Boston University, Department of Computer Science

CS 542 Assignment 3 (Total Points: 200.)

K-means Clustering and Principal Component Analysis

September 28, 2018

1 Introduction

In this exercise, you will implement the K-means clustering algorithm and apply it to compress an image. In the second part, you will use principal component analysis to find a low-dimensional representation of face images.
To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise.
For this problem set we won't be using iPython notebooks. Simply run your code with Python interpreters. Please submit a zip file with completed codes and a PDF for solutions to the written part.

2 K-means Clustering

In this this exercise, you will implement the K-means algorithm and use it for image compression. 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. You will be using k_means_clustering.py for this part of the exercise.

2.1 Implementing K-means

The K-means algorithm is a method to automatically cluster similar data examples together. Concretely, you are given a training set $\{x^1, \cdots, x^m\}$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 closest centroids and then recomputing the centroids based on the assignments.
The inner loop of the algorithm repeatedly carries out two steps:

1. Assigning each training example x to its closest centroid

2. Recomputing 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). You will implement the two phases of the K-means algorithm separately in the next sections.

2.1.1 Finding closest centroids[30pts]

In the cluster assignment phase of the K-means algorithm, 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

$$c^i := \arg\min_j ||x^i - \mu_j||^2$$

where $c(i)$ is the index of the centroid that is closest to x^i, and j is the position (index) of the j-th centroid.
Your task is to complete the code in find_closest_centroids. This function takes the data matrix samples and the locations of all centroids inside centroids and should output a one-dimensional array of clusters that holds the index (a value in $\{1, \cdots, K\}$, 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.
Once you have completed the code in find_closest_centroids, you can run it and you should see the output $[0, 2, 1,]$ corresponding to the centroid assignments for the first 3 examples.
Please take a look at Figure 2.1 to gain an understanding of the distribution of the data. It is two dimentional, with x_1 and x_2.

2.1.2 Computing centroid means[30pts]

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

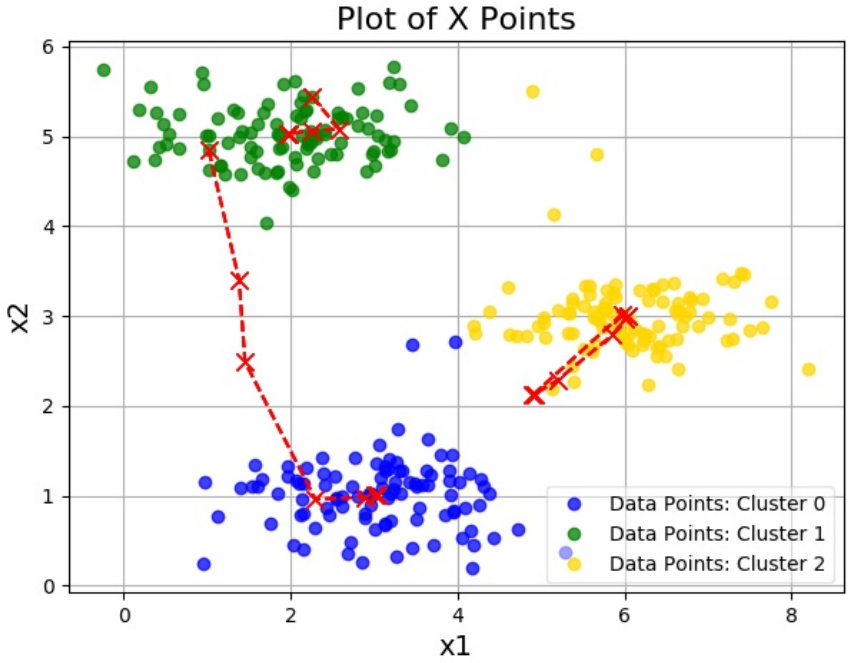

Figure 2.1: Result 1

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

where C_k is the set of examples that are assigned to centroid k. Concretely, if two examples say x^3 and x^5 are assigned to centroid $k = 2$, then you should update

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

Once you have completed the code in get_centroids, the k_means_clustering.py will run your code and output the centroids after the first step of K-means. The final centroids should be [[1.95399466 5.02557006] [3.04367119 1.01541041] [6.03366736 3.00052511]]. 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. Close figure 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 Figure 2.1.

2.2 Random initialization[10pts]

The initial assignments of centroids for the example dataset in previous section were designed so that you will see the same figure as Figure 2.1. In practice, a good strategy for initializing the centroids is to select random examples from the training set.
In this part of the exercise, you should complete the function choose_random_centroids. You should randomly permute the indices of the examples (using random seed 7). Then, it selects the first K examples based on the random permutation of the indices. This should allow the examples to be selected at random without the risk of selecting the same example twice. You will see how random initialization will affect the first few iterations of clustering, and also possibly, result in a different cluster assignment.

2.3 Image compression with K-means[30pts]

In this exercise, you will apply K-means to image compression.
In a straightforward 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. This encoding is often referred to as the RGB encoding. Our image contains thousands of colors, and in this part of the exercise, 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.
Please refer to k_means_compression.py for this exercise.
The main method creates a three-dimensional matrix bird_small whose first two indices identify a pixel position and whose last index represents red, green, or blue. For example, $bird\_small(50, 33, 3)$ gives the blue intensity of the pixel at row 50 and column 33.
The main method then reshapes it to create an $m \times 3$ matrix of pixel colors (where $m = 16384 = 128 \times 128$ and 128 is the size on the columns and rows), and calls your 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 for each one of the 128×128 pixel locations, resulting in total size of $128 \times 128 \times 24 = 393,216$ bits. The new representation requires some overhead storage in form of a dictionary of 16 colors, each of which require 24 bits,

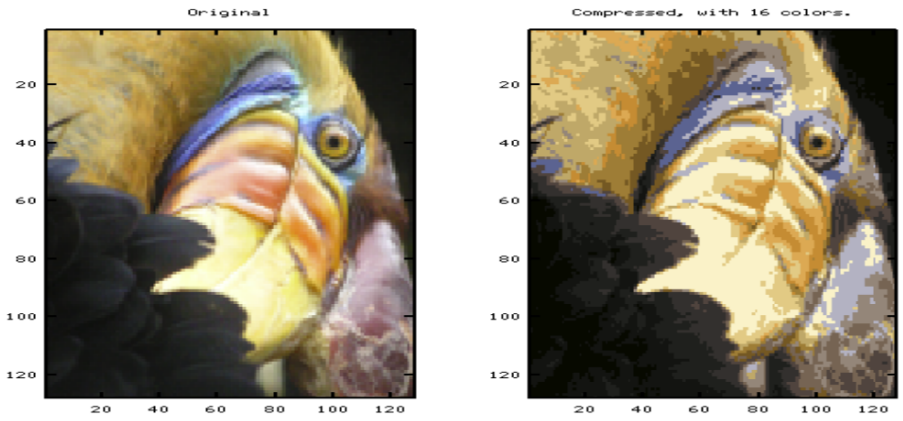

Figure 2.2: Original and reconstructed image (when using K-means to compress the image).

but the image itself then only requires 4 bits per pixel location. The final number of bits used is therefore $16 \times 24 + 128 \times 128 \times 4 = 65,920$ bits, which corresponds to compressing the original image by about a factor of 6.
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. Figure 2.2 shows the reconstruction we obtained. Even though the resulting image retains most of the characteristics of the original, we also see some compression artifacts.

3 Principal Component Analysis

In this exercise, you will use principal component analysis (PCA) to perform dimensionality reduction. You will first experiment with an example 2D dataset to get intuition on how PCA works, and then use it on a bigger dataset of 5000 face image dataset. The provided script, pca.py, will help you step through the first half of the exercise.

3.1 Example Dataset[20pts]

To help you understand how PCA works, you will first start with a 2D dataset which has one direction of large variation and one of smaller variation. The script pca.py will plot the training data (Figure 3.1). In this part of the exercise, you will visualize what happens when you use PCA to reduce the data from 2D to 1D. In practice, you might want to reduce data from 256 to 50 dimensions, say; but using lower dimensional data in this example allows us to visualize the algorithms better.

Figure 3.1: Example Dataset

3.2 Implementing PCA

In this part of the exercise, you will implement PCA. PCA consists of two computational steps: First, you compute the covariance matrix of the data. Then, you use scipy.linalg SVD function to compute the eigenvectors $U_1, U_2, \cdots, U_n$. These will correspond to the principal components of variation in the data. Before using PCA, it is important to first normalize the data by subtracting the mean value of each feature from the dataset, and scaling each dimension so that they are in the same range.

After normalizing the data, you can run PCA to compute the principal components. You task is to complete the code in pca.py to compute the principal components of the dataset. First, you should compute the covariance matrix of the data, which is given by:

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

where X is the data matrix with examples in rows, and m is the number of examples. Note that Σ is a $n \times n$ matrix and not the summation operator. After computing the covariance matrix, you can run SVD on it to compute the principal components.

Once you have completed pca.py, the script will run PCA on the example dataset and plot the corresponding principal components found (Figure 3.2). The script will also output the top principal component (eigenvector) found, and you should expect to see an output of about [-0.707 -0.707].

3.3 Dimensionality Reduction with PCA

After computing the principal components, you can use them to reduce the feature dimension of your dataset by projecting each example onto a lower dimensional space, $x^i \rightarrow z^i$ (e.g., projecting the data from 2D to 1D). In this part of the exercise, you will use the eigenvectors returned by PCA and project the example dataset into a 1-dimensional space.

In practice, if you were using a learning algorithm such as linear regression or perhaps neural networks, you could now use the projected data instead of the original data. By using the projected data, you can train your model faster as there are fewer dimensions in the input.

3.3.1 Projecting the data onto the principal components[10pts]

You should now complete the code in project_data. Specifically, you are given a dataset of samples, the principal components U, and the desired number of dimensions to reduce to K. You should project each example in samples onto the top K components in U. Note that the top K components in U are given by the first K columns of U, that is $reduced\_U = U(:, 1:K)$.

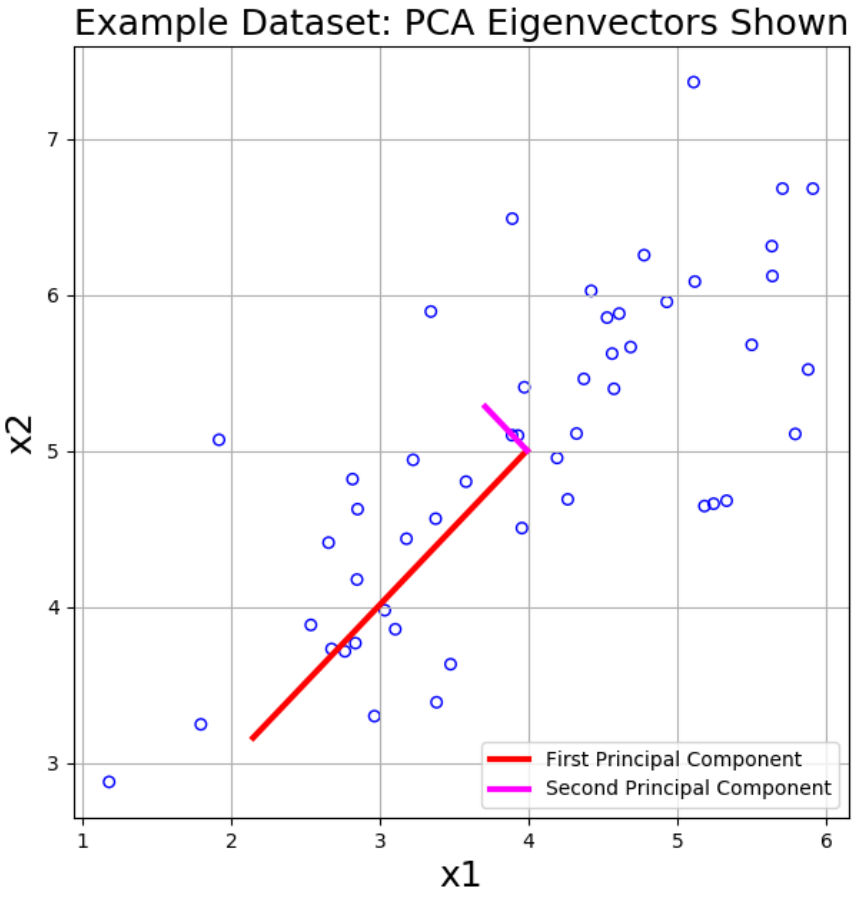

Figure 3.2: Computed eigenvectors of the dataset

Once you have completed the code, it will project the first example onto the first dimension and you should see a value of about 1.49 (or possibly -1.49, if you got -U1 instead of U1).

3.3.2 Reconstructing an approximation of the data [10 pts]

After projecting the data onto the lower dimensional space, you can approximately recover the data by projecting them back onto the original high dimensional space. Your task is to complete recover_data to project each example in Z back onto the original space and return the recovered approximation in recovered_sample.
Once you have completed the code, it will recover an approximation of the first example and you should see a value of about [-1.05 -1.05].

3.3.3 Visualizing the projections[10pts]

After completing both project_data and recover_data, the script will now perform both the projection and approximate reconstruction to show how the projection affects the data. In Figure 3.3, the original data points are indicated with the blue circles, while the projected data points are indicated with the red circles. The projection effectively only retains the information in the direction given by U1.

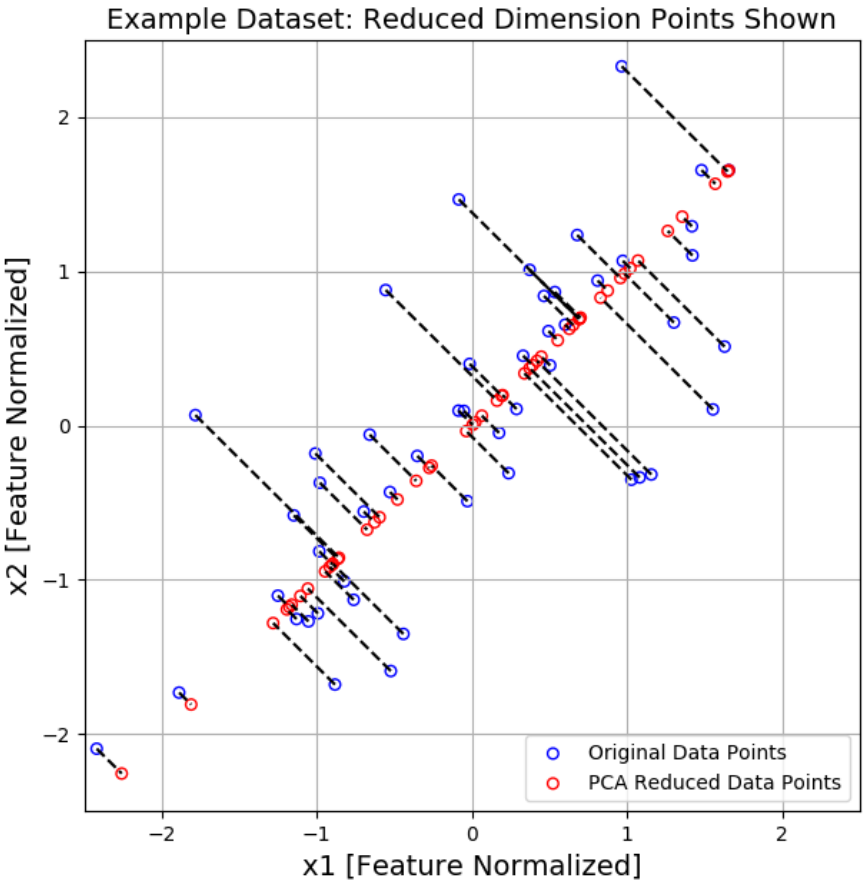

Figure 3.3: The normalized and projected data after PCA

3.4 Face Image Dataset[20 pts]

In this part of the exercise, you will use face_image.py to run PCA on face images to see how it can be used in practice for dimension reduction. The dataset faces.mat contains

a dataset X of face images, each 32×32 in grayscale. Each row of X corresponds to one face image (a row vector of length 1024). The next step in face_image.py will load and visualize the first 100 of these face images (Figure 3.4).

Figure 3.4: Faces dataset

3.4.1 PCA on Faces

To run PCA on the face dataset, we first normalize the dataset by subtracting the mean of each feature from the data matrix X.
After running PCA, you will obtain the principal components of the dataset. Notice that each principal component in U (each row) is a vector of length n (where for the face dataset, n = 1024). It turns out that we can visualize these principal components

by reshaping each of them into a 32×32 matrix that corresponds to the pixels in the original dataset. The script face_image.py displays the first 36 principal components that describe the largest variations (Figure 3.5). If you want, you can also change the code to display more principal components to see how they capture more and more details.

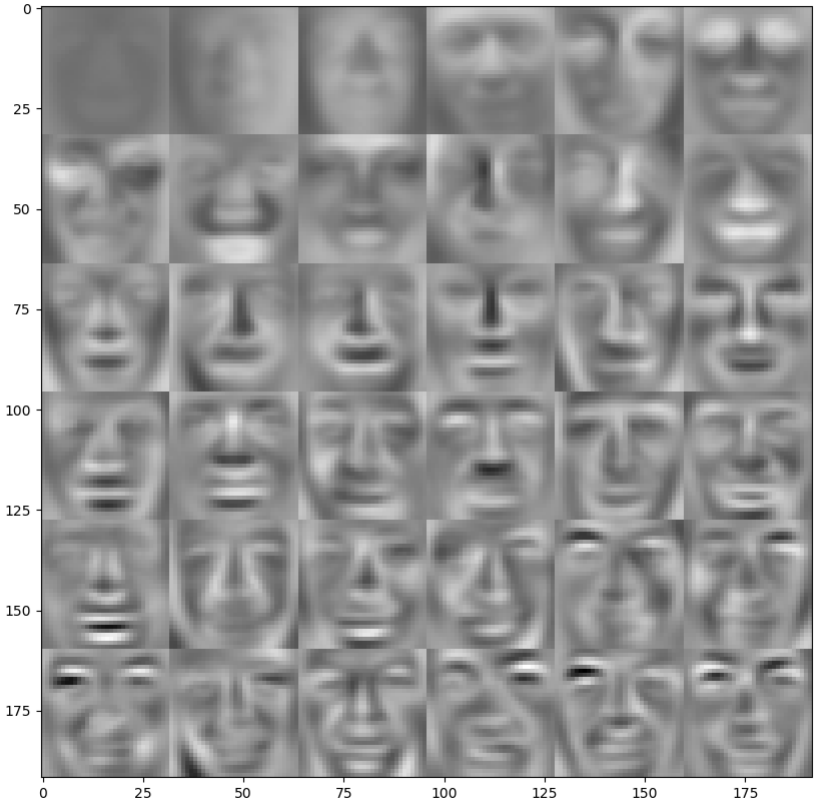

Figure 3.5: Principal components on the face dataset

3.4.2 Dimensionality Reduction

Now that you have computed the principal components for the face dataset, you can 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. This can help speed up your learning algorithm.

The next part in face_image.py will project the face dataset onto only the first 100 principal components. Concretely, each face image is now described by a vector $z^i \in \mathbb{R}^{100}$. To understand what is lost in the dimension reduction, you can recover the data using only the projected dataset. In the code, an approximate recovery of the data is performed and the original and projected face images are displayed side by side (Figure 3.6). From the reconstruction, you can observe that the general structure and appearance of the face are kept while the fine details are lost. This is a remarkable reduction (more than $10\times$) in the dataset size that can help speed up your learning algorithm significantly. For example, if you were training a neural network to perform person recognition (given a face image, predict the identity of the person), you can use the dimension reduced input of only a 100 dimensions instead of the original pixels.

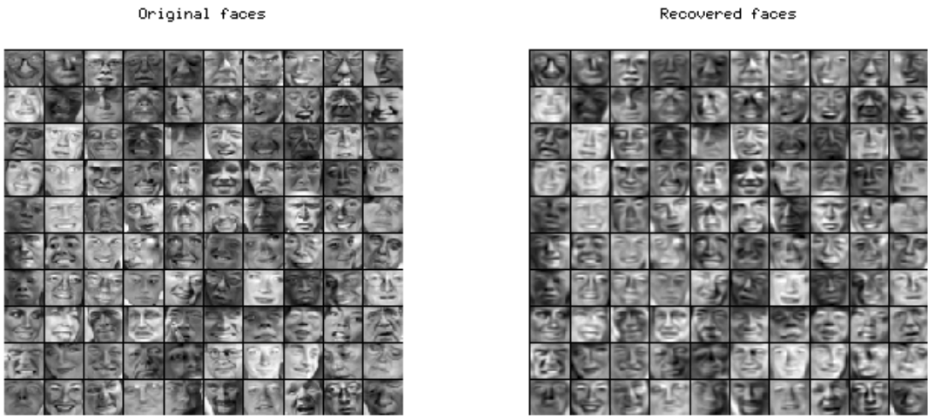

Figure 3.6: Original images of faces and ones reconstructed from only the top 100 principal components.

4 Written Questions[10 pts each]

4.1 Problem 1[10 pts]

Read Section 9.1 in Bishop that discusses the K-means algorithm, and solve problem 9.1 which asks you to prove that it converges.
Consider the K-means algorithm discussed in Section 9.1. Show that as a consequence of there being a finite number of possible assignments for the set of discrete indicator variables r_{nk} and that for each such assignment there is a unique optimum for the $\{\mu_k\}$ the K-means algorithm must converge after a finite number of iterations.

4.2 Problem 2[10 pts]

Read the beginning of Section 9.2 which describes Gaussian mixture models, and solve Problem 9.3.
Consider a Gaussian mixture model in which the marginal distribution $p(z)$ for the latent variable is given by (9.10) and the conditional distribution $p(x|z)$ for the observed variable is given by (9.11). Show that the marginal distribution $p(x)$ obtained by summing $p(z)p(x|z)$ over all possible values of z is a Gaussian mixture of the form (9.7).

4.3 Problem 3[10 pts]

Go through Section 12.1.2 which describes the Minimum-error formulation of PCA and perform omitted computations. Specifically, do all the derivations necessary to show that

0. Before (12.9)
$$\alpha_{nj} = \mathrm{x}_n^{\mathrm{T}} \mathrm{u}_j$$
1. (12.12)
$$z_{nj} = \mathrm{x}_n^{\mathrm{T}} \mathrm{u}_j$$
2. (12.13)
$$b_j = \overline{\mathrm{x}}^{\mathrm{T}} \mathrm{u}_j$$
3. In case of two-dimensional data space
$$\mathrm{S}\mathrm{u}_2 = \lambda_2 \mathrm{u}_2$$
$$J = \lambda_2$$