Problem Set 4: Clustering and Dimensionality Reduction

To run and solve this assignment, one must have a working IPython Notebook installation. The easiest way to set it up for both Windows and Linux is to install Anaconda. Then save this file to your computer, run Anaconda and choose this file in Anaconda's file explorer. Use `Python 3` version. The statements below assume that you have already followed these instructions. If you are new to Python or its scientific library, Numpy, there are some nice tutorials here and here.

To run code in a cell or to render Markdown+LaTeX press `Ctr+Enter` or `[>|]` (like "play") button above. To edit any code or text cell double click on its content. To change cell type, choose "Markdown" or "Code" in the drop-down menu above. Here are some useful resources for Markdown guide and LaTeX tutorial if you are not familiar with the basic syntax.

Submission Guidelines

You need to submit **the output of the notebook as PDF** and **an additional .py file** on Gradescope.

For the .py file, Here are the guidelines :

1. The .py file should contain ONLY your typed codes. (Do not include any other code apart from what you coded for the assignment)
2. .py should NOT contain any written answers. Only code written by you.
3. If your typed code falls under a function definition thats predefined by us, Only include your typed code and nothing else.
4. For each cell block within colab/jupyter that you typed your code in, Add 2 new lines ("\n") before pasting your typed code in the .py file.

The assignment is graded on completion. You will also need to **answer** whether you attempted the question or not in Gradescope. It is extremely important that you complete all assignments (both the programming and written questions), as they will prepare you for quizzes and exams.

Total: 160 points (90 (Q1) + 70 (Q2)) plus **40 BONUS points**.

1. Clustering

The following exercises help you better understand clustering methods. First, in exercises 1.1-5, you will implement the K-means algorithm. You will practice on an toy 2D dataset that will help you gain an intuition of how the K-means algorithm works.

In exercises 1.6-7, you will derive the Mixture of Gaussians distribution and compare and contrast the K-means algorithm to the GMM algorithm.

Problem 1.8 is a BONUS question asking you to 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.

1.1 Implementing K-means

The K-means algorithm is a method to automatically cluster similar data examples together. Concretely, you are given a training set $\{x_1, \cdots, x_m\}\text{where } x_i \in \mathbb{R}^n$, and we 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 the 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.

1.2 Visualizing the data

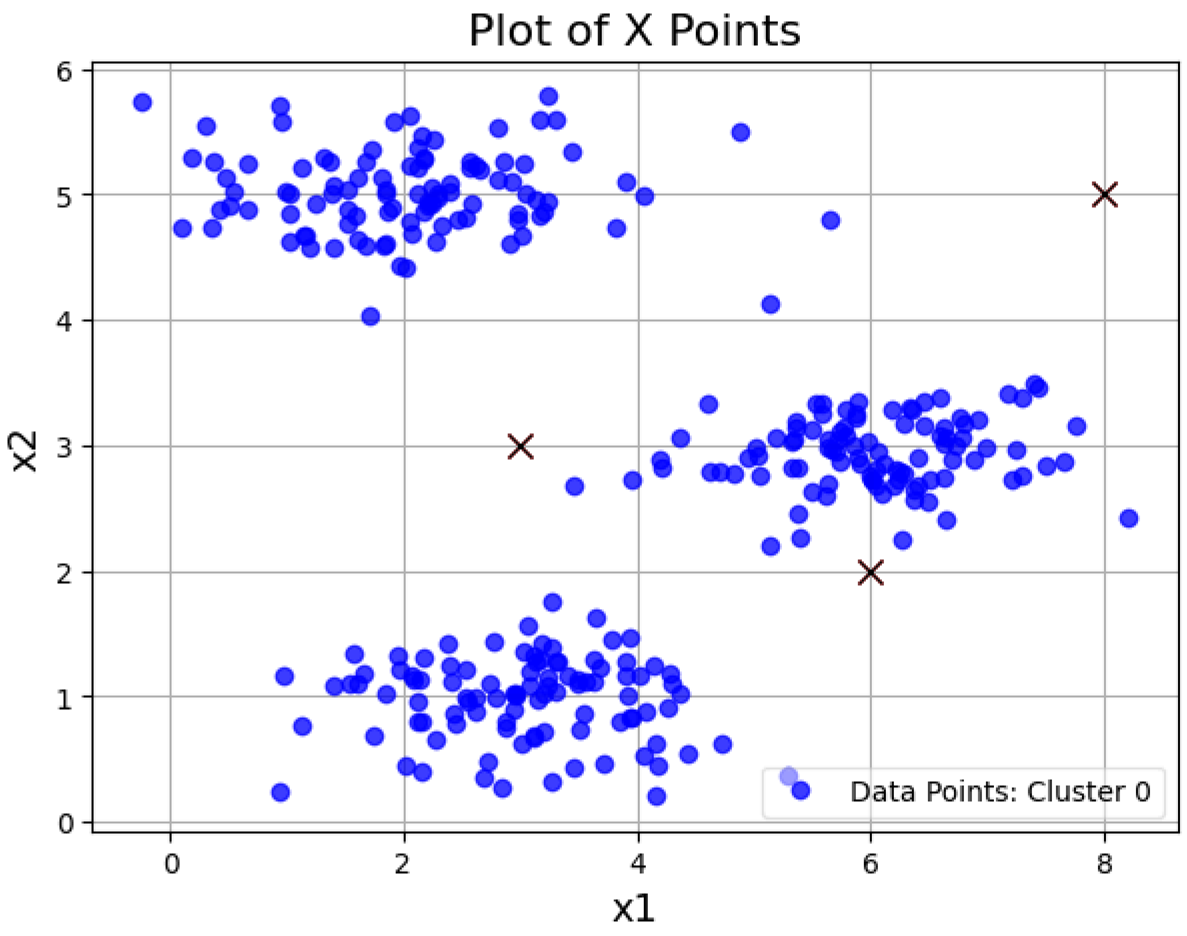
Run the provided code below and take a look at the resulting plot to gain an understanding of the distribution of the data. It is two dimensional, with x_1 and x_2. The "x" markers are the initial centroids of the clustering algorithm, where $K = 3$.

In [1]:
```python
from __future__ import absolute_import

import random

import matplotlib.pyplot as plt
import numpy as np
import scipy.io
import scipy.misc
```

```
def plot_data(samples, centroids, clusters=None):
    """
    Plot samples and color it according to cluster centroid.
    :param samples: samples that need to be plotted.
    :param centroids: cluster centroids.
    :param clusters: list of clusters corresponding to each sample.
    If clusters is None, all points are plotted with the same color.
    """

    colors = ['blue', 'green', 'gold']
    assert centroids is not None

    if clusters is not None:
        sub_samples = []
        for cluster_id in range(centroids[0].shape[0]):
            sub_samples.append(np.array([samples[i] for i in range(samples.s
    else:
        sub_samples = [samples]

    plt.figure(figsize=(7, 5))

    for cluster_id, clustered_samples in enumerate(sub_samples):
        plt.plot(clustered_samples[:, 0], clustered_samples[:, 1], 'o', colo
                 label='Data Points: Cluster %d' % cluster_id)

    plt.xlabel('x1', fontsize=14)
    plt.ylabel('x2', fontsize=14)
    plt.title('Plot of X Points', fontsize=16)
    plt.grid(True)

    # Drawing a history of centroid movement, first centroid is black
    tempx, tempy = [], []
    for mycentroid in centroids:
        tempx.append(mycentroid[:, 0])
        tempy.append(mycentroid[:, 1])

    plt.plot(tempx, tempy, 'rx--', markersize=8)
    plt.plot(tempx[0], tempy[0], 'kx', markersize=8)

    plt.legend(loc=4, framealpha=0.5)
    plt.show(block=True)
```

In [ ]:
```
datafile = 'data/data2.mat'
mat = scipy.io.loadmat(datafile)
samples = mat['X']
# samples contain 300 pts, each has two coordinates
print(samples.shape)

# Choose the initial centroids
initial_centroids = np.array([[3, 3], [6, 2], [8, 5]])
plot_data(samples, [initial_centroids])
```

```
(300, 2)
```



1.3 Finding closest centroids [20 pts]

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 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 clusters) 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*, run it and you should see the output $[0, 2, 1]$ corresponding to the centroid assignments for the first 3 examples.

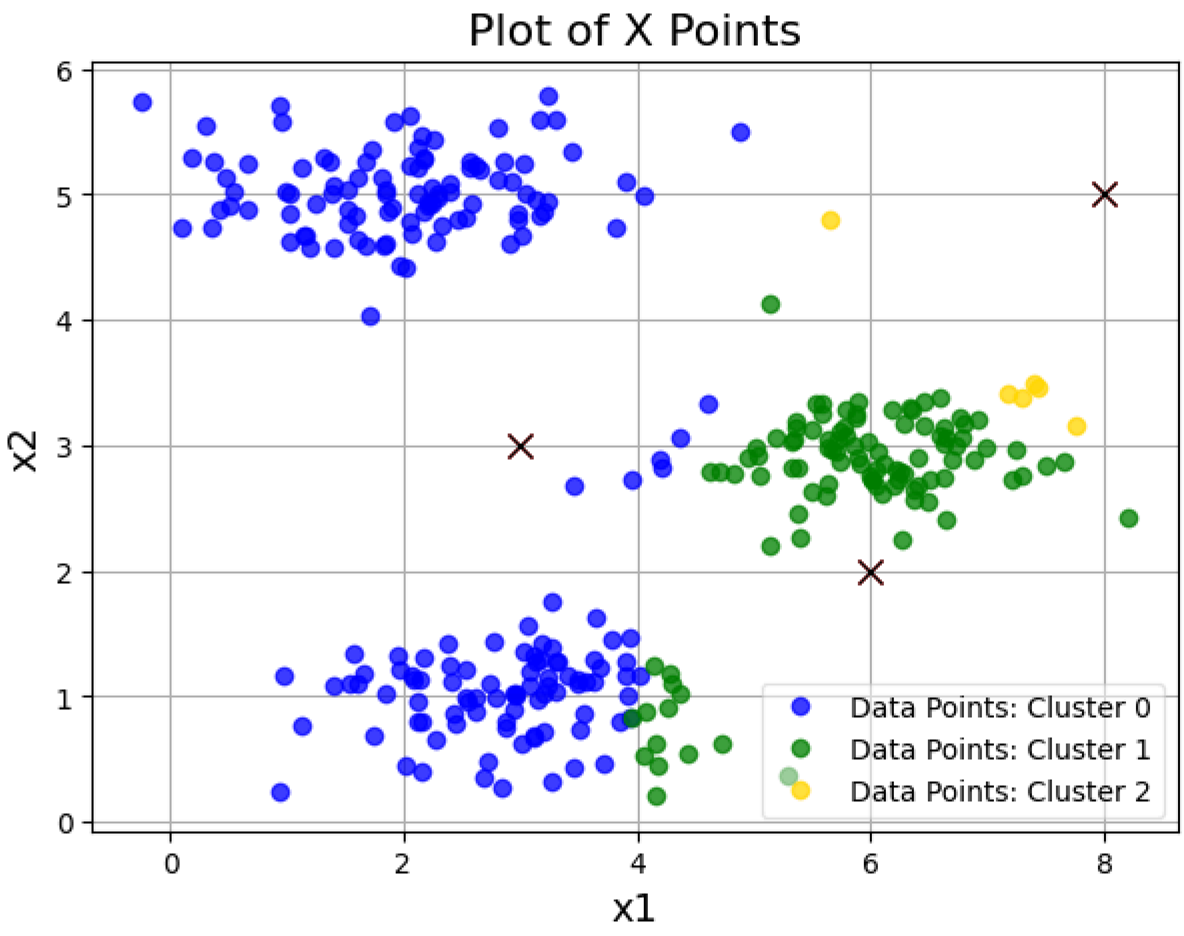
```
In [ ]: def find_closest_centroids(samples, centroids):
            """
            Find the closest centroid for all samples.
```

```
    :param samples: input data samples.
    :param centroids: an array of cluster centroids.
    :return: a list of cluster assignments (indices) for each sample.
    """
    clusters = np.zeros((samples.shape[0], 1))
    for sample_idx, sample in enumerate(samples):
        # reset distance between a sample point and a cluster to infinte pos
        distance = float('inf')
        closest_cluster = -1

        # Retrieval each centroid and get the closester cluster index
        for cluster_index, centroid in enumerate(centroids):
            dist = np.linalg.norm(sample - centroid)

            if dist < distance:
                distance = dist
                closest_cluster = cluster_index

        clusters[sample_idx] = closest_cluster

    return clusters
```

In [ ]:

```
clusters = find_closest_centroids(samples, initial_centroids)
# you should see the output [0, 2, 1] corresponding to the
# centroid assignments for the first 3 examples.
print(clusters[:3].flatten())
plot_data(samples, [initial_centroids], clusters)
```

```
[0. 2. 1.]
```



1.4 Computing centroid means [20 pts]

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

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

where C_k is the set of examples that are assigned to centroid k. Concretely, if only two examples say x_3 and x_5 are assigned to centroid $k = 2$, then you should update

$$\mu_2 = \frac{1}{2}(x_3 + x_5).$$

Complete the code in *get_centroids* to do this.

In [ ]:
```
def get_centroids(samples, clusters):
    """
    Find the new centroid (mean) given the samples and their cluster.

    :param samples: samples.
    :param clusters: list of clusters corresponding to each sample.
    :return: an array of centroids.
    """
    unique_keys = np.unique(clusters) # get the number of clusters
    K = len(unique_keys)
    D = samples.shape[1]

    centroids =  np.zeros((K, D))
    num = np.zeros(K)

    for i in range(len(samples)):
        cluster_idx = np.where(unique_keys == clusters[i])[0][0]  # 클러스터 인
        centroids[cluster_idx] += samples[i]
        num[cluster_idx] += 1

    num[num == 0] = 1
    centroids /= num[:, np.newaxis]

    # centroids = np.array([samples[clusters == k].mean(axis = 0) for k in u
    return centroids
```

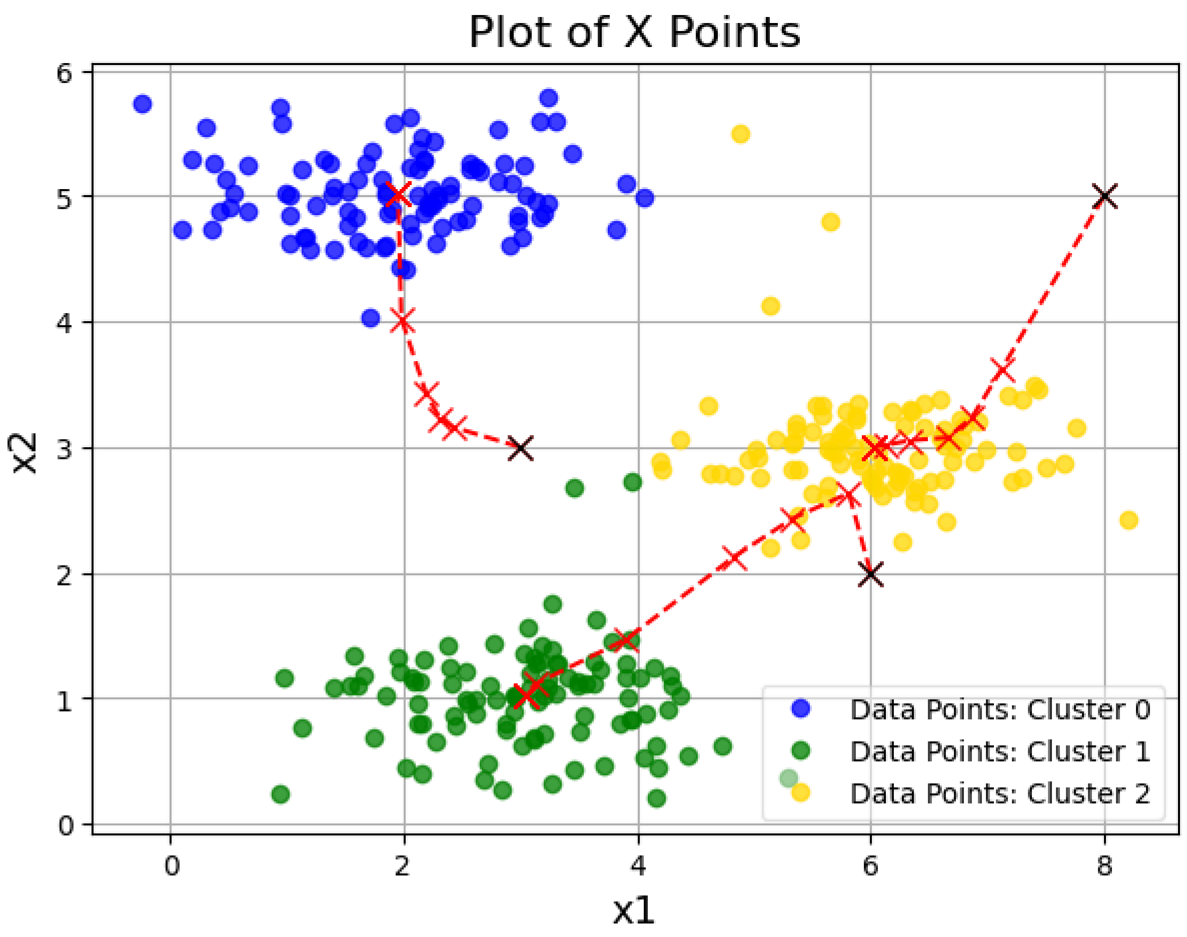
Once you have completed it, *run_k_means* below will run your code and output the centroids after the first step of K-means. The final centroids should be roughly [[1.9 5.0] [3.0 1.0] [6.0 3.0]].

The code below will produce a visualization that plots the progress of the algorithm at each iteration. The red crosses show how each step of the K-means algorithm changes the centroids. You should also see the final cluster assignments as color-coded points.

In [ ]:
```
def run_k_means(samples, initial_centroids, n_iter):
    """
    Run K-means algorithm. The number of clusters 'K' is defined by the size
    :param samples: samples.
    :param initial_centroids: a list of initial centroids.
    :param n_iter: number of iterations.
    :return: a pair of cluster assignment and history of centroids.
    """

    centroid_history = []
    current_centroids = initial_centroids
    clusters = []
    for iteration in range(n_iter):
        centroid_history.append(current_centroids)
        print("Iteration %d, Finding centroids for all samples..." % iterati
        clusters = find_closest_centroids(samples, current_centroids)
        print("Recompute centroids...")
        current_centroids = get_centroids(samples, clusters)

    return clusters, centroid_history
```

In [ ]:
```
# Run the full K-means algorithm and plot the results
clusters, centroid_history = run_k_means(samples, initial_centroids, n_iter=
plot_data(samples, centroid_history, clusters)
```

```
Iteration 0, Finding centroids for all samples...
Recompute centroids...
Iteration 1, Finding centroids for all samples...
Recompute centroids...
Iteration 2, Finding centroids for all samples...
Recompute centroids...
Iteration 3, Finding centroids for all samples...
Recompute centroids...
Iteration 4, Finding centroids for all samples...
Recompute centroids...
Iteration 5, Finding centroids for all samples...
Recompute centroids...
Iteration 6, Finding centroids for all samples...
Recompute centroids...
Iteration 7, Finding centroids for all samples...
Recompute centroids...
Iteration 8, Finding centroids for all samples...
Recompute centroids...
Iteration 9, Finding centroids for all samples...
Recompute centroids...
```



1.5 Random initialization [10 pts]

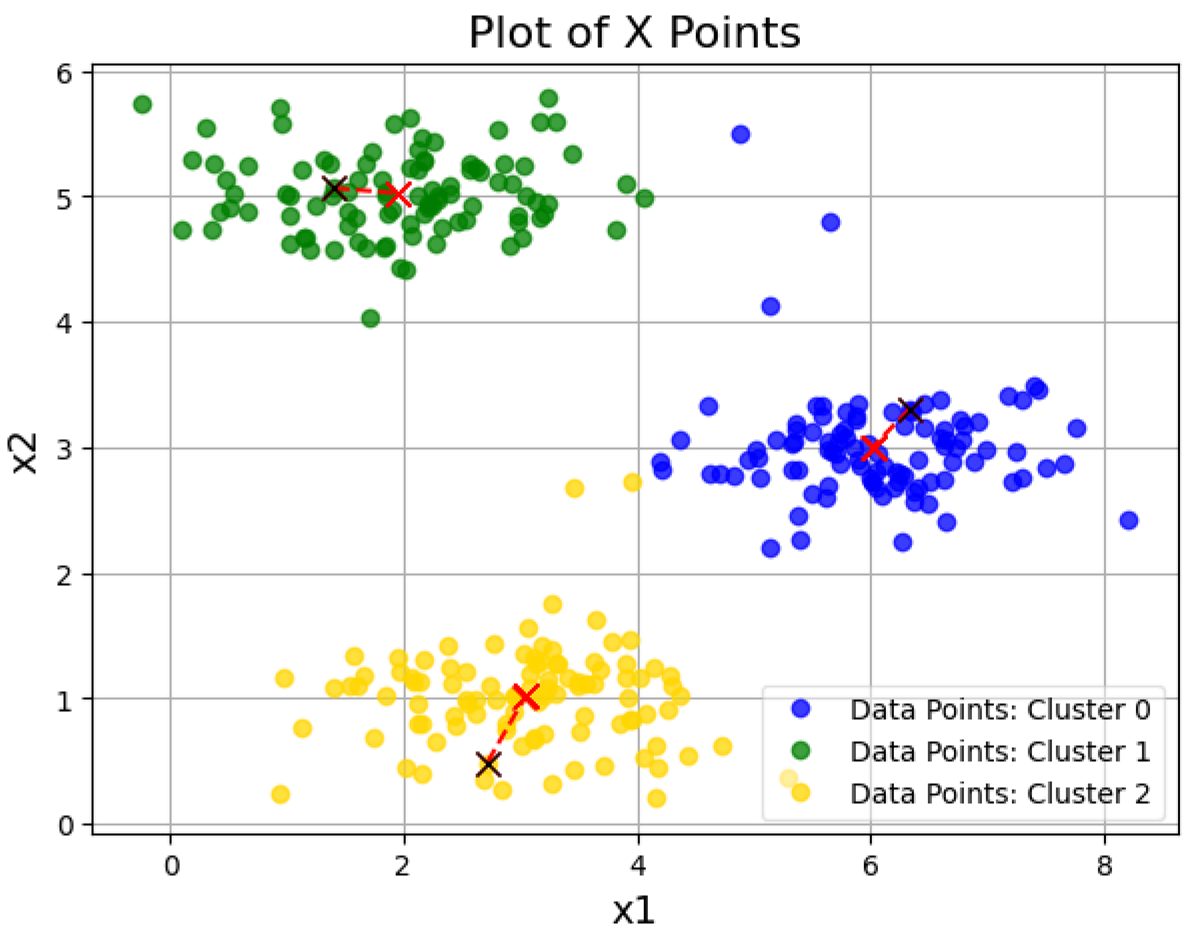
The initial assignments of centroids for the example dataset in the previous section were designed so that you could test your algorithm, i.e., if you implemented it correctly you would see the same output as we provided. In practice, a good strategy for initializing the centroids is to select random 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, select 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 the random initialization affects the first few iterations of clustering, and also possibly, results in a different cluster assignment.

In [ ]:
```
def choose_random_centroids(samples, K):
    """
    Randomly choose K centroids from samples.
    :param samples: samples.
    :param K: K as in K-means. Number of clusters.
    :return: an array of centroids.
    """
    np.random.seed(7)   # Use random seed 7
    shuffled_indices = np.random.permutation(samples.shape[0]) # random Perm
    random_selected_centroids = samples[shuffled_indices[:K]] # Select the f
    return random_selected_centroids
```

In [ ]:
```
# Let's choose random initial centroids and see the resulting
# centroid progression plot.. run it a few times, see if you can
# find a particularly bad case (no need to report it)
clusters, centroid_history = run_k_means(samples, choose_random_centroids(sa
plot_data(samples, centroid_history, clusters)
```

```
Iteration 0, Finding centroids for all samples...
Recompute centroids...
Iteration 1, Finding centroids for all samples...
Recompute centroids...
Iteration 2, Finding centroids for all samples...
Recompute centroids...
Iteration 3, Finding centroids for all samples...
Recompute centroids...
Iteration 4, Finding centroids for all samples...
Recompute centroids...
Iteration 5, Finding centroids for all samples...
Recompute centroids...
Iteration 6, Finding centroids for all samples...
Recompute centroids...
Iteration 7, Finding centroids for all samples...
Recompute centroids...
Iteration 8, Finding centroids for all samples...
Recompute centroids...
Iteration 9, Finding centroids for all samples...
Recompute centroids...
```



1.6 Deriving Mixture of Gaussians [20 pts]

(Problem 9.3 in Bishop.)

Consider a Gaussian mixture model in which the K-dimensional one-hot latent variable z represents the component and has marginal distribution

$$p(z) = \prod_{k=1}^{K} \pi_k^{z_k}$$

and the conditional distribution $p(x|z)$ for the observed variable is given by

$$\prod_{k=1}^{K} \mathcal{N}(x|\mu_k, \Sigma_k)^{z_k}$$

where μ_k, Σ_k are the parameters of the k th Gaussian distribution. Show that the marginal distribution $p(x)$ obtained by marginalizing over all possible values of z is a Gaussian mixture of the form

$$\sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

Based on the marginal and conditional distribution $p(z)$ and $P(x|z)$, I need to show $p(x)$. $p(x)$ is the marginal distribution. $p(x) = \sum_z p(z)p(x|z)$. The latent variable z is the one-hot encoded vector, so I need to iterate sum for all cluster K.

$$p(x) = \sum_{k=1}^{K} p(z_k = 1) \cdot p(x|z_k = 1)$$

Now plug in the distributions that the question defined.

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

1.7 K-means vs GMM [20 pts]

(a) Compare and contrast the two methods.

(b) Describe in detail how you would modify the K-means algorithm to make it into a Gaussian mixture algorithm, and vice versa.

(a) First of all, the K-means algorithm is a hard cluster assignment method, while the Gaussian Mixture Model (GMM) is a probabilistic approach to soft cluster assignment. In GMM, each data point has a probability of belonging to each cluster. K-means clusters each data point into only one cluster based on the Euclidean distance. It iteratively updates the centroid points based on the newly assigned data points and reassigns the clusters.

On the other hand, GMM uses a mixture of Gaussian distributions to determine the likelihood of each data point being assigned to a specific cluster. This allows GMM to handle elliptical clusters and capture overlapping or non-spherical shapes more effectively than K-means.

(b) To transform K-means into a Gaussian Mixture Model (GMM), GMM need to use probabilistic reasoning into the clustering process. First, GMM replace the hard assignments in K-means with soft assignments by computing the probability (responsibility) that each data point belongs to each cluster. This is achieved using the Gaussian probability density function, which is parameterized by cluster-specific means and covariance matrices. Instead of minimizing Euclidean distance, GMM employ the Expectation-Maximization (EM) algorithm. But EM algorithm is basically similar to K mean reassignment steps.

In the E-step, GMM calculate the responsibilities, denoted as $\gamma(z_{nk})$, which represent the probability that data point x_n was generated by cluster k. In the M-step, we use these responsibilities to re-estimate the model parameters, including the means (μ_k), covariances (Σ_k), and mixing coefficients (π_k).

This transformation converts the simple distance-based clustering into a rich probabilistic framework.

1.8 [BONUS] Image compression with K-means [20 pts]

In this OPTIONAL exercise, you will apply your implemented K-means algorithm 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. This 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.

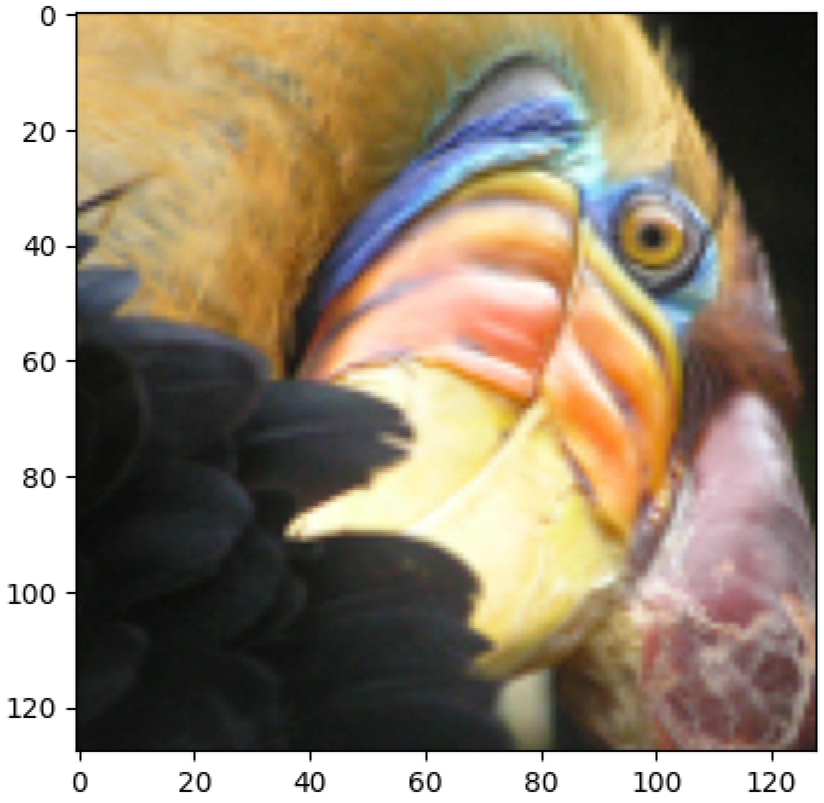
By making this reduction, it is possible to represent the photo in an efficient way (compressed). Specifically, you only need to store the RGB values of the 16 selected colors, and for each pixel in the image you now need to only store the index of the color at that location (where only 4 bits are necessary 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.

The code below 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, 2)* gives the blue intensity of the pixel at row 50 and column 33.

In [ ]:
```
from __future__ import absolute_import

import matplotlib.pyplot as plt
import numpy as np
import scipy
import imageio.v2 as imageio
# import imageio

datafile = 'data/bird_small.png'
# This creates a three-dimensional matrix bird_small whose first two indices
# identify a pixel position and whose last index represents red, green, or b
#bird_small = scipy.misc.imread(datafile)
bird_small = imageio.imread(datafile)

print("bird_small shape is ", bird_small.shape)
plt.imshow(bird_small)
plt.show()
```

```
bird_small shape is  (128, 128, 3)
```



Write code below to call your K-means function to cluster the pixels colors in the image into K clusters.

After finding the top K = 16 colors to represent the image, 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 of the 128×128 pixel locations, resulting in a total of $128 \times 128 \times 24 = 393,216$ bits.

The new representation requires some overhead storage in the form of a dictionary of 16 colors, each of which require 24 bits, but the image itself now 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.

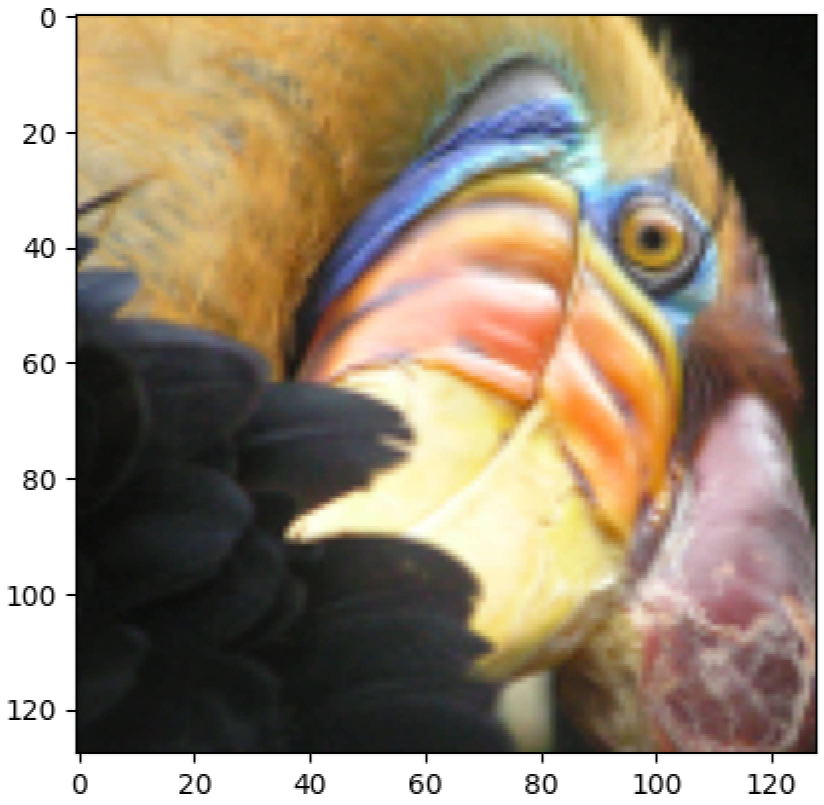
Finally, show the effect of the compression by reconstructing the image based only on the centroid assignments.

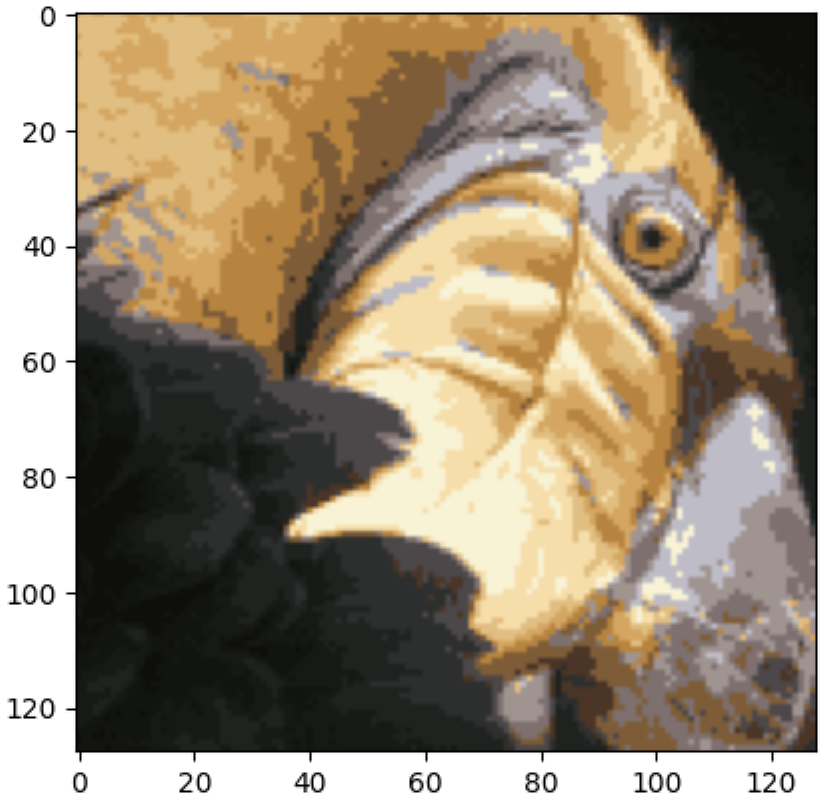
In [ ]:
```
# Divide every entry in bird_small by 255 so all values are in the range of
bird_small = bird_small / 255.

# Unroll the image to shape (16384,3) (16384 is 128*128)
bird_small = bird_small.reshape(-1, 3)

# Run k-means on this data, forming 16 clusters, with random initialization
```

```
random_initiation = choose_random_centroids(bird_small, 16) # random initial
clusters, centroid_history = run_k_means(bird_small, random_initiation, 10)
final_centroids = centroid_history[-1]

# Now we have 16 centroids, each representing a color.
# Let's assign an index to each pixel in the original image dictating
# which of the 16 colors it should be. Use your find_closest_centroids
# function and store the result in "clusters"
clusters = find_closest_centroids(bird_small, final_centroids)

# Now loop through the original image and form a new image
# that only has 16 colors in it
final_image = np.zeros((clusters.shape[0], 3)) # 128 x 128 = 16384 | (16384,
for i in range(len(bird_small)):
    final_image[i] = final_centroids[int(clusters[i])]

# multiply every entry in bird_small by 255
bird_small = (bird_small * 255).astype(np.uint8)
final_image = (final_image * 255).astype(np.uint8)

# Reshape the original image and the new, final image and draw them
# To see what the "compressed" image looks like
plt.figure()
plt.imshow(bird_small.reshape(128, 128, 3))
plt.figure()
plt.imshow(final_image.reshape(128, 128, 3))
plt.show()
```

```
Iteration 0, Finding centroids for all samples...
Recompute centroids...
Iteration 1, Finding centroids for all samples...
Recompute centroids...
Iteration 2, Finding centroids for all samples...
Recompute centroids...
Iteration 3, Finding centroids for all samples...
Recompute centroids...
Iteration 4, Finding centroids for all samples...
Recompute centroids...
Iteration 5, Finding centroids for all samples...
Recompute centroids...
Iteration 6, Finding centroids for all samples...
Recompute centroids...
Iteration 7, Finding centroids for all samples...
Recompute centroids...
Iteration 8, Finding centroids for all samples...
Recompute centroids...
Iteration 9, Finding centroids for all samples...
Recompute centroids...
```

```
/var/folders/xt/y_17jz6n3p77v5p6cbm1tstr0000gn/T/ipykernel_43640/2283966473.
py:22: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar
is deprecated, and will error in future. Ensure you extract a single element
from your array before performing this operation. (Deprecated NumPy 1.25.)
  final_image[i] = final_centroids[int(clusters[i])]
```





2. Principal Components Analysis

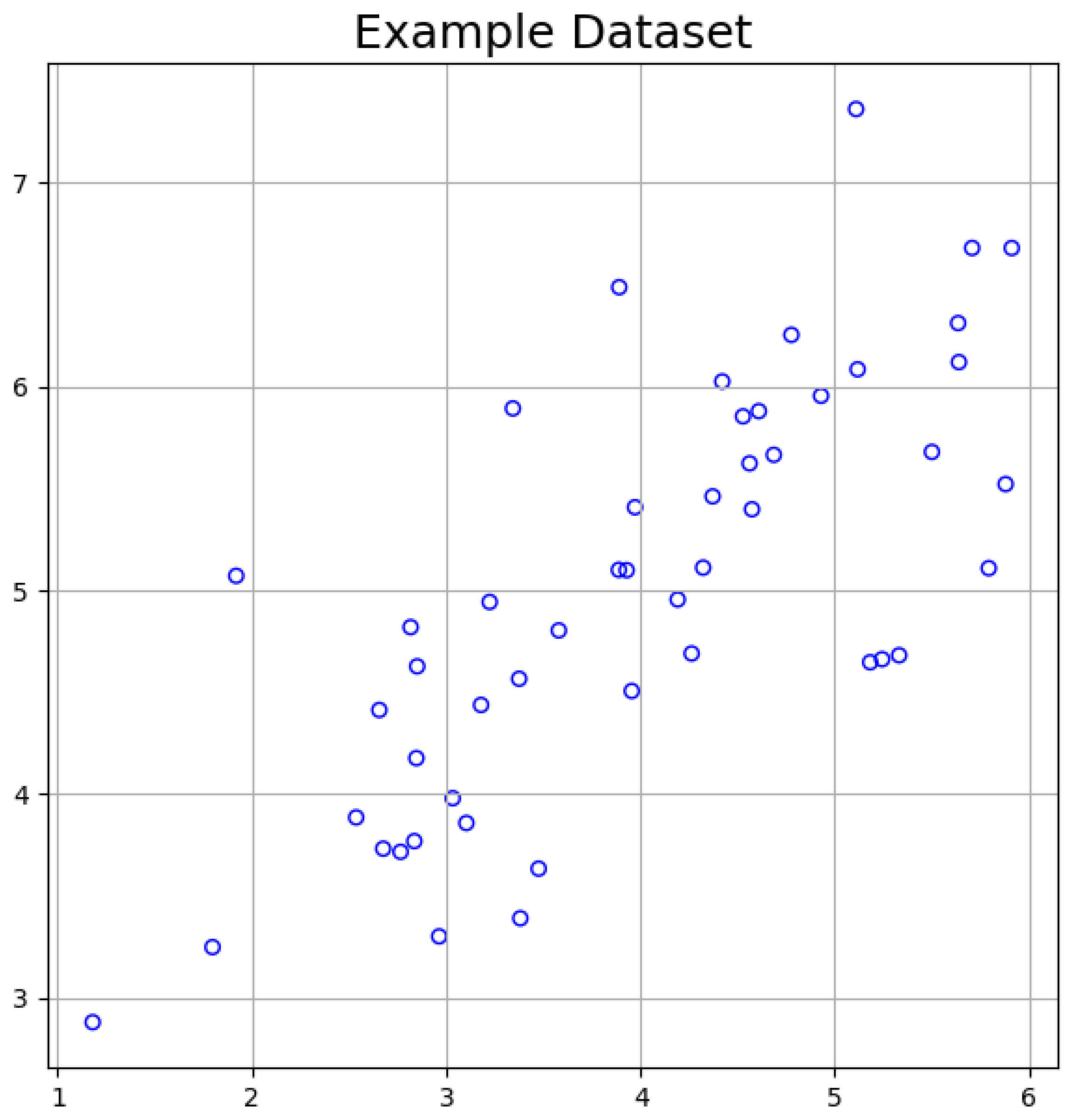
In this exercise, you will use principal component analysis (PCA) to perform dimensionality reduction. You will first experiment with an example 2D dataset to get intuition on how PCA works, and then use it on a bigger dataset of 5000 face image dataset. We will help you step through the first half of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

2.1 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 code below will plot the training data.

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 high dimensional data (such as from 256 to 50 dimensions), but using lower dimensional data in this example allows us to better visualize the algorithm.

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import scipy
import scipy.io

datafile = 'data/data1.mat'
mat = scipy.io.loadmat(datafile)
samples = mat['X']

plt.figure(figsize=(7, 7))
plt.scatter(samples[:, 0], samples[:, 1], s=30, facecolors='none', edgecolor
plt.title("Example Dataset", fontsize=18)
plt.grid(True)
plt.show()
```



2.2 Implementing PCA [20 pts]

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 SVD (the 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. Normalize each feature by subtracting the mean value (to have zero mean), and scaling (to have unit variance) so that they are in the same range.

After normalizing the data, your task is to complete the code 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 here Σ refers to 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.

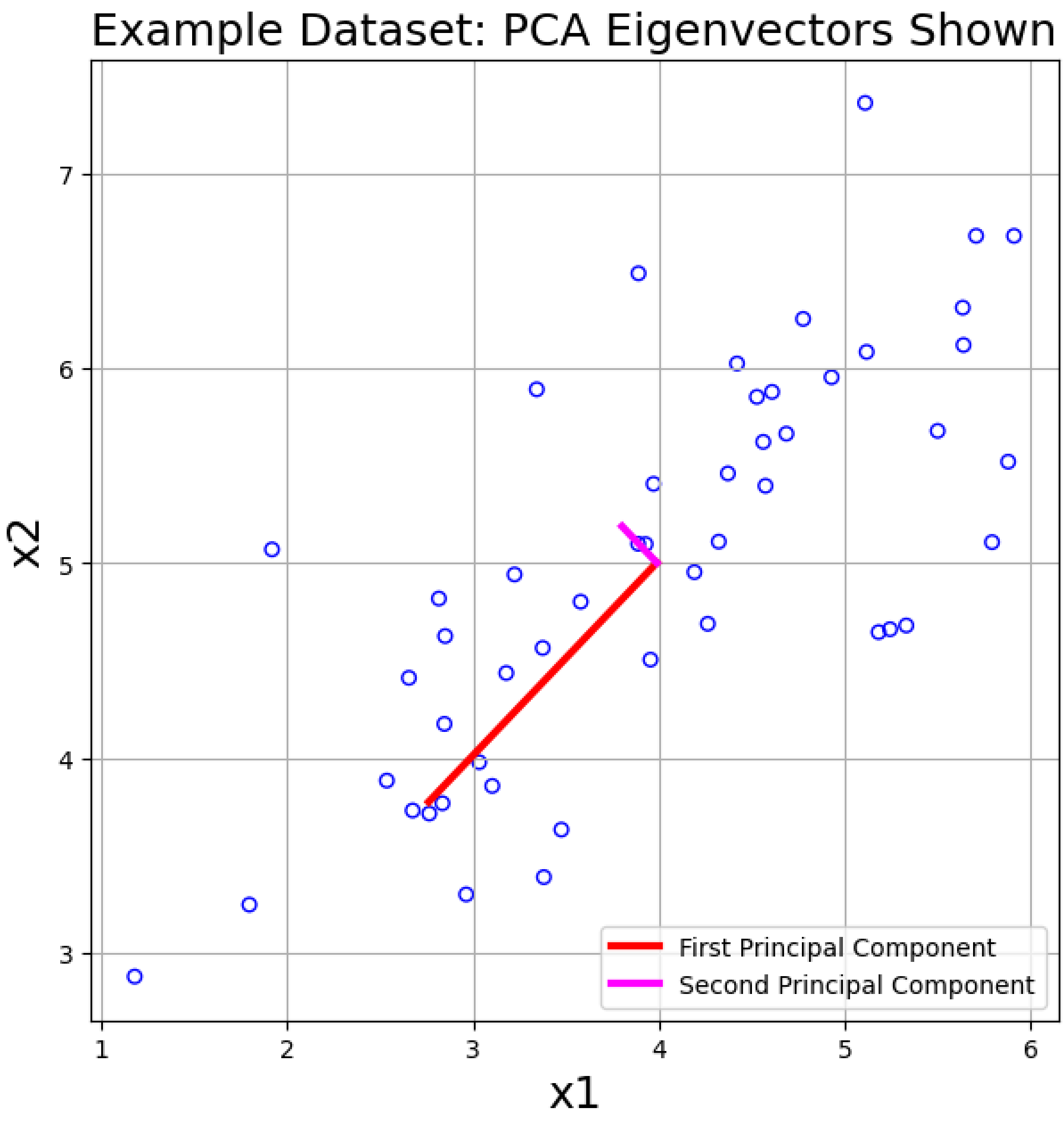
Once you have completed the code, the script below will run PCA on the example dataset and plot the corresponding principal components found. The script will also output the top principal component (eigenvector) found, which should be around [-0.707 -0.707].

In [ ]:
```
import scipy.linalg

def feature_normalize(samples):
    """
    Feature-normalize samples
    :param samples: samples.
    :return: normalized feature
    """
    sample_mean = np.mean(samples, axis=0)
    sample_std = np.std(samples, axis=0)
    samples_norm = (samples - sample_mean) / sample_std
    return sample_mean, sample_std, samples_norm


def get_usv(sample_norm):
    # Compute the covariance matrix
    # Run single value decomposition to get the U principal component matrix
    length = sample_norm.shape[0]
    covariance_matrix = (sample_norm.T @ sample_norm) / length
    U, s, Vh = scipy.linalg.svd(covariance_matrix)
    return U, s, Vh
```

In [ ]:
```
# Feature normalize
means, stds, samples_norm = feature_normalize(samples)

# Run SVD
U, S, V = get_usv(samples_norm)

# output the top principal component (eigen- vector) found
# should expect to see an output of about [-0.707 -0.707]"
print('Top principal component is ', U[:, 0])

plt.figure(figsize=(7, 7))
plt.scatter(samples[:, 0], samples[:, 1], s=30, facecolors='none', edgecolor
plt.title("Example Dataset: PCA Eigenvectors Shown", fontsize=18)
plt.xlabel('x1', fontsize=18)
plt.ylabel('x2', fontsize=18)
plt.grid(True)
# To draw the principal component, you draw them starting
# at the mean of the data
```

```
plt.plot([means[0], means[0] + S[0] * U[0, 0]],
         [means[1], means[1] + S[0] * U[1, 0]],
         color='red', linewidth=3,
         label='First Principal Component')
plt.plot([means[0], means[0] + S[1] * U[0, 1]],
         [means[1], means[1] + S[1] * U[1, 1]],
         color='fuchsia', linewidth=3,
         label='Second Principal Component')
plt.legend(loc=4)
plt.show()
```

```
Top principal component is  [-0.70710678 -0.70710678]
```



2.3 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.

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

You should now complete the code in *project_data(samples, U, K)*. 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 $reduced\_U = U(:, 1:K)$.

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).

```
In [ ]: def project_data(samples, U, K):
            """
            Computes the reduced data representation when
            projecting only on to the top "K" eigenvectors
            """
            # Reduced U is the first "K" columns in U
            reducted_U = U[:, :K]
            z = samples @ reducted_U
            return z

        # project the first example onto the first dimension
        # should see a value of about 1.481"
        z = project_data(samples_norm, U, 1)
        print(samples_norm[0])
        print('Projection of the first example is %0.3f.' % float(z[0]))
```

```
[-0.52331306 -1.59279252]
Projection of the first example is 1.496.
/var/folders/xt/y_17jz6n3p77v5p6cbm1tstr0000gn/T/ipykernel_15804/3511652997.
py:15: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar
is deprecated, and will error in future. Ensure you extract a single element
from your array before performing this operation. (Deprecated NumPy 1.25.)
  print('Projection of the first example is %0.3f.' % float(z[0]))
```

2.3.2 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(Z, U, K)* 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].

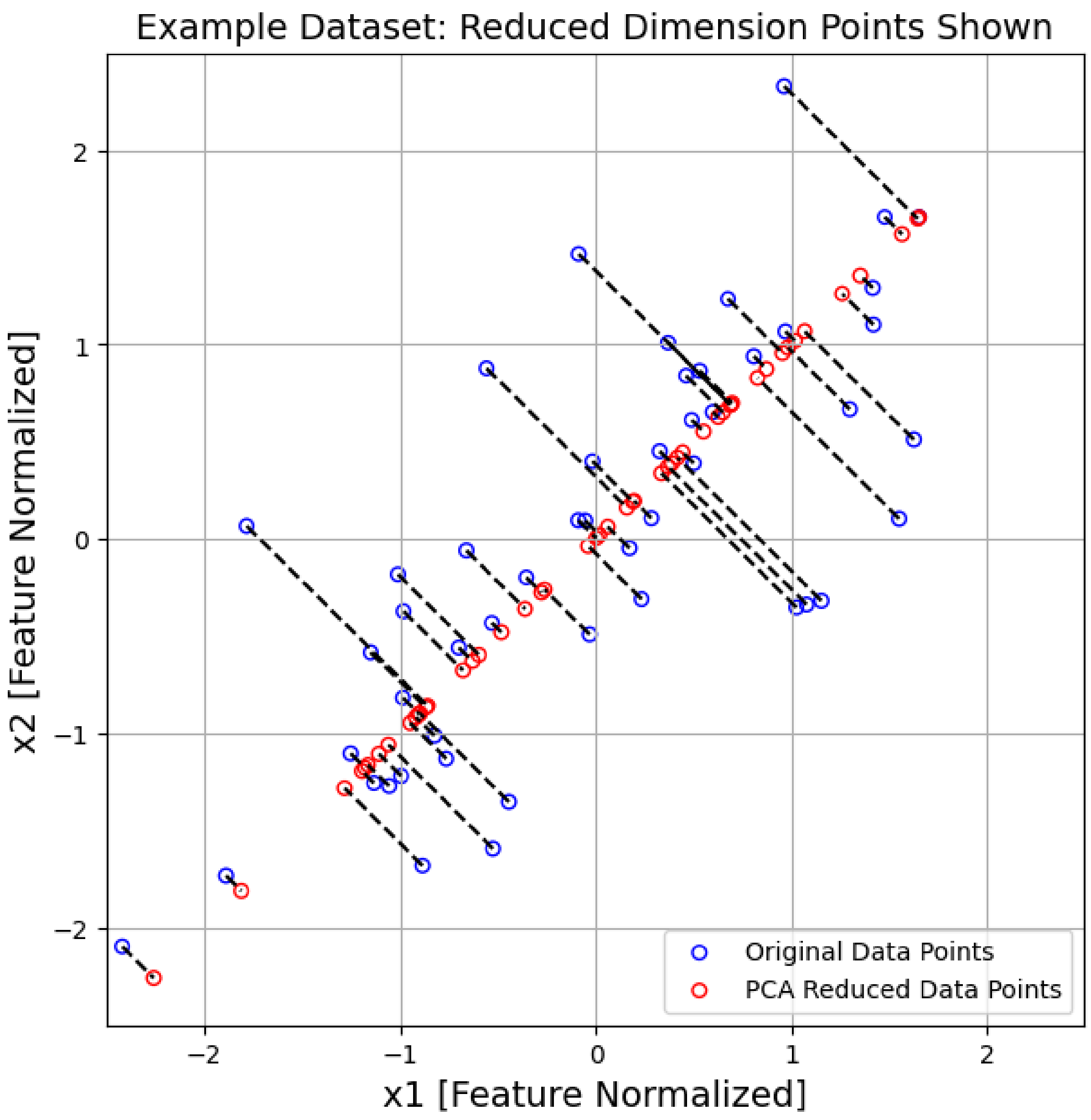
In [ ]:
```
def recover_data(Z, U, K):
    reduced_U = U[:, :K]
    recover = Z @ reduced_U.T
    return recover


# reconstruct the samples
recovered_sample = recover_data(z, U, 1)
print('Recovered approximation of the first example is ', recovered_sample[0
```

```
Recovered approximation of the first example is  [-1.05805279 -1.05805279]
```

2.3.3 Visualizing the projections [10 pts]

After completing both *project_data* and *recover_data*, the script below should now perform both the projection and approximate reconstruction to show how the projection affects the data. In the plot, 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. If the plot looks correct, you get full points for this part.

In [ ]:
```
plt.figure(figsize=(7, 7))
plt.scatter(samples_norm[:, 0], samples_norm[:, 1], s=30, facecolors='none',
            edgecolors='b', label='Original Data Points')
plt.scatter(recovered_sample[:, 0], recovered_sample[:, 1], s=30, facecolors
            edgecolors='r', label='PCA Reduced Data Points')

plt.title("Example Dataset: Reduced Dimension Points Shown", fontsize=14)
plt.xlabel('x1 [Feature Normalized]', fontsize=14)
plt.ylabel('x2 [Feature Normalized]', fontsize=14)
plt.grid(True)

for x in range(samples_norm.shape[0]):
    plt.plot([samples_norm[x, 0], recovered_sample[x, 0]], [samples_norm[x,

plt.legend(loc=4)
plt.xlim((-2.5, 2.5))
plt.ylim((-2.5, 2.5))
plt.show()
```



2.4 Deriving PCA [20 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. Eqn (12.12)

$$z_{nj} = \mathrm{x}_n^{\mathrm{T}} \mathrm{u}_j$$

2. Eqn (12.13)

$$b_j = \bar{\mathrm{x}}^{\mathrm{T}} \mathrm{u}_j$$

3. In case of two-dimensional data space

$$\mathrm{S}\mathrm{u}_2 = \lambda_2 \mathrm{u}_2$$

$$J = \lambda_2$$

Let's explain Derivigin PCA through Minimum-error formulation of PCA. From the textbooks's part 12.7 it starts explaining Minimum-error formulation.

(12.7) Let’s consider a complete orthonormal set of D-dimensional basis vectors u_i for $i = 1, \cdots, D$, which satisfy the condition:

$$u_i^T u_j = \delta_{ij} \quad (12.7)$$

To make this clearer, let’s take a simple example. Suppose we have two orthonormal vectors u_i and u_j such that $\delta_{ij} = 1$ when $i = j$, and 0 otherwise. For instance, let

$$u_i = [\,1\ 0\,], \quad u_j = [\,0\ 1\,]$$

Then,

$$u_i^T u_i = 1, \quad u_j^T u_j = 1, \quad u_i^T u_j = 0$$

This demonstrates that u_i and u_j are orthonormal.

(12.8) Each data point x_n can be expressed as a linear combination of the basis vectors:

$$x_n = \sum_{i=1}^{D} a_{ni} u_i \quad (12.8)$$

Here, a_{ni} is the scalar coefficient representing how much of u_i contributes to x_n.

For example, consider a data point

$$x = [\,3\ 4\,]$$

Using the orthonormal basis vectors

$$u_1 = [\,1\ 0\,], \quad u_2 = [\,0\ 1\,],$$

I can decompose x as:

$$x = a_1 u_1 + a_2 u_2 = 3 \cdot [\,1\ 0\,] + 4 \cdot [\,0\ 1\,] = [\,3\ 4\,]$$

This illustrates how x is formed from its projections onto the basis vectors.

(12.9)

$$x_n = \sum_{i=1}^{D}(x_n^T u_i)u_i \quad (12.9)$$

Since I established that $a_{ni} = x_n^T u_i$, I can rewrite equation (12.8) as (12.9).

Using the earlier example:

$$x = [\,3\;4\,], \quad u_1 = [\,1\;0\,], \quad u_2 = [\,0\;1\,]$$

I get:

$$x^T u_1 = 3, \quad x^T u_2 = 4$$

Therefore,

$$x = (x^T u_1)u_1 + (x^T u_2)u_2 = 3 \cdot [\,1\;0\,] + 4 \cdot [\,0\;1\,] = [\,3\;4\,]$$

This confirms that:

$$\alpha_{nj} = x_n^T u_j$$

(12.10) The goal of PCA is to reduce the dimensionality by selecting M basis vectors, where $M < D$. Using only the first M basis vectors, we can approximate x_n as:

$$\tilde{x}n = \sum i = 1^M z_{ni} u_i + \sum_{i=M+1}^{D} b_i u_i$$

Here, $z_{ni} = x_n^T u_i$ represents the projection of x_n onto the first M basis vectors. The terms b_i are fixed constants shared across all data points and represent the contribution from the remaining $D - M$ basis vectors.

Now, the task becomes choosing the optimal set of basis vectors u_i and the constants b_i to best approximate the original data using only M dimensions.

(12.11) To measure how well the approximation $\tilde{x}n$ reconstructs the original data point x_n, we define a cost function based on the mean squared reconstruction error:

$$J = \frac{1}{N} \sum n = 1^N |\mathbf{x}_n - \tilde{\mathbf{x}}_n|^2$$

So what is the goal of PCA?

To minimize the squared error between the original data points and their

To begin the optimization, I take the derivative of this cost function with respect to z_{ni} (the projection coefficients for each data point onto the chosen basis vectors). By substituting $\tilde{x}n = \sum i = 1^M z_{ni} u_i + \sum_{i=M+1}^{D} b_i u_i$ into the cost function and setting the derivative $\frac{\partial J}{\partial z_{ni}}$ to zero, we can solve for the optimal z_{ni}.

(12.12) This leads me to the result:

$$z_{ni} = \mathbf{x}_n^T \mathbf{u}_i$$

which shows that the best projection of x_n onto the basis vector u_i is given by the inner product $x_n^T u_i$ — exactly as I saw earlier.

(12.13) Next, I minimize the cost function J with respect to the constants b_j (for $j = M+1, \dots, D$), which are shared across all data points.

Taking the derivative of J with respect to b_j, and using the orthonormality of the basis vectors u_i.

Now expand the error term:

$$\mathbf{x}_n - \tilde{\mathbf{x_n}} = \mathbf{x_n} - \sum_{i=1}^{M} z_{ni}\mathbf{u_i} - \sum_{j=M+1}^{D} b_j \mathbf{u_j}$$

To find the optimal constants b_j, I take the derivative of the cost function J with respect to b_j:

$$\frac{\partial J}{\partial b_j} = \frac{1}{N}\sum_{n=1}^{N} 2(\mathbf{x}_n - \tilde{\mathbf{x}}_n)^\top(-\mathbf{u}_j)$$

Setting the derivative to zero:

$$\sum_{n=1}^{N}\left(\mathbf{x_n} - \sum_{i=1}^{M} z_{ni}\mathbf{u_i} - \sum_{k=M+1}^{D} b_k \mathbf{u_k}\right)^\top \mathbf{u_j} = 0$$

Using the orthonormality of the basis vectors (i.e., $\mathbf{u}_i^\top \mathbf{u} j = \delta ij$), all terms orthogonal to $\mathbf{u}_j$ vanish, and I have left with:

$$\sum_{n=1}^{N}\left(\mathbf{x_n}^\top \mathbf{u}_j - b_j\right) = 0$$

Solving for b_j:

$$b_j = \frac{1}{N}\sum_{n=1}^{N} \mathbf{x_n}^\top \mathbf{u_j}$$

This is the projection of the mean vector $\bar{\mathbf{x}}$ onto the basis vector $\mathbf{u}_j$, so we can write:

$$b_j = \bar{\mathbf{x}}^\top \mathbf{u_j}$$

This shows that the optimal b_j is the component of the data mean in the direction of $\mathbf{u_j}$.

(12.14) Now replace z_{ni} and b_j with x_n^T and $x_n^T u_j$, I get

$$x_n - \tilde{x_n} = \sum_{i=M+1}^{D} \{(x_n - \bar{x_n})^T u_i\} u_i$$

(12.15) Expand this equation and plug back into the cost function J.

$$J = \frac{1}{N} \sum_{n=1}^{N} (\sum_{i=m+1}^{D} (x_n^T u_i - \bar{x}^T u_i)^2) = \sum_{i=M+1}^{D} u_i^T S u_i$$

From the Lagrange Multiplier equation to enforce the constraint.

$$\bar{J} = u_i^T S u_i + \lambda_i (1 - u_i^T u_1)$$

Derive this respect to u_i. Obtain $Su_2 = \lambda_i u_i$, using the eigenvector traits so multiplying u_i^T both side

$$u_i^T S u_i = \lambda_i$$

So where M = 1 and total dimension is 2. The total Cost is λ_2

2.5 [BONUS] Face Image Dataset [20 pts]

In this part of the exercise, you will run your PCA algorithm on face images to see how it can be used in practice for dimension reduction. The file *faces.mat* contains a dataset X of face images, each 32×32 in grayscale. Each row of X corresponds to one face image vectorized into a row vector of length 1024. The next step will load and visualize the first 100 of these face images.

2.5.0 Visualizing the data

To display each row of X as an image, we need to first reshape it into a 32x32 array. Implement the *get_datum_img* function below and run the visualization code.

In [ ]:
```
import pylab

def get_datum_img(row):
    """
    Creates an image from a single np array with shape 1x1024
    :param row: a single np array with shape 1x1024
    :return: the constructed image, np array of shape 32 x 32
    """
    image = row.ravel().reshape(32, 32, order='F') # column-major order (for
    return image


def display_data(samples, num_rows=10, num_columns=10):
    """
    Function that picks the first 100 rows from X, creates an image from eac
    then stitches them together into a 10x10 grid of images, and shows it.
    """
    width, height = 32, 32
    num_rows, num_columns = num_rows, num_columns

    big_picture = np.zeros((height * num_rows, width * num_columns))

    row, column = 0, 0
    for index in range(num_rows * num_columns):
        if column == num_columns:
            row += 1
            column = 0
        img = get_datum_img(samples[index])
        big_picture[row * height:row * height + img.shape[0], column * widtl
        column += 1
    plt.figure(figsize=(10, 10))
    plt.imshow(big_picture, cmap=pylab.gray())
    plt.show()
```

In [ ]:

```
datafile = 'data/faces.mat'
mat = scipy.io.loadmat(datafile)
samples = mat['X']
display_data(samples)
```

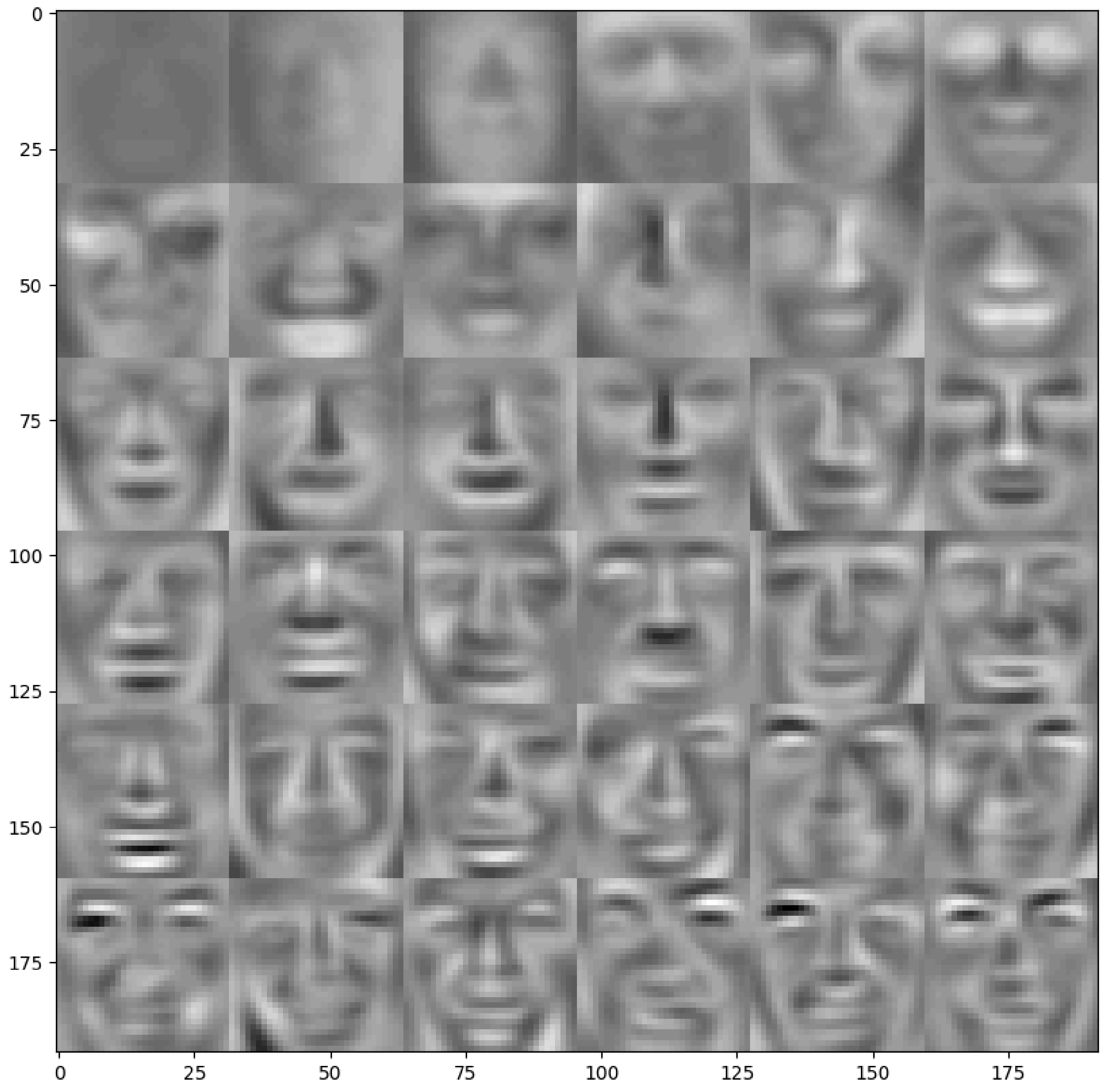
2.5.1 PCA on Faces

To run PCA on the face dataset, we first normalize the dataset by normalizing each feature from the data matrix X to have zero mean and unit variance.

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 (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 code cell below displays the first 36 principal components that describe the largest variations. If you want, you can also change the code to display more principal components to see how they capture more and more details.

In [ ]:
```
# Feature normalize
means, stds, samples_norm = feature_normalize(samples)

# Run SVD
```

```
U, S, V = get_usv(samples_norm)

# Visualize the top 36 eigenvectors found
top_eigenvectors = U[:, :36].T
display_data(top_eigenvectors, 6, 6)
```



2.5.2 Dimensionality Reduction

Now that you have computed the principal components for the face dataset, you can use it to perform dimensionality reduction. Reducing the dimensionality with PCA would allow you to use a learning algorithm with a smaller input size (e.g., 100 dimensions) instead of the original 1024 dimensions.

The next step 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 below, an approximate recovery of the data is performed and the projected face images are displayed. Comparing the reconstruction to the originals, you can observe that the general structure and appearance of the face are retained while fine details are lost. This is a remarkable reduction (more than 10×) in the dataset size that can help speed up your learning algorithm significantly. For example, if you were training a logistic regression model to perform face recognition (given a face image, predict the identity of the person), you can use the reduced input of only 100 dimensions instead of the original pixels.

In [ ]:
```
# Project each image down to 100 dimensions
z = project_data(samples_norm, U, 100)  # complete this code

# Attempt to recover the original data
recovered_samples = recover_data(z, U, 100)  #complete this code

# Plot the dimension-reduced data
display_data(recovered_samples)
plt.show()
```