COMP 4999/COMP 5015 Homework 3

Instructions

Upload your homework to Moodle. There are two assignments for this homework. A complete submission should include:

1. A PDF report which includes your answers to both the writing questions and the coding questions. ⟶ Submit to "A3-report.pdf". We are going to **grade the assignment mainly based on this report**. Please make sure this report contains all the answers to the questions as the graders only refer to your source codes sparsely when grading the assignment. If your answer refers to part of the source codes, it might be a good idea to include snippets of themin the report.

2. Source code for all of your experiments (AND figures) zipped into a single .zip file. ⟶ Also, submit to "A3-code.zip" with your code. If you use Python, please include all `.py` files. If you use IPythonNotebook, please include all `.ipynb` files. If you use some other language, include all build scripts necessary to build and run your project along with instructions on how to compile and run your code.

The write-up should summarize your work, solution performance, and insights. Include a cover page with the class name, homework number, and team member names. Answers should be clear and organized. Templates from Overleaf (*Homework Assignment* or *Project/Lab Report*) are recommended.

Please include all relevant information for a question, including text response, equations, figures, graphs, output, etc. If you include graphs, be sure to include the source code that generated them. You are encouraged to work in groups of 2.

You are allowed to use well known libraries such as `scikit-learn, scikit-image, numpy, scipy, etc.` in this assignment. Any reference or copy of public code repositories should be properly cited in your submission (examples include Github,Wikipedia, Blogs).

1 Programming Exercises

1.1 Eigenface for face recognition. (40 pts)

In this assignment, you will implement the Eigenface method for recognizing human faces. You will use face images from *The Yale Face Database B*, where there are 64 images under different lighting conditions per each of 10 distinct subjects, 640 face images in total.

Read more (optional):

- Eigenface on Wikipedia: `https://en.wikipedia.org/wiki/Eigenface`
- Eigenface on Scholarpedia: `http://www.scholarpedia.org/article/Eigenfaces`

(a) (**2 pts**) Download *The Face Dataset* and unzip faces.zip. You will find a folder called `images` which contains all the training and test images; `train.txt` and `test.txt` specify the training set and test (validation) set split respectively, each line gives an image path and the corresponding label.

(b) (**2 pts**) Load the training set into a matrix $\mathbf{X}$: there are 540 training images in total, each has 50×50 pixels that need to be concatenated into a 2500-dimensional vector. So the size of X should be 540×2500, where each row is a flattened face image. Pick a face image from X and display that image in grayscale. Do the same thing for the test set. The size of matrix X_{test} for the test set should be 100×2500.

Below is the sample code for loading data from the training set. You can directly run it in Jupyter Notebook:

```
import numpy as np
from scipy import misc
from matplotlib import pylab as plt
import matplotlib.cm as cm
%matplotlib inline

train_labels, train_data = [], []
for line in open('./faces/train.txt'):
    im = misc.imread(line.strip().split()[0])
    train_data.append(im.reshape(2500,))
    train_labels.append(line.strip().split()[1])

    train_data, train_labels = np.array(train_data, dtype=float), np.array(
        train_labels, dtype=int)

    print(train_data.shape, train_labels.shape)
    plt.imshow(train_data[10, :].reshape(50,50), cmap = cm.Greys_r)
    plt.show()
```

(c) (**3 pts**) Compute the **average face** μ from the whole training set by summing up every row in X then dividing by the number of faces. Display the **average face** as a grayscale image.

(d) (**3 pts**) Subtract average face μ from every row in X. That is, $x_i = x_i - \mu$, where x_i is the i-th row of X. Pick a face image after mean subtraction from the new X and display that image in grayscale. Do the same thing for the set X_{test} using the pre-computed average face μ in (c).

(e) (**10 pts**) Perform eigendecomposition on $X^T X = V \Lambda V^T$ to get eigenvectors V^T, where each row of V^T has the same dimension as the face image. We refer to v_i, the i-th row of V^T, as the i-th *eigenface*. Display the first 10 eigenfaces as images in grayscale.

(f) (**10 pts**) The top r eigenfaces $V^T[:, r:] = \{v_1, v_2, ..., v_r\}$ span an r-dimensional linear subspace of the original image space called *face space*, whose origin is the average face μ, and whose axes are the eigenfaces $\{v_1, v_2, ..., v_r\}$. Therefore, using the top r eigenfaces $\{v_1, v_2, ..., v_r\}$, we can represent a 2500-dimensional face image z as an r-dimensional feature vector f. Define $f = V^T[:, r :]z = [v_1, v_2, ..., v_r]^T z$. Write a function to generate r-dimensional feature matrix F and F_{test} for training images X and test images X_{test}, respectively (to get F, multiply X to the transpose of first r rows of V^T, F should have the same number of rows as X and r columns; similarly for X_{test}).

(g) (**10 pts**) For this problem, you are welcome to use libraries such as `scikit-learn` to perform logistic regression. Extract training and test features for $r = 10$. Train a Logistic Regression model using F and test on F_{test}. Report the classification accuracy on the test set. Plot the classification accuracy on the test set as a function of r when $r = 1, 2, ..., 200$. Use *one-vs-rest* logistic regression, where a classifier is trained for each possible output label. Each classifier is trained on faces with that label as positive data and all faces with other labels as negative data. `sklearn` calls this the *ovr* mode.

1.2 Implement EM algorithm. (40 pts)

In this problem, you will implement a bimodal GMM model fit using the EM algorithm. Bimodal means that the distribution has two peaks, or that the data is a mixture of two groups. If you want, you can assume the covariance matrix is diagonal (i.e., it has the form $\text{diag}(\sigma_1^2, \sigma_2^2, ..., \sigma_d^2)$ for scalars σ_j) and you can randomly initialize the parameters of the model.

You will need to use the Old Faithful Geyser Dataset. The data file contains 272 observations of the waiting time between eruptions and the duration of each eruption for the Old Faithful geyser in Yellowstone National Park.

You should do this without calling the Gaussian Mixture library in `scikit-learn`. You can use `numpy` or `scipy` for matrix calculations or generating Gaussian distributions.

(a) (**2 pts**) Treat each data entry as a 2-dimensional feature vector. Parse and plot all data points on the 2-D plane.

(b) (**3 pts**) Recall that EM learns the parameter θ of a Gaussian mixture model $P_\theta(x, z)$ over a dataset $D = \{x^{(i)} | i = 1, 2, ..., n\}$ by performing the E-step and the M-step for $t = 0, 1, 2, ...$ We repeat the E-step and M-step until convergence.

In the E-step, for each $x^{(i)} \in D$, we compute a vector of probabilities $p_{\theta_t}(z = k|x)$ for the event that each $x^{(i)}$ originates from a cluster k given the current set of parameters θ_t.

Write the expression for $P_{\theta_t}(z = k|x)$, which is the posterior of each data point $x^{(i)}$. Recall that by Bayes' rule,

$$P_{\theta_t}(z = k|x) = \frac{P_{\theta_t}(z = k, x)}{P_{\theta_t}(x)} = \frac{P_{\theta_t}(z = k, x)}{\sum_{l=1}^{K} P_{\theta_t}(z = l) P_{\theta_t}(x|z = l)}$$

Note that we have seen this formula in class. We are asking you to write it down and try to understand it before implementing it in part (e).

(c) (**5 pts**) In the M-step, we compute new parameters θ_{t+1}. Our goal is to find μ_k, Σ_k and ϕ_k that optimize

$$\max_{\theta} \left(\sum_{k=1}^{K} \sum_{x \in D} P_\theta(z_k|x) \log P_\theta(x|z_k) + \sum_{k=1}^{K} \sum_{x \in D} P_{\theta_t}(z_k|x) \log P_\theta(z_k) \right)$$

Write down the formula for μ_k, Σ_k, and for the parameters ϕ at the M-step (we have also seen these formulas in class).

(d) (**25 pts**) Implement and run the EM algorithm. Specifically:

1. (10 pts) Implement the EM algorithm from scratch (e.g., in Python and `numpy`).
2. (5 pts) Choose a termination criterion for when the algorithm stops repeating the E-step and the M-step. State your termination criterion and explain the reasoning behind it.
3. (10 pts) Plot the trajectories of the two mean vectors (μ_1 and μ_2) in two dimensions as they change over the course of running EM. You might want to use a scatter plot for this.

(e) (**5 pts**) If you were to run K-means clustering instead of the EM algorithm you just implemented, do you think you will get different clusters? You are welcome to experiment with K-means clustering on the same dataset with $K = 2$. (The KNN library from `scikit-learn` is a good way to try). Comment on why you think the results will or will not change.

Written Exercises

1. K-means Clustering (10 pts)

For a dataset consisting of n points, $x^{(i)} \in \mathbb{R}^d$ for $i = 1, 2, ..., n$, the goal of K-means clustering is to find a function $f : \mathbb{R}^d \to \{1, 2, ..., K\}$ that assigns each point to one of K clusters. Each cluster is represented by a *centroid*, $c^{(k)} \in \mathbb{R}^d$ for $k = 1, 2, ..., K$.

Recall from the lecture on unsupervised learning that the K-means objective is optimized by an iterative process. For each iteration t, let f_t denote the cluster assignment function and $c_t^{(k)}$ denote the kth centroid.

Then, at each iteration t, we perform two steps:

1. Update cluster assignments $f_t(x^{(i)})$ for each point $x^{(i)}$.

2. Update centroids $c_t^{(k)}$ for each cluster $k = 1, 2, \ldots, K$.

Initialization: Set centroids $c_0^{(k)}$ randomly or using an initialization heuristic.
For $t = 1, 2, \ldots,$ **until** K-means converges, **do**:

Step 1. Update cluster assignments such that:

$$f_t(x^{(i)}) = \arg\min_k \|x^{(i)} - c_{t-1}^{(k)}\|_2$$

is the cluster of the closest centroid to $x^{(i)}$, where $\|\cdot\|_2$ denotes the Euclidean norm.

Step 2. Set each centroid c_t^k to be the average of its cluster. Letting:

$$S^{(k)} = \{x^{(i)} \mid f_t(x^{(i)}) = k\}$$

be the number of points assigned to cluster k, we refit centroids as follows:

$$c_t^k = \frac{1}{S^{(k)}} \sum_{i:f_t(x^{(i)})=k} x^{(i)}$$

Letting c_t (i.e., without any superscript) be a shorthand representation for all K centroids $c_t^{(1)}, c_t^{(2)}, \ldots, c_t^{(K)}$, we can express the K-means optimization objective at each time t as follows:

$$J(c_t, f_t) = \sum_{i=1}^{n} \|x^{(i)} - c_t^{f_t(x^{(i)})}\|_2$$

where $c_t^{f_t(x^{(i)})}$ denotes "the centroid for the cluster assignment of $x^{(i)}$ at time t."

In this question, we want to prove that K-means optimization is guaranteed to converge. In other words, we want to prove that the K-means objective $J(c_t, f_t)$ is monotonically decreasing after each step of the K-means optimization procedure:

$$J(c_t, f_t) \leq J(c_{t-1}, f_{t-1}).$$

If we prove this, then convergence follows from the monotone convergence theorem, which states that a monotonically decreasing sequence with a lower bound (the bound is $J(c_t, f_t) \geq 0$, $\forall t$, in our case) is guaranteed to converge.

(a) (**5 pts**) Show that $J(c_{t-1}, f_t) \leq J(c_{t-1}, f_{t-1})$. This is equivalent to proving that the K-means objective is decreasing after updating cluster assignments (but before updating the centroids).

(b) (**5 pts**) Show that $J(c_t, f_t) \leq J(c_{t-1}, f_t)$. This is equivalent to proving that the K-means objective is decreasing after refitting centroids.

2. SVD and Eigendecomposition (10 pts)

Recall that the SVD of an $m \times n$ matrix X is the factorization of X into three matrices:

$$X = UDV^T,$$

where U is a $m \times m$ orthonormal matrix, D is a $m \times n$ diagonal matrix with non-negative real numbers on the diagonal, and V is a $n \times n$ orthonormal matrix. An orthonormal matrix just means that:

$$U^T U = I \quad \text{and} \quad V^T V = I.$$

Show that we can obtain the eigendecomposition of $X^T X$ from the SVD of a matrix X.

(This tells us that we can do an SVD of X and get the same result as the eigendecomposition of $\mathbf{X}^T\mathbf{X}$, but the SVD is faster and easier.)