Problem Set 4

See website for due date

What to submit (staple in this order):

1. grade5.txt
2. Write up (put all code, comments, figures and results into a single document),
3. Written question answers

Grading. We will randomly choose a subset of submissions and verify that 1) the code produces the answers and 2) the points computation in "grade5.txt" is correct. There are three written problems, one of them will be chosen at random and will be worth 30 points. Total: 130 points.

Programming Questions

1. 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 ex6.m for this part of the exercise.

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 $\{x^{(1)}, \ldots, x^{(m)}\}$ (where $x^{(i)} \in \Re^n$), 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 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 cˆ(i), the index
    % of the centroid assigned to example i
    idx = findClosestCentroids(X, centroids);

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

The inner-loop of the algorithm repeatedly carries out two steps: (i) Assigning each training example $x^{(i)}$ to its closest centroid, and (ii) 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.

1.1.1 Finding closest centroids

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

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

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 idx(i) in the starter 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,...,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 findClosestCentroids.m, the script ex6.m will run your code and you should see the output [1 3 2] corresponding to the centroid assignments for the first 3 examples. [30 points for getting this answer]

1.1.2 Computing centroid means

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 two examples say $x^{(3)}$ and $x^{(5)}$ are assigned to centroid k = 2, then you should update $\mu_2 = \frac{1}{2}(x^{(3)} + x^{(5)})$.

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 ex6.m will run your code and output the centroids after the first step of K- means. The centroids should be [2.428301 3.157924], [5.813503 2.633656], [7.119387 3.616684]. [30 points for getting this answer]

1.2 K-means on example dataset

After you have completed the two functions (findClosestCentroids and computeCentroids), the next step in ex6.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.

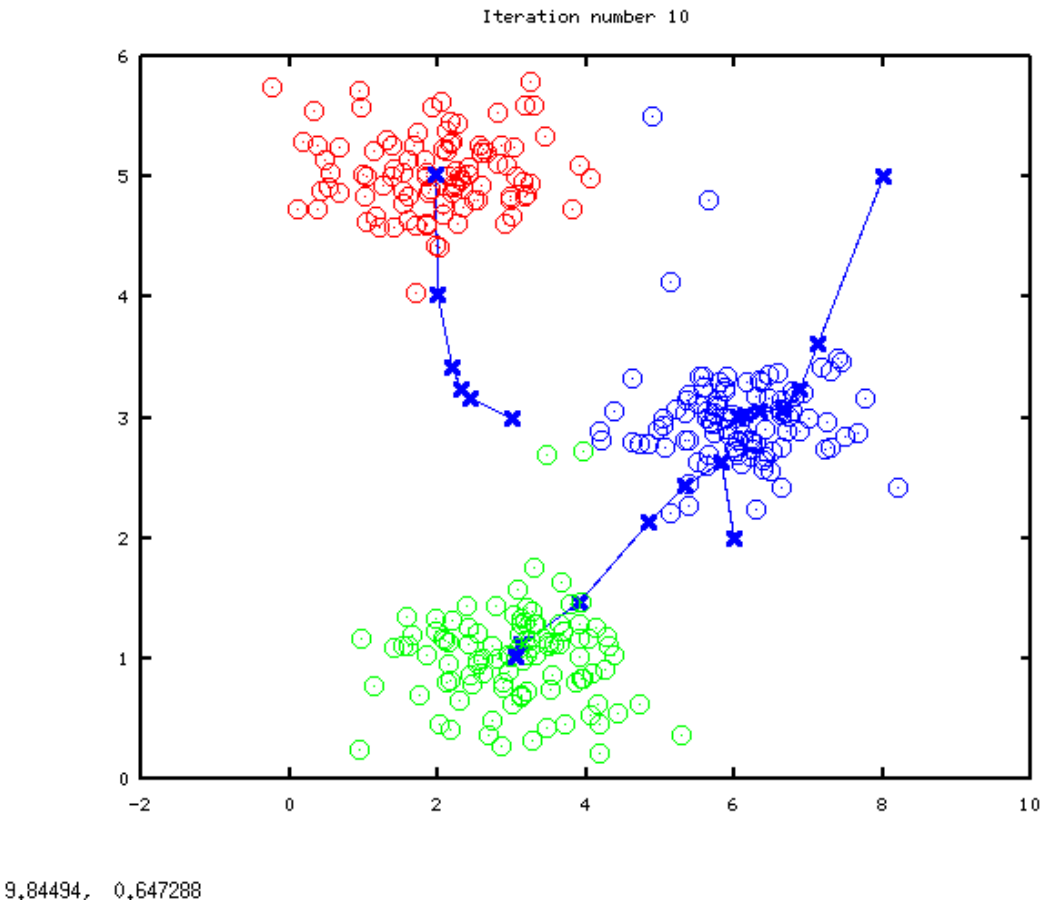


Figure 1: The expected output.

1.3 Random initialization

The initial assignments of centroids for the example dataset in ex6.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 with the following code:

```
% Initialize the centroids to be random examples

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

The code above first randomly permutes the indices of the examples (using randperm).

Then, it selects the first K examples based on the random permutation of the indices. This allows the examples to be selected at random without the risk of selecting the same example twice.

1.4 Image compression with K-means

Figure 2: The original 128x128 image.

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.

1.4.1 K-means on pixels

In Matlab and Octave, images can be read in as follows:

```
% Load 128x128 color image (bird small.png)
A = imread('bird_small.png');

% You will need to have installed the image package to used
% imread. If you do not have the image package installed, you
```

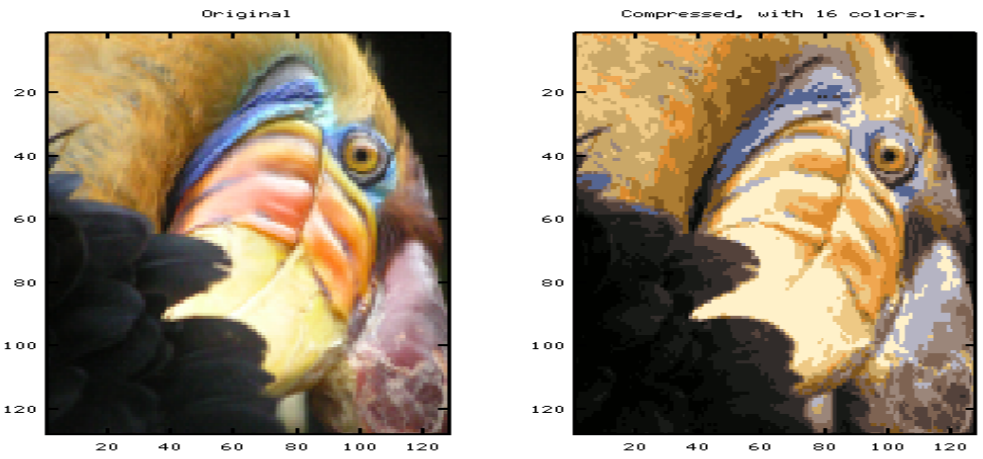
```
% should instead change the following line to
%
% load('bird small.mat'); % Loads the image into the variable A
```

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 the pixel at row 50 and column 33.

The code inside ex6.m first loads the image, and then reshapes it to create an m × 3 matrix of pixel colors (where m = 16384 = 128 × 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 × 128 × 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 × 24 + 128 × 128 × 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 3 shows the reconstruction we obtained. Even though the resulting image retains most of the characteristics of the original, we also see some compression artifacts.



228.117, -10.5316

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

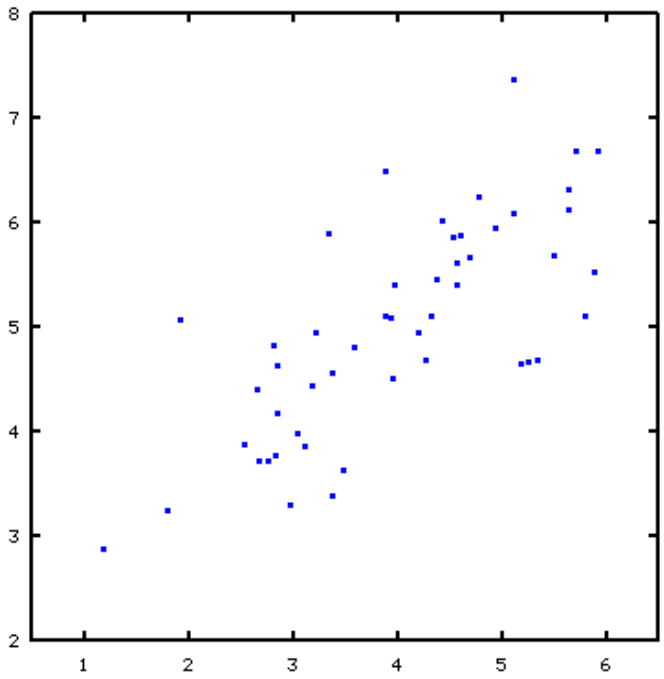
2 Principal Component Analysis

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, ex6_pca.m, will help you step through the first half of the exercise.

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 script ex6_pca.m will plot the training data (Figure 4). 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.



7.28386, 2.37219

Figure 4: Example Dataset 1

2.2 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 Octave's SVD function to compute the eigenvectors U_1, U_2, . . . , 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. In the provided script ex6_pca.m, this normalization has been performed for you using the featureNormalize function.

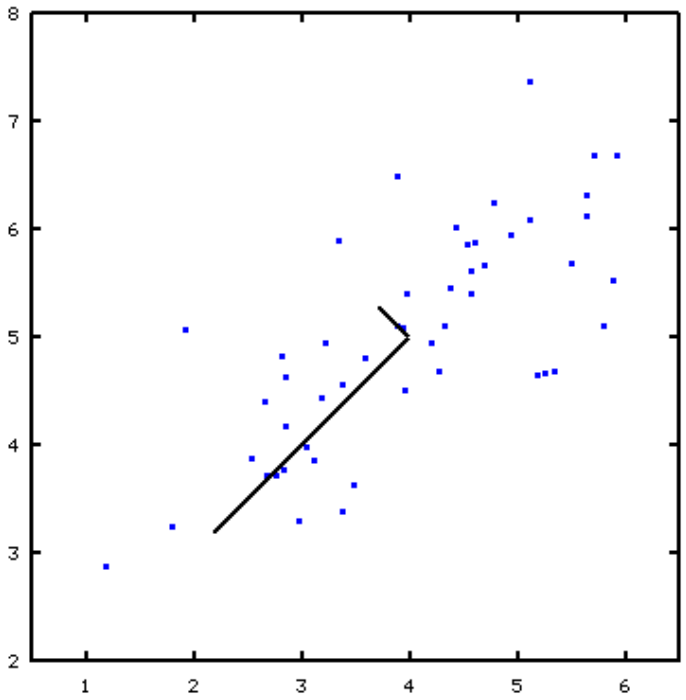
After normalizing the data, you can run PCA to compute the principal components. You task is to complete the code in pca.m to compute the principal components of the dataset. First, you should compute the covariance matrix of the data, which is given by:

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

where X is the data matrix with examples in rows, and m is the number of examples. Note that Σ is a n × n matrix and not the summation operator.

After computing the covariance matrix, you can run SVD on it to compute the principal components. In Octave, 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 ex6_pca.m script will run PCA on the example dataset and plot the corresponding principal components found (Figure 5). 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 Octave may instead output the negative of this, since U_1 and $-U_1$ are equally valid choices for the first principal component.) [20 points for getting this answer]



7.10887, 2.32005

Figure 5: Computed eigenvectors of the dataset

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 less dimensions in the input.

2.3.1 Projecting the data onto the principal components

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 reduce = U(:, 1:K).

Once you have completed the code in projectData.m, ex6_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).[10 points for getting this answer]

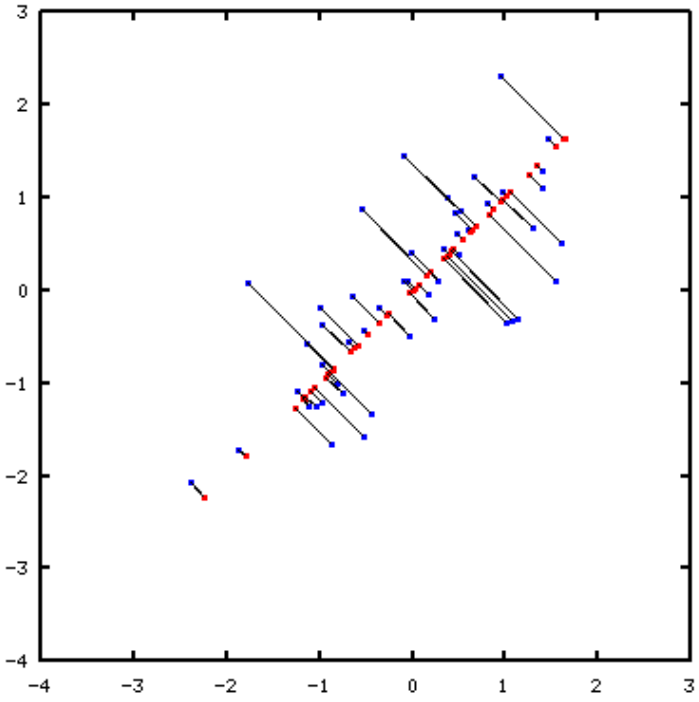
2.3.2 Reconstructing an approximation of the data

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 projectData.m, ex6_pca.m will recover an approximation of the first example and you should see a value of about [-1.047 -1.047]. [10 points for getting this answer]

2.3.3 Visualizing the projections

After completing both projectData and recoverData, ex6_pca.m will now perform both the projection and approximate reconstruction to show how the projection affects the data. In Figure 6, 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.



5.01918, -3.60773

Figure 6: The normalized and projected data after PCA.

2.4 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 ex6faces.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 ex6_pca.m will load and visualize the first 100 of these face images (Figure 7).

```
426.190,  312.507
```

Figure 7: Faces dataset

2.4.1 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. The script ex6_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 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 ex6_pca.m displays the first 36 principal components that describe the largest variations (Figure 8). If you want, you can also change the code to display more principal components to see how they capture more and more details.

```
256.114,  187.904
```

Figure 8: Principal components on the face dataset

2.4.2 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 ex6_pca.m will project the face dataset onto only the first 100 principal components. Concretely, each face image is now described by a vector $z^{(i)} \in \Re^{100}$.

To understand what is lost in the dimension reduction, you can recover the data using only the projected dataset. In ex6_pca.m, an approximate recovery of the data is performed and the original and projected face images are displayed side by side (Figure 9). 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×) 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.

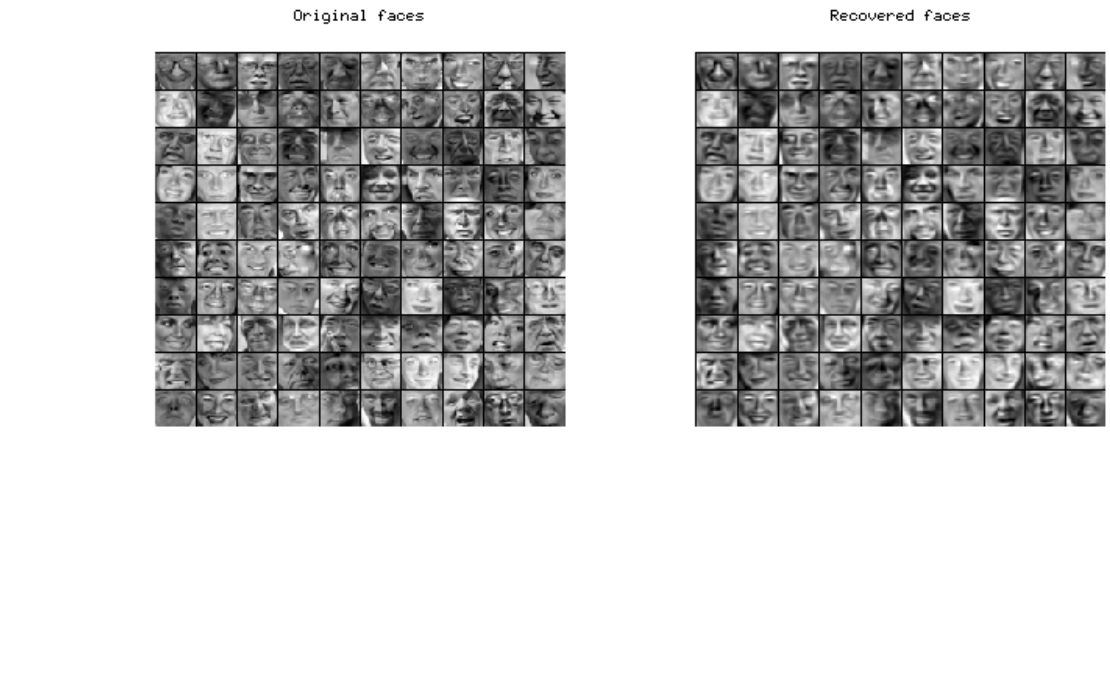


Figure 9: Original images of faces and ones reconstructed from only the top 100 principal components.

Written Questions

Problem 1

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.

9.1 (⋆) **WWW** 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 $\{\boldsymbol{\mu}_k\}$, the K-means algorithm must converge after a finite number of iterations.

Problem 2

Read the beginning of Section 9.2 which describes Gaussian mixture models, and solve Problem 9.3.

9.3 (⋆) **WWW** Consider a Gaussian mixture model in which the marginal distribution $p(\mathbf{z})$ for the latent variable is given by (9.10), and the conditional distribution $p(\mathbf{x}|\mathbf{z})$ for the observed variable is given by (9.11). Show that the marginal distribution $p(\mathbf{x})$, obtained by summing $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ over all possible values of $\mathbf{z}$, is a Gaussian mixture of the form (9.7).

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k). \tag{9.7}$$

$$p(\mathbf{z}) = \prod_{k=1}^{K} \pi_k^{z_k}. \tag{9.10}$$

$$p(\mathbf{x}|\mathbf{z}) = \prod_{k=1}^{K} \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_k}. \tag{9.11}$$

Problem 3

PCA is sometimes used to reduce the dimensionality of the data points before training a classifier, either to reduce computational requirements or to prevent overfitting. However, this can sometimes make the problem more difficult to solve. Give a very simple example of such a case for a 2-dimensional dataset and a binary classification problem, and explain your reasoning.