Machine Learning	**Assignment 6**
comp540 Spring 2017	**due April 14 2017 at 8 pm**

This homework is due April 14 at 8 pm on Canvas. The code base `hw6.zip` *for the assignment is an attachment to Assignment 6 on Canvas. You will add your code at the indicated spots in the files there. Place your answers to Problems 1, and 2 (typeset) in a file called* `writeup.pdf` *and add it to the zip archive. Upload the entire archive back to Canvas before the due date and time.*

1 EM for mixtures of Bernoullis (10 points)

- (5 points) Show that the M step for ML estimation of a mixture of Bernoullis is given by

$$\mu_{kj} = \frac{\sum_{i=1}^{m} r_k^{(i)} x_j^{(i)}}{\sum_{i=1}^{m} r_k^{(i)}}$$

- (5 points) Show that the M step for the MAP estimation of a mixture of Bernoullis with a $Beta(\alpha, \beta)$ prior is given by

$$\mu_{kj} = \frac{(\sum_{i=1}^{m} r_k^{(i)} x_j^{(i)}) + \alpha + 1}{(\sum_{i=1}^{m} r_k^{(i)}) + \alpha + \beta - 2}$$

2 Principal Components Analysis (10 points)

In class, we showed that PCA finds the variance maximizing directions onto which to project the data. In this problem, we find another interpretation of PCA. Suppose we are given a set of points $\{x^{(1)}, \ldots, x^{(m)}\}$. Let us assume that we have preprocessed the data to have zero-mean and unit variance in each dimension. For a given unit-length vector u, let $f_u(x)$ be the projection of point x into the direction given by u. Let $V = \{\alpha u : \alpha \in \Re\}$. Then,

$$f_u(v) = argmin_{v \in V} ||x - v||^2$$

Show that the unit length vector u that minimizes the mean squared error between projected points and original points corresponds to the first principal component for the data, i.e., show that

$$argmin_{u:uu^T=1} \sum_{i=1}^{m} ||x^{(i)} - f_u(x^{(i)})||^2$$

gives the first principal component of the data. Hint: First show that $f_u(x) = u^T x u$.

3 K-means clustering (15 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 `kmeans.ipynb` for this part. The relevant files for this part are in the folder `kmeans`.

Name	Edit?	Read?	Description
kmeans.ipynb	No	Yes	Python notebook that will run your functions for k-means clustering
utils_kmeans.py	Yes	Yes	Functions for implementing k-means
kmeansdata2.mat	No	No	Simple 2D Example dataset for k-means
bird_small.png	No	No	Example image for compression by k-means
hw6.pdf	No	Yes	this document

Implementing k-means

The k-means algorithm is a method to automatically cluster similar examples together. That is, given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$ (where $x^{(i)} \in \Re^d$), 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 clsuter 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 = kmeans_init_centroids(X, K)
for iter in range(iterations):
     # Cluster assignment step: Assign each data point to the
     # closest centroid. idx[i]  is the index
     # of the centroid assigned to example i
     idx = find_closest_centroids(X, centroids)
     # Move centroid step: Compute means based on centroid
     # assignments
     centroids = compute_centroids(X, idx, K);
```

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.

Problem 3.1: Finding closest centroids (5 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 `idx[i]` in our code.

Your task is to complete the function `find_closest_centroids` in `utils_kmeans.py`. This function takes the data matrix `X` and the locations of all centroids inside `centroids` and outputs a one-dimensional array `idx` that holds the index (a value in {0, ...,K-1}, 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 appropriate cell of the notebook `kmeans.ipynb` you should see the output [0 2 1] corresponding to the centroid assignments for the first 3 examples in our data set.

Problem 3.2: Computing centroid means (5 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 function `compute_centroids` in `utils_kmeans.py`. 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 should run faster. Once you have completed the function, the appropriate cell in `means.ipynb` will run your function and output the centroids after the first step of k-means.

k-means on example dataset

After you have completed the two functions (`find_closest_centroids` and `compute_centroids`), the next step in `means.ipynb` 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 `run_kmeans` function in `utils_kmeans.py`. We encourage you to take a look at the function to understand how it works. Notice that the function 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. At the end, your figure should look as the one displayed in Figure 1.

Problem 3.3: Random initialization (5 points)

The initial assignments of centroids for the example dataset 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 `kmeans_init_centroids` in `utils_kmeans.py`. First, randomly permute the indices of the examples. 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.

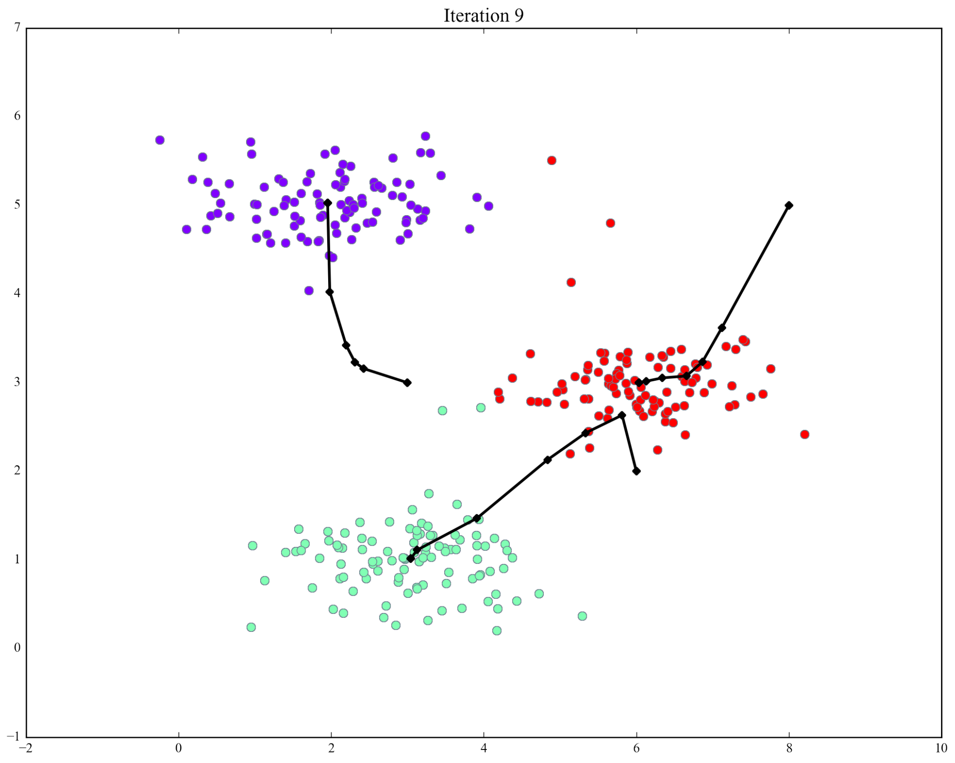

Figure 1: Expected output of k-means

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 refered 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. 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.

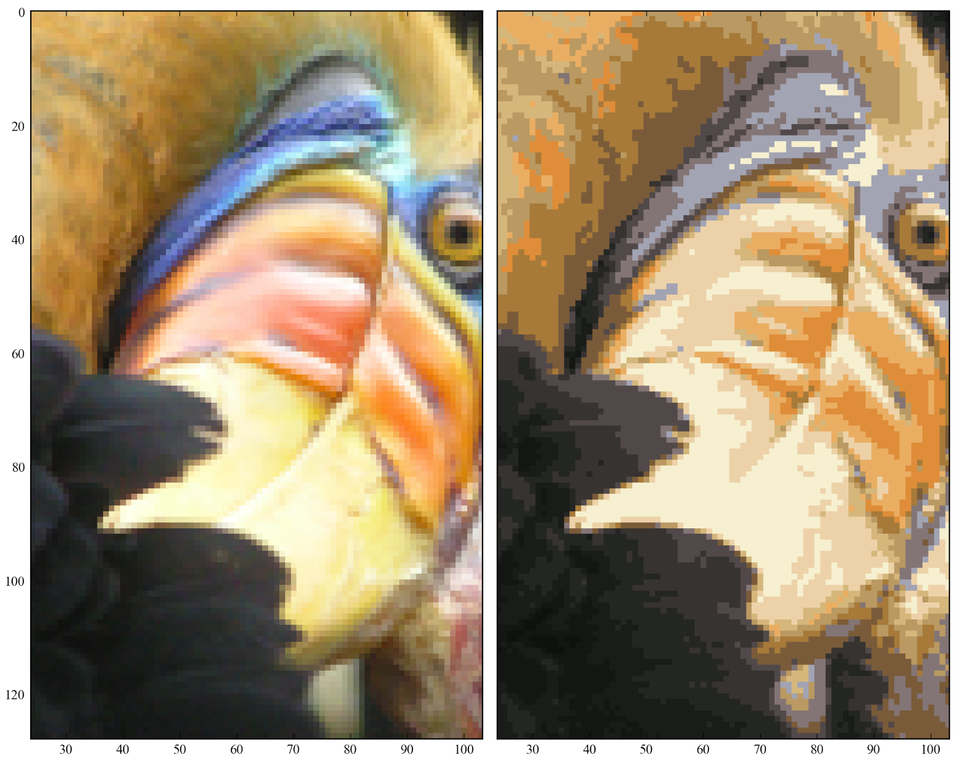

Figure 2: Original and reconstructed image (when using k-means to compress the image).

k-means on pixels

A cell in `kmeans.ipynb` 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 `find_closest_centroids` 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.

4 Principal Components Analysis (15 points)

In this exercise, you will use principal component analysis (PCA) to perform dimensionality reduc-

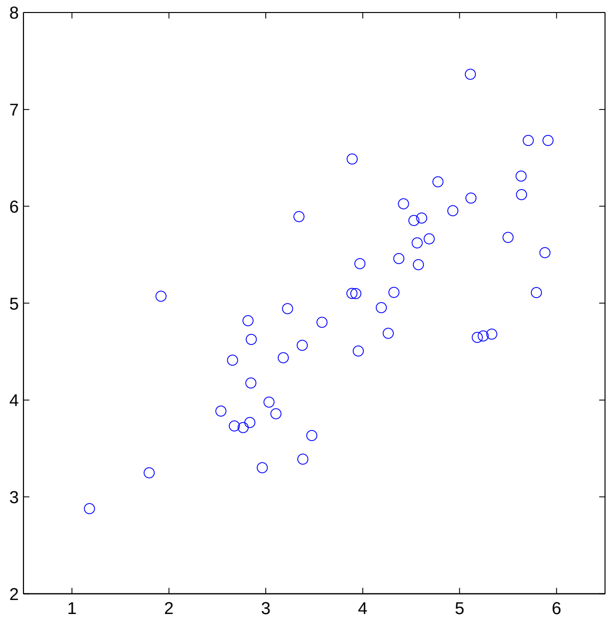

Figure 3: Example Dataset 1

tion. 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 faces. The notebook `pca.ipynb`, will help you step through the exercise.

Name	**Edit?**	**Read?**	**Description**
pca.ipynb	Yes	Yes	Python notebook for running your functions for principal component analysis
utils_pca.py	Yes	Yes	functions for implementing PCA
pcadata1.mat	No	No	Simple 2D Example dataset for PCA
pcafaces.mat	No	No	faces dataset for PCA
hw6.pdf	No	Yes	this document

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 notebook `pca.ipynb` 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.

Problem 4.1: 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 `numpy`'s SVD function to compute the eigenvectors $U_1, U_2, \ldots, U_n$. 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. The notebook `pca.ipynb`, does this normalization for you using the `feature_normalize` function. After normalizing the data, you can run PCA to compute the principal components. You task is to complete the function `pca` in `utils_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 $d \times d$ matrix and not the summation operator.

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

```
U, S, V = np.linalg.svd(Sigma,full_matrices = False)
```

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

Once you have completed the function `pcs`, the `pca.ipynb` notebook 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 `numpy` 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.

Problem 4.2: Projecting the data onto the principal components (5 points)

You should now complete the function `project_data` in `utils_pca.py`. Specifically, you are given a dataset `X`, the principal components `U`, and the desired number of dimensions to reduce to `K`. You

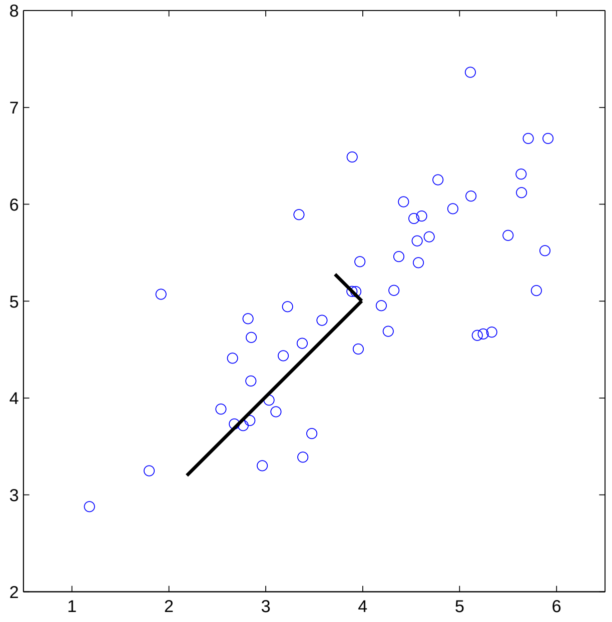

Figure 4: Computed eigenvectors of the dataset

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`. Once you have completed `project_data`, the `pica.ipynb` notebook 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).

Problem 4.3: 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 the function `recover_data` in `utils_pca.py` to project each example in `Z` back onto the original space and return the recovered approximation in `X_rec`. Once you have completed the function `recover_data`, `pca.ipynb` 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 `project_data` and `recover_data`, `pca.ipynb` 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

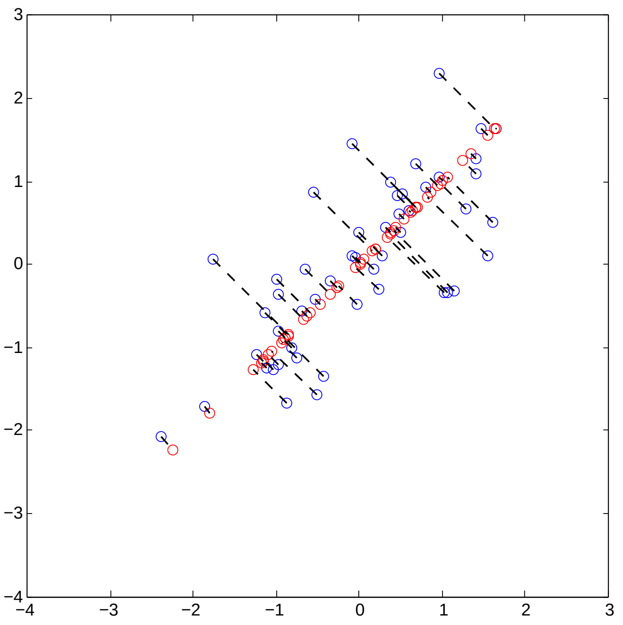

Figure 5: The normalized and projected data after PCA

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 `pcafaces.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 cell in `pcaipynb` 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 `X`. The notebook `pca.ipynb` will do this for you and then run your PCA function. 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 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 notebook `pca.ipynb` displays the first 25 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.

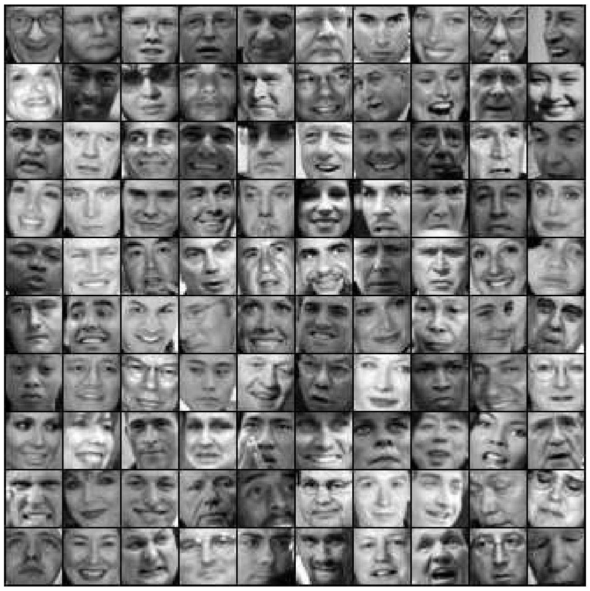

Figure 6: Faces dataset

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 `pca.ipynb` 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 `pica.ipynb`, 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 identitfy of the person), you can use the dimension reduced input of only a 100 dimensions instead of the original pixels.

5 Anomaly detection (10 points)

In this problem, you will implement an anomaly detection algorithm to detect anomalous behavior in server computers. The features measure the throughput (mb/s) and latency (ms) of response of

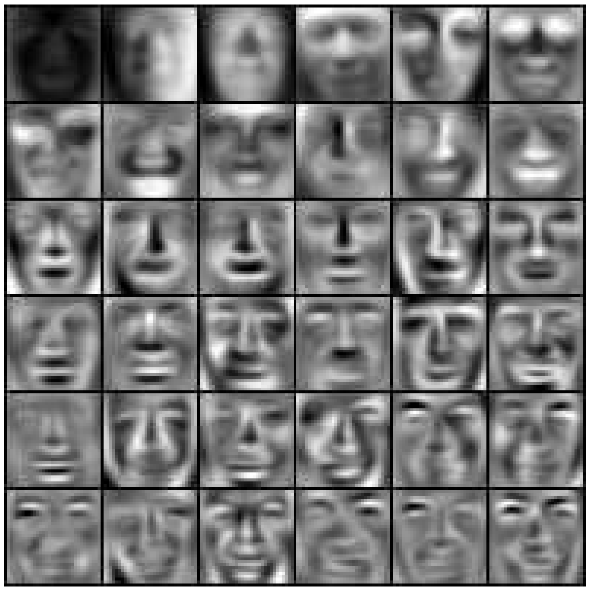

Figure 7: Principal components on the face dataset

each server. We have a dataset $\mathcal{D} = \{x^{(1)}, \ldots, x^{(m)}\}$ of unlabeled examples where $m = 307$ and each example $x^{(i)} \in \Re^2$. You suspect that the vast majority of these examples are normal (non-anomalous) examples of the servers operating normally, but there might also be some examples of servers acting anomalously within this dataset. The files for this exercise are:

Name	**Edit?**	**Read?**	**Description**
anomaly_detection.ipynb	No	Yes	Python notebookt that will run anomaly detection functions
utils_anomaly.py	Yes	Yes	functions to implement anomaly detection
anomalydata1.mat	No	No	First example dataset for anomaly detection
anomalydata2.mat	No	No	Second example dataset for anomaly detection
hw6.pdf	No	Yes	this document

You will use a Gaussian model to detect anomalous examples in your dataset. You will first start on a 2D dataset that will allow you to visualize what the algorithm is doing. On that dataset you will fit a Gaussian distribution and then find values that have very low probability and hence can be considered anomalies. After that, you will apply the anomaly detection algorithm to a larger dataset with many dimensions. You will be using `anomaly_detection.ipynb` for this problem.

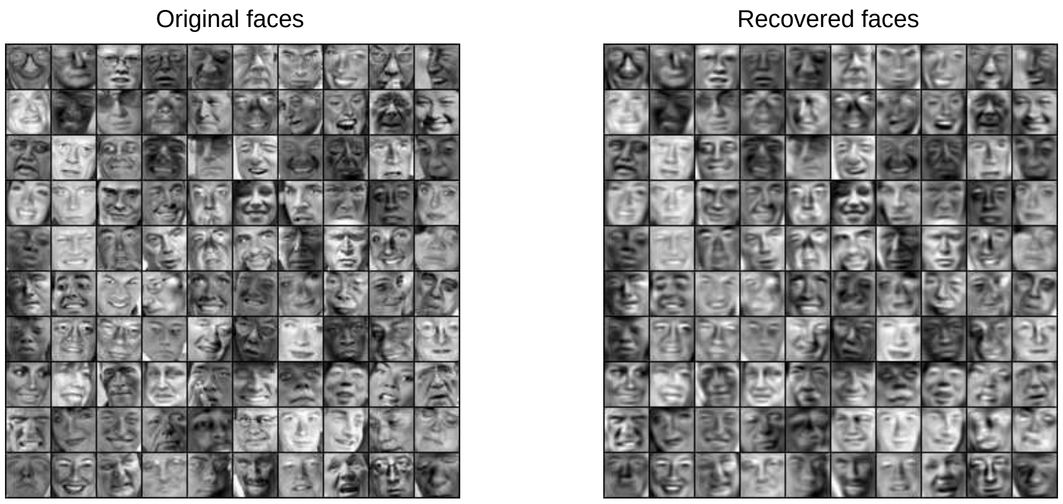

Figure 8: Original and reconstructed face dataset reconstructed from only the top 100 principal components

The first part of `anomaly_detection.ipynb` will visualize the dataset as shown in Figure 9.

Gaussian distribution

To perform anomaly detection, you will first need to fit a model to the datas distribution. Given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$ (where $x^{(i)} \in \Re^2$), you want to estimate the Gaussian distribution for each of the features x_j. For each feature $j = 1 \ldots d$, you need to find parameters μ_j and σ_j^2, that fit the data in the j^{th} dimension in each example. Recall that a univariate Gaussian distribution is given by

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where μ is the mean and σ^2 is the variance.

Problem 5.1: Estimating parameters of a Gaussian distribution (5 points)

You can estimate the parameters, μ_j and σ_j^2 of the j^{th} feature by using the following equations. To estimate the mean, you will use:

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

To estimate the variance, you will use:

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m} (x_j^{(i)} - \mu_j)^2$$

Your task is to complete the function `estimate_gaussian` in `utils_anomaly.py`. This function takes as input the data matrix `X` and should output an d-dimensional vector `mu` that holds the mean of all the d features and another d-dimensional vector `var` that holds the variances of all

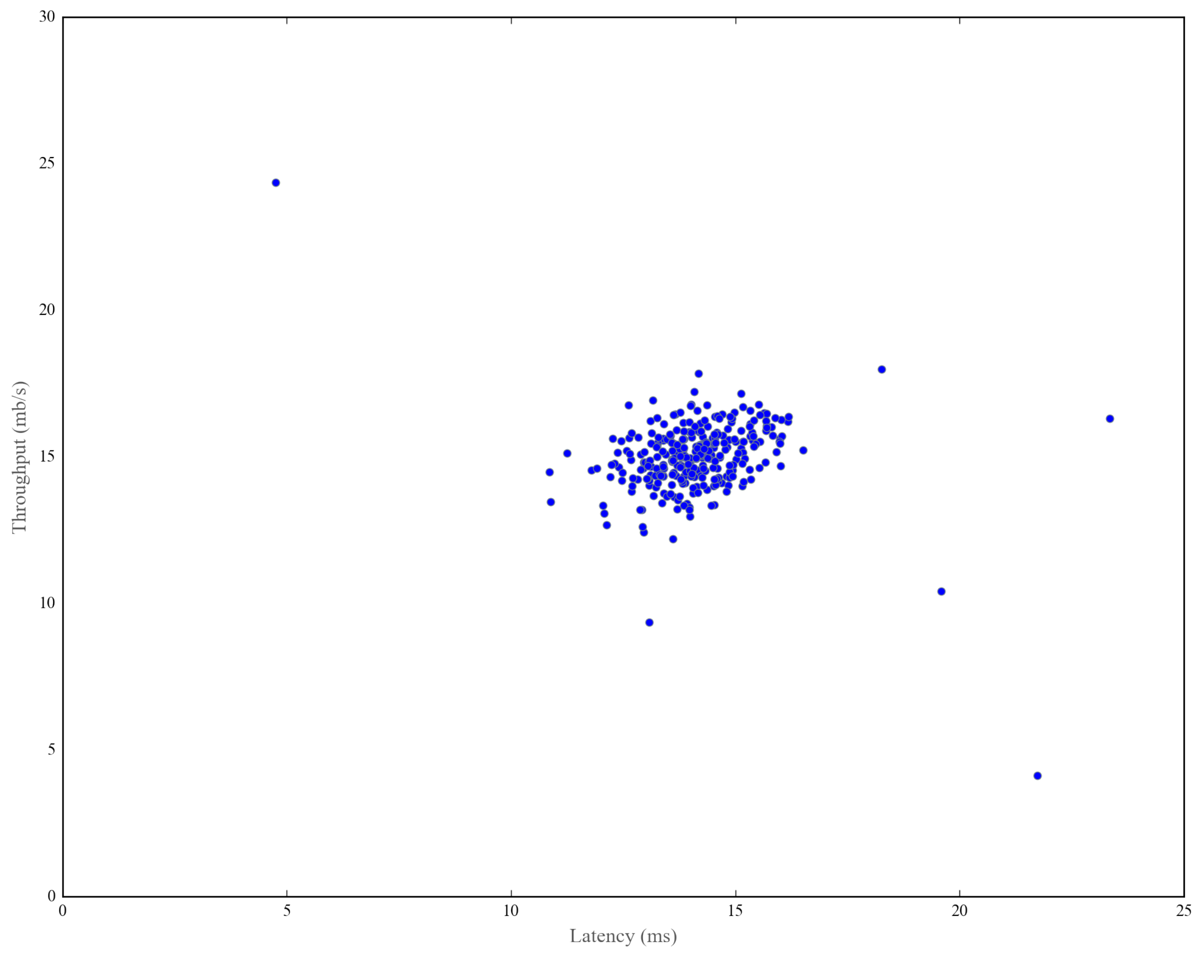

Figure 9: The first dataset

the features. You should implement this in a vectorized way to be more efficient. Note that in `numpy`, the `var` function will (by default) use $\frac{1}{m-1}$ instead of $\frac{1}{m}$ when computing σ_j^2. Use the `ddof` parameter of `numpy.var` and set it to zero for calculating variance.

Once you have completed the function `estimate_gaussian`, the next cell of `anomaly_detection.ipynb` will visualize the contours of the fitted Gaussian distribution. You should get a plot similar to Figure 10. From your plot, you can see that most of the examples are in the region with the highest probability, while the anomalous examples are in the regions with lower probabilities.

Problem 5.2: Selecting the threshold ϵ (5 points)

Now that you have estimated the Gaussian parameters, you can investigate which examples have a very high probability given this distribution and which examples have a very low probability. The low probability examples are more likely to be the anomalies in our dataset. One way to determine which examples are anomalies is to select a threshold based on a validation set. In this part of the assignment, you will implement an algorithm to select the threshold ϵ using the F1 score on a

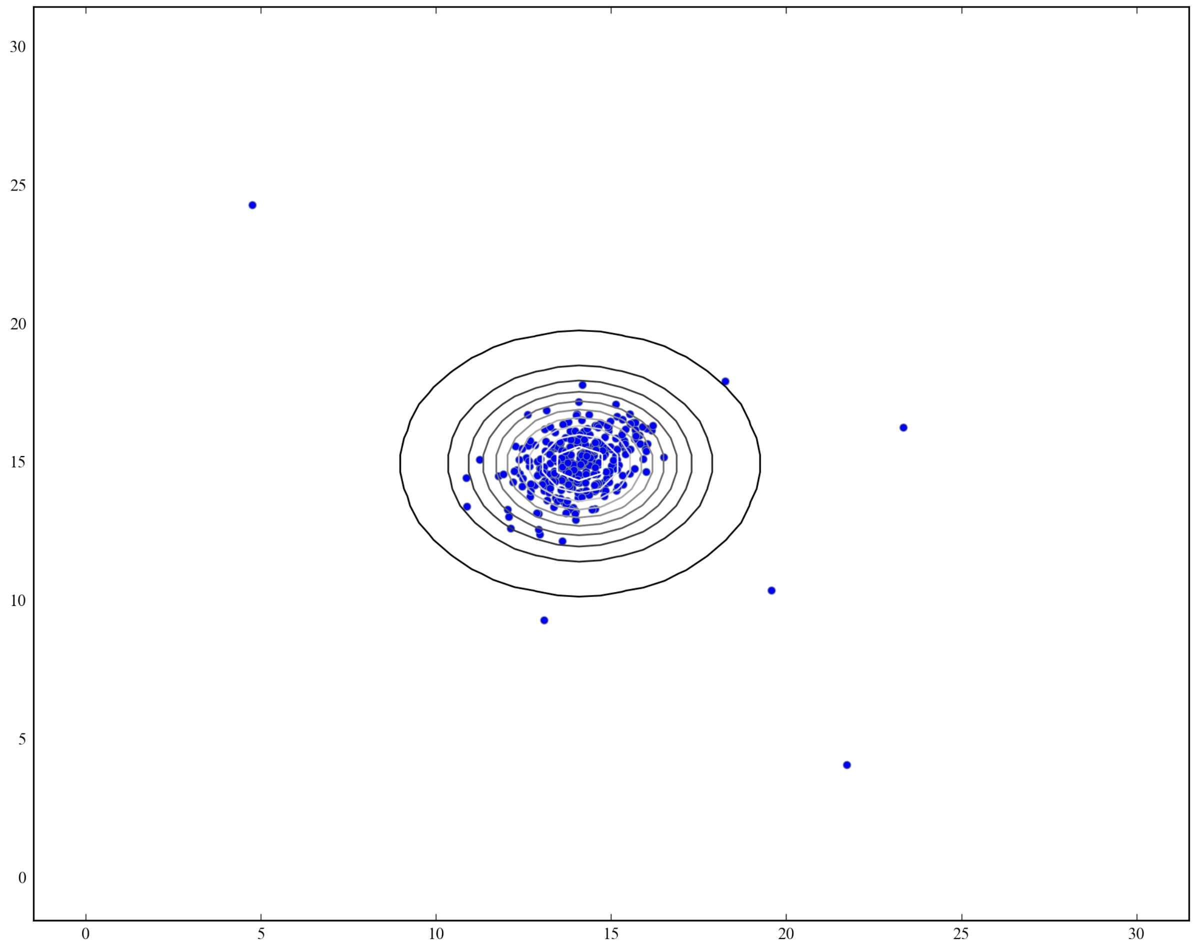

Figure 10: The Gaussian distribution contours of the distribution fit to the dataset

validation set.

You should now complete the function `select_threshold` in `utils_anomaly.py`. For this, we will use a validation set $\{(x_v^{(1)}, y_v^{(1)}), \ldots, (x_v^{(m_v)}, y_v^{(m_v)})\}$, where the label $y_v = 1$ corresponds to an anomalous example, and $y_v = 0$ corresponds to a normal example. For each example in the validation set, we will compute $p(x_v^{(i)})$. The vector of these probabilities $p(x_v^{(1)}), \ldots, p(x_v^{(m_v)})$ is passed to `select_threshold` function in the vector `pval`. The corresponding set of labels $y_v^{(1)}, \ldots, y_v^{(m_v)}$ is passed to the same function in the vector `yval`.

The function `select_threshold.` should return two values; the first is the selected threshold ϵ. If an example x has a low probability, i.e., $p(x) < \epsilon$, then it is considered to be an anomaly. The function should also return the F1 score, which tells you how well you are doing on finding the ground truth anomalies given a certain threshold. For many different values of ϵ, you will compute the resulting F1 score by computing how many examples the current threshold classifies correctly and incorrectly.

The F1 score is computed using precision (*prec*) and recall (*rec*):

$$F_1 = \frac{2 \times prec \times rec}{prec + rec}$$

You compute precision and recall by:

$$\begin{aligned} prec &= \frac{tp}{tp + fp} \\ rec &= \frac{tp}{tp + fn} \end{aligned}$$

where

- *tp* is the number of true positives: the ground truth label says it is an anomaly and our algorithm correctly classified it as an anomaly.
- *fp* is the number of false positives: the ground truth label says it is not an anomaly, but our algorithm incorrectly classified it as an anomaly.
- *fn* is the number of false negatives: the ground truth label says it is an anomaly, but our algorithm incorrectly classified it as not being anomalous.

In the provided function `select_threshold`, there is already a loop that will try many different values of ϵ and select the best ϵ based on the F1 score. You need to implement the computation of the F1 score over all the validation examples (to compute the values *tp*, *fp*, *fn*). You should see a value for epsilon of about $8.99 * 10^{-5}$.

Implementation hints: In order to compute *tp*, *fp* and *fn*, you should use a vectorized implementation rather than loop over all the examples.

Once you have completed the function `select_threshold`, the next cell in `anomaly_detection.ipynb` will run your function and color the anomalies red in the plot as in Figure 11.

High dimensional dataset

The last part of the notebook `anomaly_detection.ipynb` will run the anomaly detection algorithm you implemented on a more realistic and much harder dataset. In this dataset, each example is described by 11 features, capturing many more properties of the compute servers. The script will use your functions to estimate the Gaussian parameters, evaluate the probabilities for both the training data `X` from which you estimated the Gaussian parameters, and do so for the the validation set `Xval`. Finally, it will use `select_threshold` to find the best threshold ϵ. You should see a value of about $1.38 * 10^{-18}$ for ϵ, and 117 anomalies should be found.

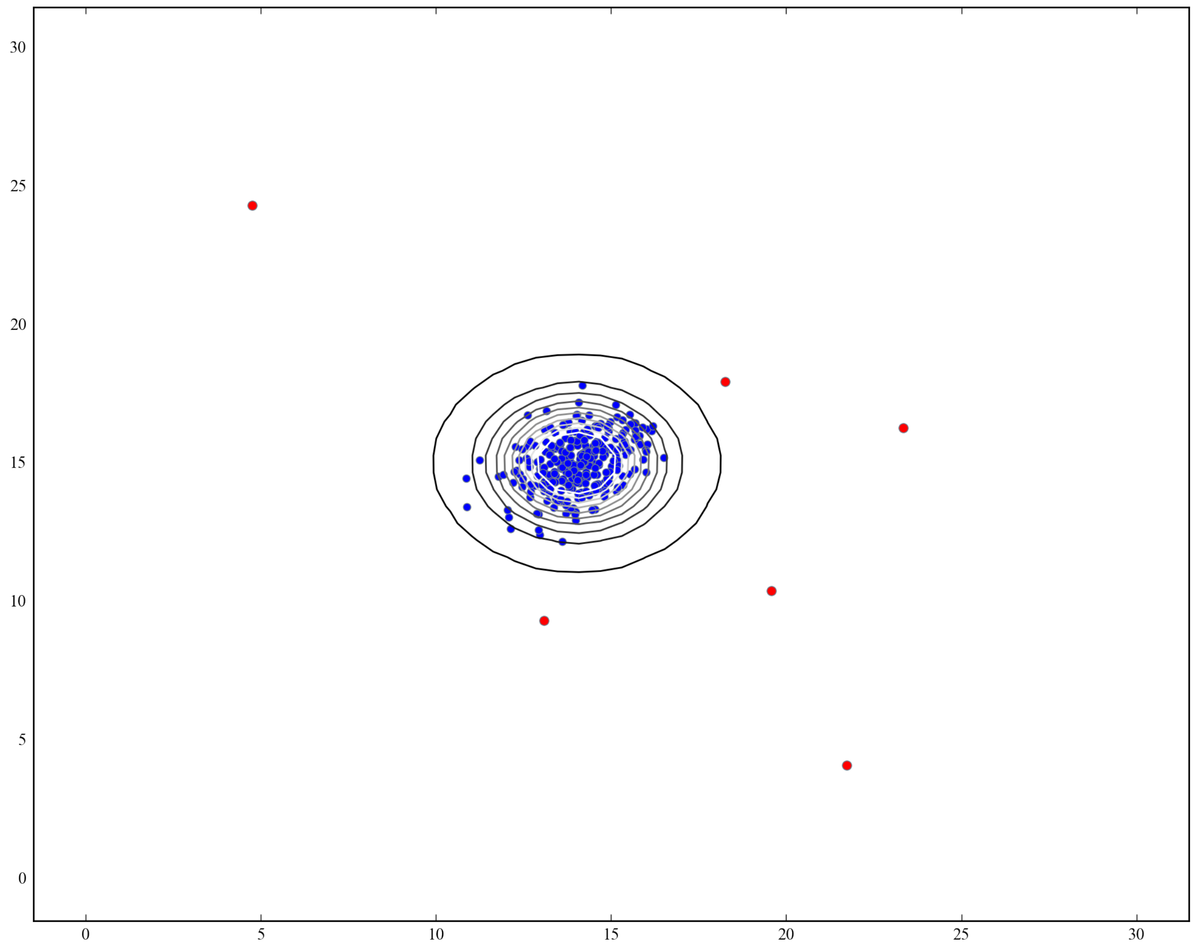

Figure 11: The classified anomalies