments.

The algorithm repeatedly carries out the following two steps:

1. Assigning each training example $x^{(i)}$ to its closet centroid; and,
2. Recomputing the mean of each centroid using the points assigned to it.

The *K*-means algorithm is not convex, *i.e.* the solutions are not unique, and depends on the initial setting of the centroids. Therefore, in practice, the algorithm is usually run a few times with different solutions from different random initializations is to choose the one with the lowest cost function value (distortion).

Now, you are ready to implement these two steps of the algorithm in the next two problems.

Problem 1.1 [Python from scratch]. In first step, 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:

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

where $c^{(i)}$ is the index of the centroid that is closest to $x^{(i)}$, and μ_j is the position (value) of the j^{th} centroid. Your task is to complete the code in `findClosestCentroid`. This function takes the data matrix `X` and the locations of all centroids inside `centroids` and should output a one-dimensional array `indices` that holds the index (a value in $\{1,\ldots,K\}$ where *K* is total number of centroids) of the closet centroid to every training example. Once you have completed the code, you should see the output [1 3 2] corresponding to the centroid assignments for the first 3 examples.

[1] Taken and revised from Andrew Ng's programming assignments on Coursera

Problem 1.2 [Python from scratch]. The second step of the algorithm recomputes, for each centroid, the mean of the points that were assigned to it. Specifically, for every centroid k, we set:

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

where C_k is the set of examples that are assigned to centroid k. For example, if two examples $x^{(3)}, x^{(5)}$ are assigned to centroid $k = 2$, then you should update

$$\mu_2 = \frac{1}{2}\left(x^{(3)} + x^{(5)}\right).$$

Now, you are asked to complete the code in computeCentroids. After you have completed it, the centroids computed after initial finding of closest centroids will be:

[[2.42830111 3.15792418]
[5.81350331 2.63365645]
[7.11938687 3.6166844]]

Problem 1.3 [Python from scratch]. Now, you are ready to implement the K-means algorithm on a provided 2D dataset. Your task is to complete the code in runKmeans. When you implement this function, your code should call the above two functions.

Once you have completed the code, you will see how each step of the algorithm changes the centroids and cluster assignments. Figure 1 - 3 shows the results after running the algorithm for 6 iterations.

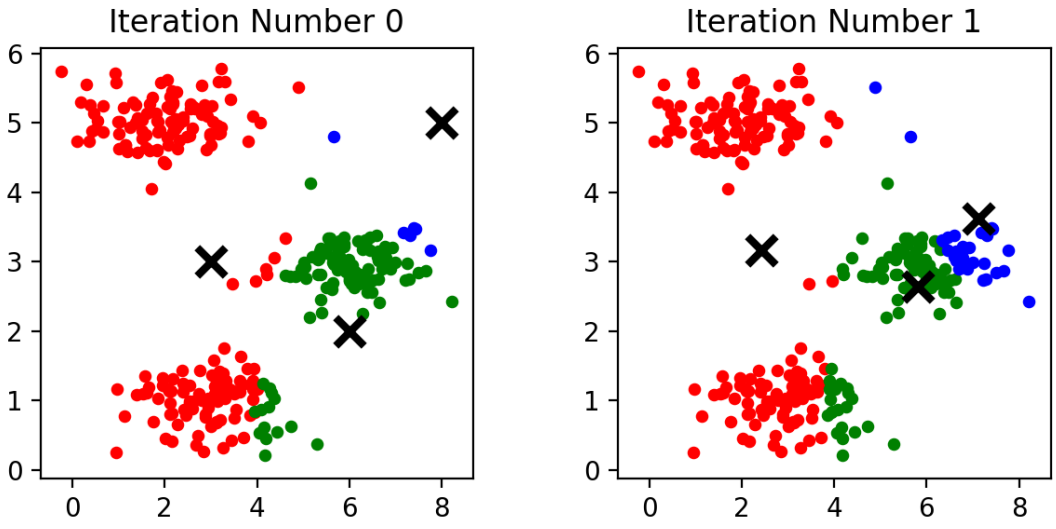

Figure 1 Result after running K-means for two iterations

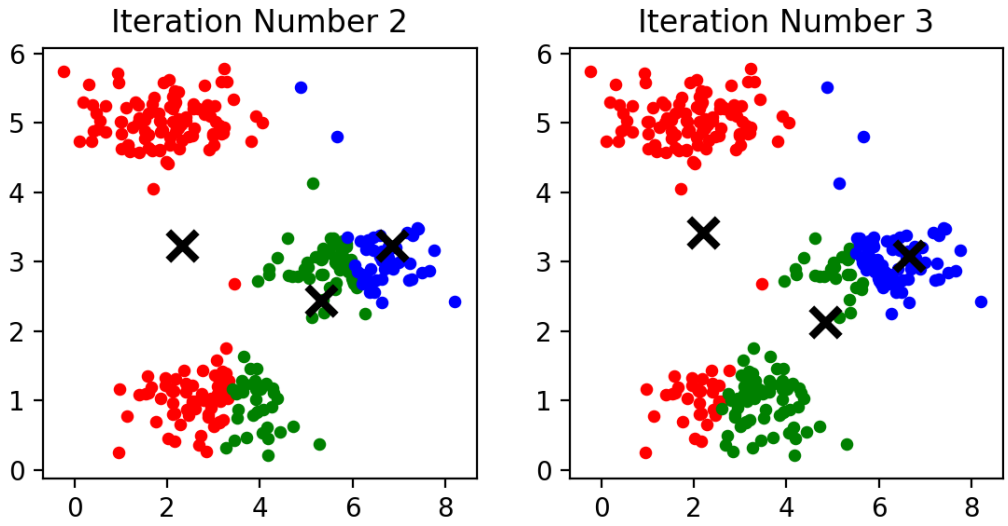

Figure 2 Result after running *K*-means for four iterations

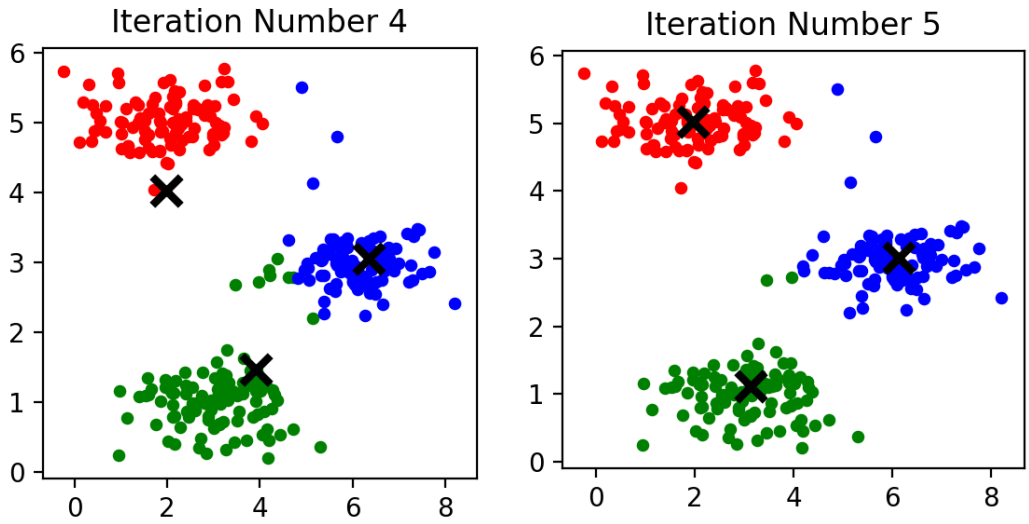

Figure 3 Result after running *K*-means for 6 iterations

Problem 1.4 [Python from scratch]. In the previous problems, the initial assignments of centroids were fixed. That's why you have seen the same figures every time you run it. In practice, a good strategy for initializing the centroids is to select random examples from the training set.

Your task now is to complete the code in kMeansInitCentroids. Your code should also avoid to select duplicated examples.

Principle Component Analysis Setup

In this exercise, you will practice to use principle component analysis (PCA) to perform dimensionality reduction.

To help you understand how PCA works, let's write a program to visualize underlying steps of reducing the data from 2D to 1D. In reality, you may want to reduce data from 256 to 50 dimensions; however, using lower dimensional data (*e.g.* 2D) in this exercise helps us to understand the algorithm better.

PCA consists of two computational steps as follows:

1. Computing the covariance matrix of the data; and
2. Computing the eigenvectors. These will correspond to the principle components of variation in the data.

Problem 2.1 [Python from scratch]. It is important to normalize the data first so that dimensions are scaled on the same range. As we have done in the previous exercises, we may normalize each value x_i in the dataset as follows:

$$x_i := \frac{x_i - \mu_i}{\sigma_i}$$

where μ_i, σ_i represents the mean and the standard deviation of feature i, respectively.

Your task is to complete the code in function `featureNormalize`. The first 3 normalized examples should be:

```
[[-0.52331306 -1.59279252]
 [ 0.46381518  0.84034523]
 [-1.14839331 -0.58315891]]
```

Problem 2.2 [Python from scratch]. Now, you are ready to implement a function to compute the covariance matrix of a provided data. Your task is to complete the code in function `pca`. This function can be defined as follows:

$$\Sigma := \frac{1}{m} X^T X$$

where X is the data matrix with examples in rows, and m is the number of examples. Note that Σ is an $n \times n$ matrix (not the summation operator).

Problem 2.3 [Python from scratch]. After you have computed the covariance matrix, the next step is to write code in function `pca` for computing the principle

components. In numpy.linalg, you can call method svd to compute the principle components and a diagonal matrix.

Once you have completed implementing the function, you should see the principal components (eigenvectors) are:

[[-0.70710678 -0.70710678]
[-0.70710678 0.70710678]]

and the diagonal matrix is [1.73553038 0.26446962].

Dimensionality Reduction with PCA

Now, we are ready to use the computed principal components for reducing the feature dimension of your dataset. As mentioned in the lecture slide, PCA reduces the dimension by projecting each example onto a lower dimensional space *i.e.* $x^{(i)} \rightarrow z^{(i)}$. In this part of the exercise, you will practice to use the eigenvectors to project the dataset into a 1-dimensional space.

In practice, we may train our model with the projected data instead of the original one to speed up your learning performance because there are less dimensions in the input.

Problem 3.1 [Python from scratch]. We are about to project the data onto the principal components. Your task now is to complete the code in function projectData. In particular, this function accepts a dataset X, the principal components U, and the desired number of dimensions to reduce to K. Noted that the top K components in U are given by the first K columns *i.e.* we write U_reduced = U[:,:K] in Python. Once you have completed the code, you should see the result of projecting the first example about 1.4963126084578515.

Problem 3.2 [Python from scratch]. As discussed in the lecture slide, we can also approximately recover the data by 'projecting them back' onto to the original high dimensional space. Now, you are asked to complete the function recoverData so that each example in Z is projected back onto the original space and return the recovered approximation in X_recovered. Once you have completed the implementation, you should see the result of an approximation of the first example about [-1.05805279 -1.05805279].

Problem 3.3 [Python from scratch]. After completing both projectData and recoverData, your next task is to visualize the projection *i.e.* you are asked to complete the code in Step 5 for plotting both the projection and approximation of the recovered data. You should obtain a result similar to Figure 4.

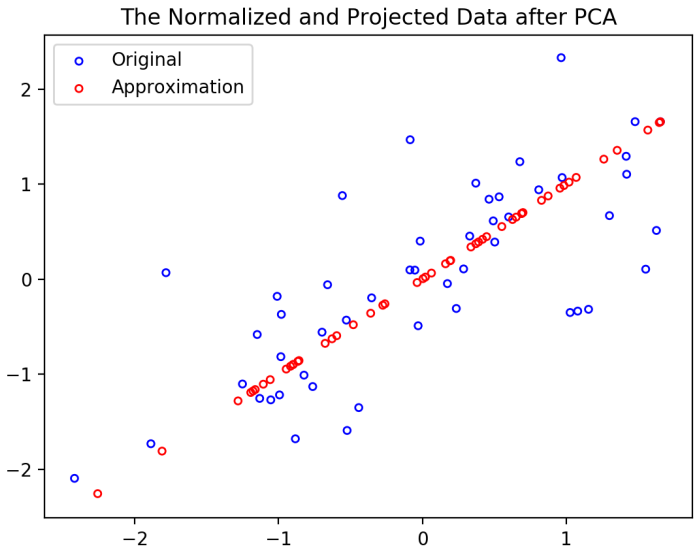

Figure 4 The normalized and projected data after running PCA

Face Image Dataset

In this part of the exercise, you will run PCA on face images to see how it can be used in practice for dimension reduction. The dataset lab9faces.mat contains a dataset X of face images, where each image is represented by 32×32 grayscale. Note that this dataset is based on a cropped version of the labeled faces in the wild dataset[2].

Problem 4.1. [Python from scratch]. It is a good practice to visualize the dataset and have some understanding about it. So, your task is to load and visualize the first 100 of these face images (*cf.* Figure 5).

Problem 4.2 [Python from scratch]. Now, we will run PCA on the face dataset. As we have done previously, we should normalize the dataset first. We can do this task by calling the existing function featureNormalize on the data matrix X; then, call another existing function pca to obtain the principal components of the dataset.

[2] http://vis-www.cs.umass.edu/lfw/

It is worth mentioning that each principal component in U (each row) is a vector of length n, where n = 1,024 for the face dataset. We can also visualize these principal components by 'reshaping' each of them into a 32×32 matrix that corresponds to the pixels in the original dataset. Now, you are asked to visualize the first 100 principal components that describe the largest variation (*cf.* Figure 6).

Figure 5 Visualizing the face datasets

Figure 6 Visualizing the first 100 eigenvectors

Problem 4.3 [Python from scratch]. You have computed the principal components for the face dataset. Now, you will use it to reduce the dimension of the face dataset. This allows you to use your learning algorithm with a smaller input size (*e.g.* 100 dimensions) instead of the original 1024 dimensions. Since the dimension is reduced, it helps speeding up our learning algorithm.

Your task is to project the face dataset onto the first 100 principal components. That is, each face image is now described by a vector $z^{(i)} \in \mathbb{R}^{100}$.

Once you have done that projection, you should understand what is lost in the dimension reduction. To understand that, you are asked to recover the data using only the projected dataset. After you have completed this reconstruction, you may observe that the general structure and appearance of the face do not change; however, the fine details are lost. This is a remarkable reduction (more than 10x) in the dataset size that can help speed up your learning algorithm significantly. For example, when you are training a neural network to perform face recognition, you can use the reduced dimension (*e.g.* 100 dimensions $\Longleftrightarrow$ 10×10 pixels) instead of 32×32 pixels. Figure 7 shows the example of this reconstruction.

Figure 7 Visualizing the reconstructed data