

Introduction

In this problem, 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, please download the code base `pa5.zip` from Owlspace. When you unzip the archive, you will see the following files.

Name	Edit?	Read?	Description
<code>findClosestCentroids.m</code>	Yes	Yes	Find closest centroids
<code>computeCentroids.m</code>	Yes	Yes	Compute centroid means
<code>kMeansInitCentroids.m</code>	Yes	Yes	Initialization for K-means centroids
<code>pca.m</code>	Yes	Yes	Perform principal component analysis
<code>projectData.m</code>	Yes	Yes	Projects a data set into a lower dimensional space
<code>recoverData.m</code>	Yes	Yes	Recovers the original data from the projection
<code>ex5.m</code>	No	Yes	Matlab script that will run your functions for k-means clustering
<code>ex5_pca.m</code>	No	Yes	Matlab script that will run your functions for PCA
<code>ex5data1.mat</code>	No	No	Example dataset for PCA
<code>ex5data2.mat</code>	No	No	Example dataset for k-means
<code>ex5faces.mat</code>	No	No	faces dataset
<code>bird_small.png</code>	No	No	Example image
<code>displayData.m</code>	No	No	Displays 2D data stored in a matrix
<code>drawLine.m</code>	No	No	Draws a line over an existing figure
<code>plotDataPoints.m</code>	No	No	Plot data points
<code>plotProgresskMeans.m</code>	No	No	Plots each step of k-means
<code>featureNormalize.m</code>	No	Yes	normalize data for PCA
<code>runkMeans.m</code>	No	Yes	Runs the k-means algorithm
<code>pa5_2015.pdf</code>	No	Yes	this document

Problem 1: k-means clustering (25 points)

In this problem, 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 intuition about 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 `ex5.m` for this part.

The k-means algorithm is a method to automatically cluster similar examples together. That is, given a training set $\{x^{(1)}, \dots, x^{(m)}\}$ (where $x^{(i)} \in \mathbb{R}^d$), k-means groups the data into a few cohesive clusters. The intuition behind k-means is an iterative procedure that starts by guessing the initial cluster centroids, and then refines this guess by repeatedly assigning examples to their closest centroids and then recomputing 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 = computeCentroids(X, idx, K);
end
```

The inner-loop of the algorithm repeatedly carries out two steps: (i) Assigning each training example $x^{(i)}$ to its closest centroid, and (ii) Recomputing each centroid using the points assigned to it. The k-means algorithm will always converge to some final set of centroids. Note that the converged solution may not always be ideal and will depend 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 (distortion). You will implement the two phases of the k-means algorithm separately in the next sections.

Finding closest centroids (10 points)

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

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

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. Note that $c^{(i)}$ corresponds to $\text{idx}(i)$ in our code.

Your task is to complete the code in `findClosestCentroids.m`. This function takes the data matrix `X` and the locations of all centroids inside `centroids` and should output a one-dimensional array `idx` that holds the index (a value in $\{1, \dots, K\}$, 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.

When you run the script `ex5.m` you should see the output `[1 3 2]` corresponding to the centroid assignments for the first 3 examples in our data set.

Computing centroid means (10 points)

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 j we set

$$\mu_j = \frac{1}{|C_j|} \sum_{i \in C_j} x^{(i)}$$

where C_j is the set of examples that are assigned to centroid j .

You should now 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 `ex5.m` will run your code and output the centroids after the first step of k-means.

k-means on example dataset

After you have completed the two functions (`findClosestCentroids` and `computeCentroids`), the next step in `ex5.m` will run the k-means algorithm on a toy 2D dataset to help you understand how k-means works. Your functions are called from inside the `runKmeans.m` script. We encourage you to take a look at the function to understand how it works. Notice that 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 Figure 1.

Random initialization (5 points)

The initial assignments of centroids for the example dataset in `ex5.m` were designed so that you will see the same figure as in Figure 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 `kMeansInitCentroids.m`. First, randomly permute the indices of the examples (using `randperm`). Then, select the first K examples based on the random permutation of the indices. This allows the examples to be selected at random without the risk of selecting the same example twice.

Image compression with k-means

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

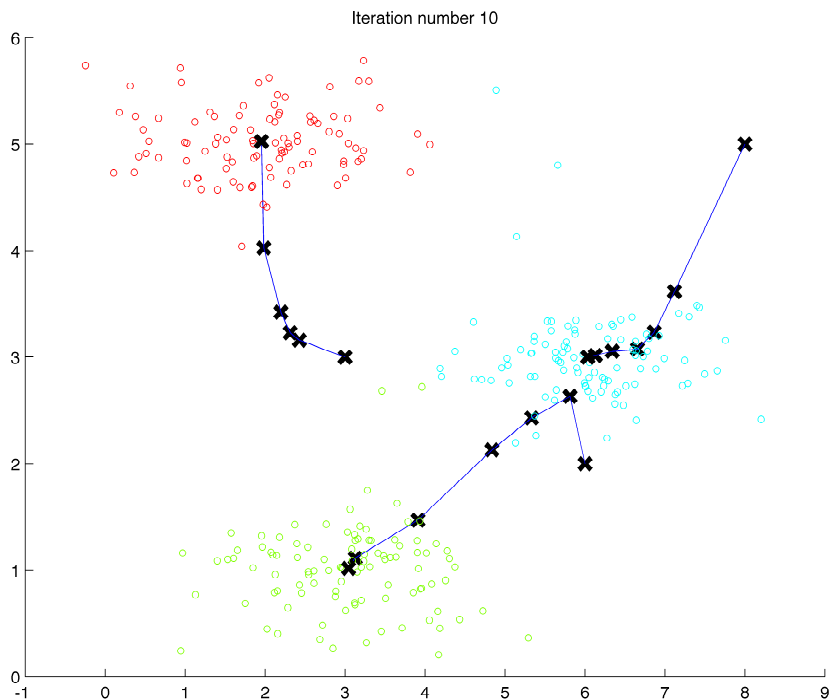


Figure 1: Expected output of k-means

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. In particular, 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.

k-means on pixels

In Matlab, images can be read in as follows:

```
A = imread('bird_small.png');
```

This creates a three-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

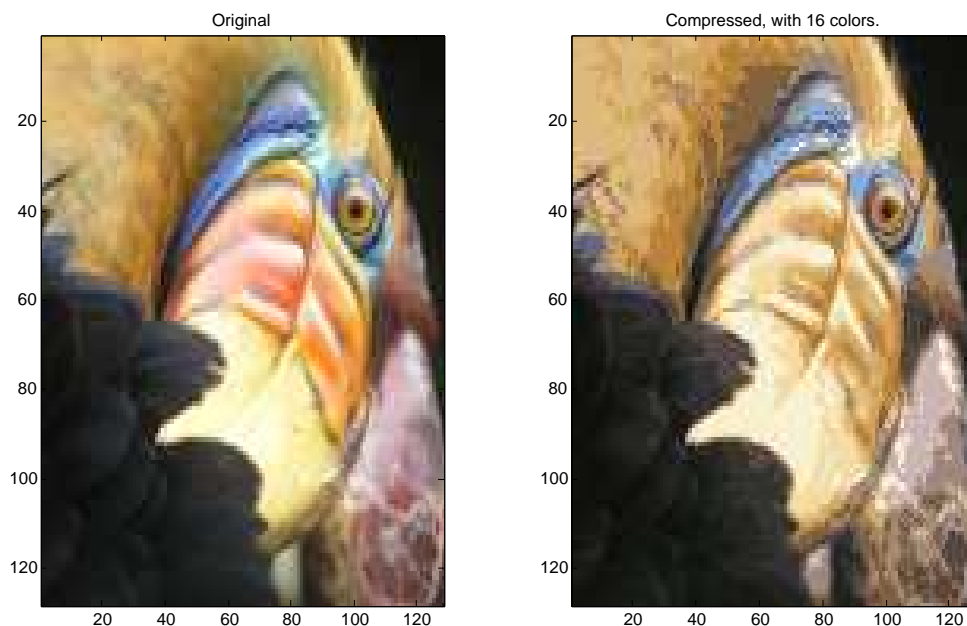


Figure 2: Original and reconstructed image (when using k-means to compress the image).

the pixel at row 50 and column 33. The code inside `ex5.m` first loads the image, and then reshapes it to create an $m \times 3$ matrix of pixel colors (where $m = 16384 = 128 \times 128$), 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, 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 shows the reconstruction we obtained. Even though the resulting image retains most of the characteristics of the original, we also see some compression artifacts.

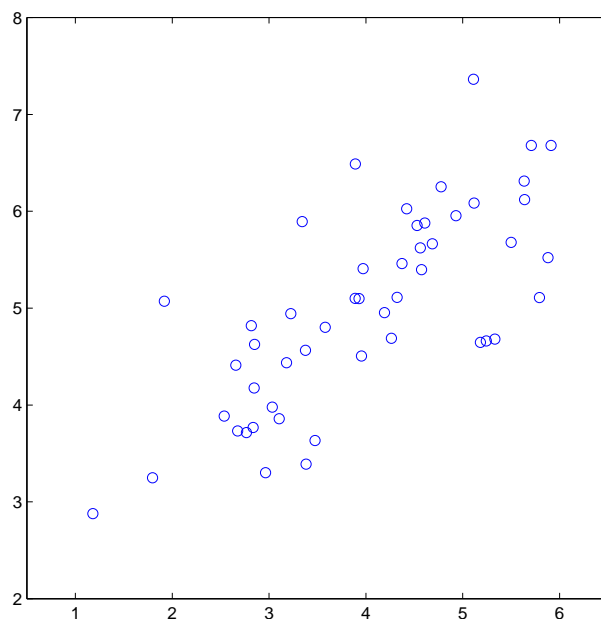


Figure 3: Example Dataset 1

Principal Components Analysis (15 points)

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, `ex5_pca.m`, will help you step through the exercise.

Example Dataset

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 `ex5_pca.m` will plot the training data (Figure 3). 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.

Implementing PCA (5 points)

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 Matlabs SVD function to

compute the eigenvectors U_1, U_2, \dots, U_n . These will correspond to the principal components of variation in the data. 7

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. In the provided script `ex5_pca.m`, this normalization has been performed for you using the `featureNormalize` function. After normalizing the data, you can run PCA to compute the principal components. Your task is to complete the code in `pca.m` 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 $d \times d$ matrix and not the summation operator.

After computing the covariance matrix, you can run SVD on it to compute the principal components. In Matlab, you can run SVD with the following command:

```
[U, S, V] = svd(Sigma)
```

where U will contain the principal components and S will contain a diagonal matrix.

Once you have completed `pca.m`, the `ex5_pca.m` script will run PCA on the example dataset and plot the corresponding principal components found (Figure 4). The script will also output the top principal component (eigenvector) found, and you should expect to see an output of about $[-0.707 \ -0.707]$. (It is possible that Matlab may instead output the negative of this, since U_1 and $-U_1$ are equally valid choices for the first principal component.)

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.

Projecting the data onto the principal components (5 points)

You should now complete the code in `projectData.m`. Specifically, you are given a dataset X , the principal components U , and the desired number of dimensions to reduce to K . You should project each example in X 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 $U(:, 1:K)$. Once you have completed the code in `projectData.m`, `ex5_pca.m` will project the first example onto the first dimension and you should see a value of about 1.481 (or possibly -1.481, if you got $-U_1$ instead of U_1).

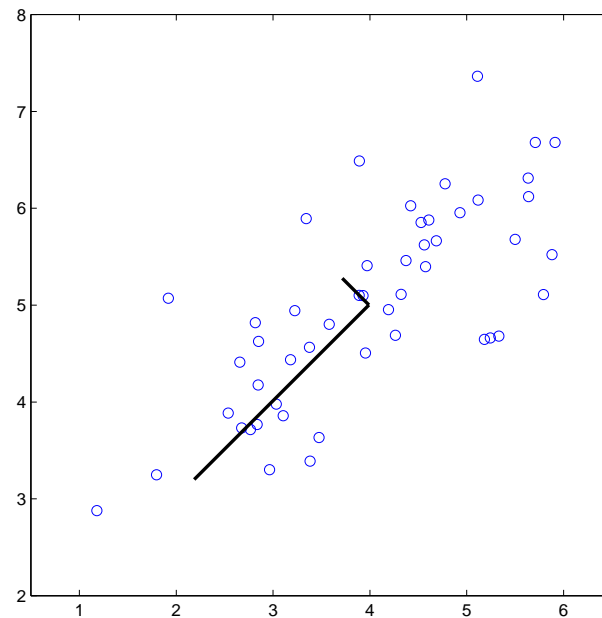


Figure 4: Computed eigenvectors of the dataset

Reconstructing an approximation of the data (5 points)

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 `recoverData.m` to project each example in `Z` back onto the original space and return the recovered approximation in `X_rec`. Once you have completed the code in `recoverData.m`, `ex5_pca.m` will recover an approximation of the first example and you should see a value of about `[-1.047 -1.047]`.

Visualizing the projections

After completing both `projectData` and `recoverData`, `ex5_pca.m` will now perform both the projection and approximate reconstruction to show how the projection affects the data. In Figure 5, 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 U_1 .

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 `ex5faces.mat` contains a dataset `X` of face images, each 32×32

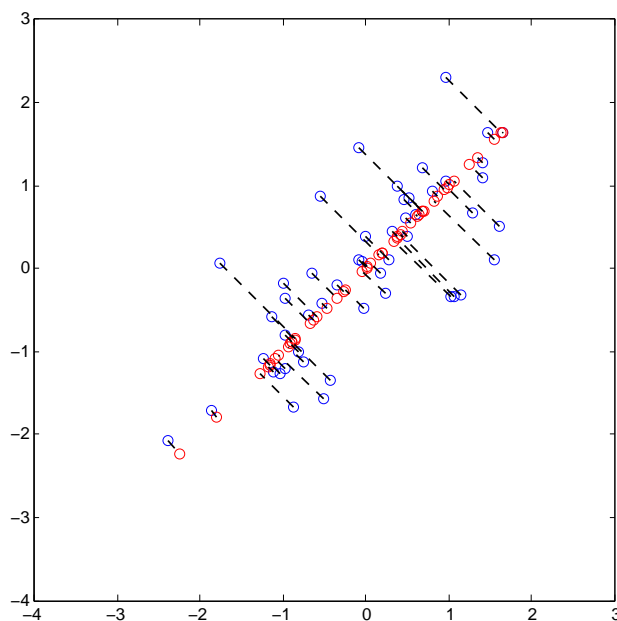


Figure 5: The normalized and projected data after PCA

in grayscale. Each row of \mathbf{X} corresponds to one face image (a row vector of length 1024). The next step in `ex5_pca.m` will load and visualize the first 100 of these face images (Figure 6).

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 \mathbf{X} . The script `ex5_pca.m` will do this for you and then run your PCA code. After running PCA, you will obtain the principal components of the dataset. Notice that each principal component in \mathbf{U} (each row) is a vector of length d (where for the face dataset, $d = 1024$). 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 `ex5_pca.m` displays the first 36 principal components that describe the largest variations (Figure 7). If you want, you can also change the code to display more principal components to see how they capture more and more details.

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



Figure 6: Faces dataset

your learning algorithm.

The next part in `ex5_pca.m` 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 `ex5_pca.m`, an approximate recovery of the data is performed and the original and projected face images are displayed side by side (Figure 8). 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.

What to turn in

Please zip up all the files in the archive (including files that you did not modify) and submit it as `pa5_netid.zip` on Owlspace before the deadline. Include a PDF file in the archive that presents your plots and your discussion of results from the problems above.



Figure 7: Principal components on the face dataset

Acknowledgment

These problems are from Andrew Ng's exercise on k-means and principal components analysis.

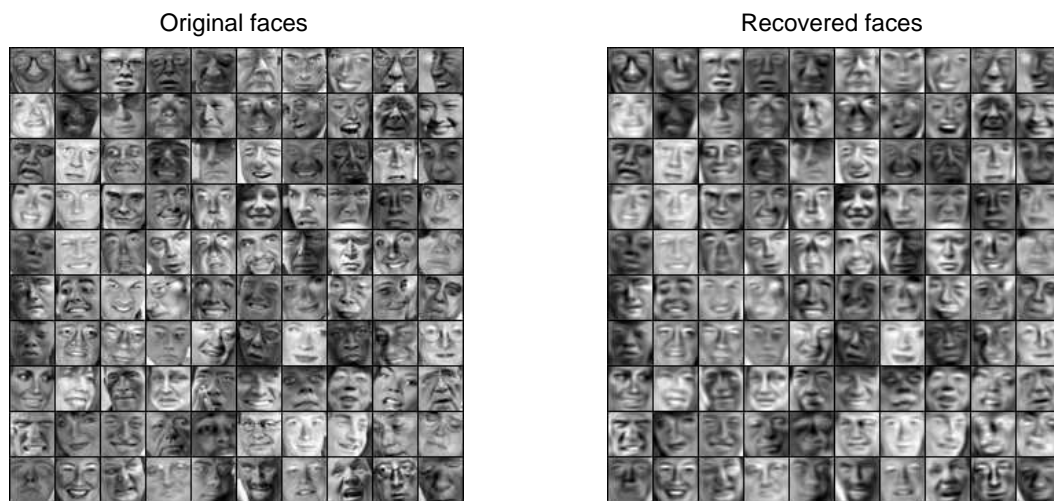


Figure 8: Original and reconstructed face dataset reconstructed from only the top 100 principal components