Machine Learning HW7

Kernel Eigenfaces and t-SNE

GMBA 310712009 楊家碩

a. code with detailed explanations

I. LDA & PCA

a. code with detailed explanations

The main procedure of this program is following:

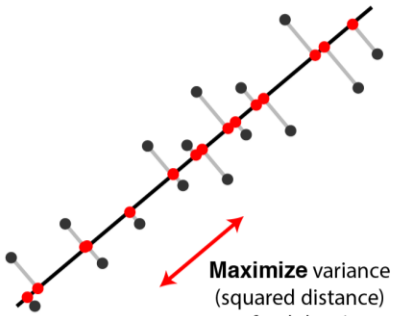
- **read_data.**

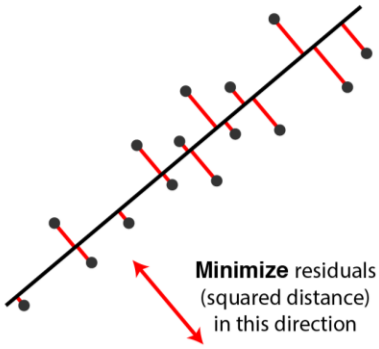
```
def read_data(path, num_subjects):
    filenames = os.listdir(path)
    p = [] #pixel
    label = [[i]*num_subjects for i in range(15)]
    for filename in filenames:
        image = cv2.imread(path+filename, -1)
        p.append(list(image.reshape(-1)))
    images = np.array(p)
    train_labels = np.array(label).reshape(-1)

    return images,train_labels
```

We build a function to read all image data and transform it into a list of multi-dimensional data that is then transformed into a data matrix.

We use read_data function to obtain the picture training data and its labels, and return as images and train_labels.

- **PRINCIPAL COMPONENT ANALYSIS (PCA)**
  - The purpose of PCA is to find a set of directions (technically, a linear subspace) that maximizes the variance of the data once it is projected into that space.
  - Then, to an **orthogonal projection W (Black line)** in which **data X (Black dots)** that maximizes the variance of the data once it is projected into that space. **y=WX (Red dots)**
  - We would like to maximum variance and minimum mean square error (MSE).






```python
def PCA(train_data, train_label, test_data, test_label, k, mode, pathsave):
    mean = np.mean(train_data, axis=0)
    center = train_data - mean

    K_S = kernel(mode, train_data)

    eigenValue, eigenVector = np.linalg.eig(K_S)
    index = np.argsort(-eigenValue) # inverse -> max sort
    eigenValue = eigenValue[index]
    eigenVector = eigenVector[:,index]

    # remove negtive eigenValue
    for i in range(len(eigenValue)):
        if (eigenValue[i] <= 0):
            eigenValue = eigenValue[:i].real
            eigenVector = eigenVector[:, :i].real
            break

    transform = center.T@eigenVector
    save_transform(transform, pathsave, mode, k, 'PCA')

    z_trans = transform.T @ center.T
    reconstruct = transform @ z + mean.reshape(-1, 1)
    index = np.random.choice(135, 10, replace=False)
    save_reconstruct(train_data[index], reconstruct[:, index], pathsave, mode, k, 'PCA')

    # test
    test("PCA", transform, z_trans, train_data, train_label, test_data, test_label, mean, mode)

    return transform, z_trans
```

PCA Steps

After we got the image data, we can define our PCA calculation. With respect to the PCA algorithm and the code represents above.

- Step1: Find the mean

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i$$

- Step2: Remove the mean from the data

$$X = X - \overline{x}$$

- Step3: Form the covariance matrix $S = X^T X$

- By multiplying the data picture centered with its transpose, you could compute the covariance matrix S (to see how the input data are changing from the mean with regard to each other or, in other words, to see if there is any link between them).

$$S = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)(x_i - \mu)^T$$

- Step4: Compute the eigenvalues and the eigenvectors U of S (pxp)

$$Sv_i = \lambda_i v_i, i = 1, 2, \ldots, n$$

- Step5: Order the eigenvectors descending by their eigenvalues.
  - Use only the first k vectors (k <= p)

We will pick K eigenvectors corresponding to the k biggest eigenvalues in order.

➢ Linear Discriminant Analysis (LDA)

While LDA and PCA are similar, LDA attempts to project the data in a manner that maximize between-class distance and minimizes within-class distance.

To maintain the discriminative information and find the combination feature that best separates the classes, LDA focuses on creating a new linear axis and projecting the data points on that axis (blue line). Accordingly, data within the same class should cluster tightly (minimum variance), and data within different classes should be as far away from one another as possible (maximum mean).

```
def LDA(pca_transform, pca_z, train_data, train_label, test_data, test_label, k, mode, pathsave):
    mean = np.mean(pca_z, axis=1)
    N = pca_z.shape[0] # (134, 135)

    S_w = np.zeros((N, N)) #within
    for i in range(15):
        S_w += np.cov(pca_z[:, i*9:i*9+9], bias=True)

    S_b = np.zeros((N, N)) #between
    for i in range(15):
        class_mean = np.mean(pca_z[:, i*9:i*9+9], axis=1).T
        S_b += 9 * (class_mean - mean) @ (class_mean - mean).T

    S = np.linalg.inv(S_w) @ S_b
    eigenValue, eigenVector = np.linalg.eig(S)
    index = np.argsort(-eigenValue) # inverse -> max sort
    eigenValue = eigenValue[index]
    eigenVector = eigenVector[:, index]
```

```
# remove negtive eigenValue
for i in range(len(eigenValue)):
    if (eigenValue[i] <= 0):
        eigenValue = eigenValue[:i].real
        eigenVector = eigenVector[:, :i].real
        break

transform = pca_transform @ eigenVector
save_transform(transform, pathsave, mode, k, 'LDA')

mean = np.mean(train_data, axis=0)
center = train_data - mean
z = transform.T @ center.T
reconstruct = transform @ z + mean.reshape(-1, 1)
save_reconstruct(train_data[index], reconstruct[:, index], pathsave, mode, k, 'LDA')

# test
test("LDA", transform, z, train_data, train_label, test_data, test_label, mean, mode)
```

LDA Steps

- Step1: Calculate the mean classes in our case we have *i = 15* subjects.

$$\mu_i = \frac{1}{|X_i|} \sum_{x_j \in X_i} x_j$$

- Step2: Calculate the total mean in our case we have *i =10* classes of facial expressions

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i$$

- Step3: Calculate the distance between classes *Sb* and distance within classes *Sw*.

$$S_B = \sum_{i=1}^{c} N_i (\mu_i - \mu)(\mu_i - \mu)^T$$

$$S_W = \sum_{i=1}^{c} \sum_{x_j \in X_i} (x_j - \mu_i)(x_j - \mu_i)^T$$

- Step4: The solution of this optimization algorithm is also given by solving the eigenvalues and eigenvectors.

$$S_B v_i = \lambda_i S_w v_i$$

$$S_W^{-1} S_B v_i = \lambda_i v_i$$

  - However, in our experiment, the number of samples *N* is nearly always less than the dimension of the input data, and as a result, the inverse of the *Sw* matrix becomes singular (the number of pixels). However, by first running a PCA on the picture data and projecting samples into *(N-c)* dimensional space, the problem can be resolved. LDA will therefore be run on the reduced data since *Sw* is no longer single.

- We can arrange the eigenvectors in descending order by their eigenvalues after obtaining the eigenvalues and eigenvector. Similar to the previous approach, *PCA*, the k main components (Fisherfaces) are the eigenvectors corresponding to the *k* greatest eigenvalues.

- Step5: Then to generate the fisherfaces we can take its orthogonal projection that defined by this formula.

  $$W = W_{fld}^T W_{pca}^T$$

- Step6: 5th, test the accuracy. The detail will show later. Since LDA will use some outputs from PCA, the function return transform matrix and z_trans.

➢ The kernel function

```
def kernel(mode, data):
    if mode == "none":
        K = np.cov(data, bias=True)
    else:
        if mode == "linear":
            K = data @ data.T
        elif mode == "polynomial":
            K = (0.01 * data @ data.T)**3
        elif mode == "RBF":
            gamma = 0.0001
            dist = cdist(data, data, 'sqeuclidean')
            K = np.exp( -gamma * dist)
        M = data.shape[0]
        MM = np.zeros((M, M))/M
        K = K - MM.dot(K) - K.dot(MM) + MM.dot(K).dot(MM)
```

I set different mode in kernel which are the linear, polynomial and RBF kernels. The kernel function first determines if mode == none. It will return covariance if kernel PCA and kernel LDA are absent.

$$K^C = K - 1_N K - K 1_N + 1_N K 1_N$$

➢ The testing part function

```
def test(testItem, transform, z_trans, train_data, train_label, test_data, test_label, mean, mode):
    test_z = transform.T @ (test_data - mean).T
    dist = np.zeros(train_data.shape[0])
    acc = 0
    for i in range(test_data.shape[0]):
        for j in range(train_data.shape[0]):
            dist[j] = cdist(test_z[:, i].reshape(1, -1), z_trans[:, j].reshape(1, -1), 'sqeuclidean')
        knn = train_label[np.argsort(dist)[:k]]
        uniq_knn, uniq_knn_count = np.unique(knn, return_counts=True)
        predict = uniq_knn[np.argmax(uniq_knn_count)]

        if predict == test_label[i]:
            acc += 1

    if mode == "none":
        print(testItem+f" acc: {100*acc/test_data.shape[0]:.2f}%")
    else:
        print('kernel '+testItem+f" ({mode})"+f" acc: {100*acc/test_data.shape[0]:.2f}%")
```

Both the PCA and LDA use the testing component. It computes z_trans for test data and determines how far apart training and testing z_trans are from one another. To figure out what class the test data belongs to, locate the kth closest neighborhood. at last, achieve precision and accuracy.

II. SNE and t-SNE

a. code with detailed explanations

SNE (Symmetric-SNE)

SNE stands for Stochastic Neighbor Embedding. It is one of machine learning algorithms for dimension reduction. By transforming the high-dimensional **Euclidean distances into conditional probabilities that express similarities**, this technique serves as *dimensionality reduction*.

Therefore, using the cost function of KL Divergence as illustrated below, we project high dimension data into low dimension that will resemble high dimension as closely as possible.

$$C = \sum_i KL(P_i \mid\mid Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

Dimensional Representation

High dimension (p) $p_{ij} = \frac{\exp(- \mid\mid x_i - x_j \mid\mid^2 / (2\sigma^2))}{\sum_{k \neq l} \exp(- \mid\mid x_l - x_k \mid\mid^2 / (2\sigma^2))}$

Low dimensional (q) $q_{ij} = \frac{\exp(- \mid\mid y_i - y_j \mid\mid^2)}{\sum_{k \neq l} \exp(- \mid\mid y_l - y_k \mid\mid^2)}$

- We compare the two spaces and use KL Divergence to discover the minimal cost between the two-dimensional spaces. As a result, we can obtain high similarity in low dimensionality space.

- The normal distribution probability function, also known as the gaussian distribution, is used to calculate the conditional probability of the symmetric SNE.
- such that pji=pij, qji=qij, the main advantage is simplifying gradient

$$\frac{\partial C}{\partial y_i} = 2\sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

- However, in practice we symmetrize (or average) the conditionals, joint probability of picking the pair i, j

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

- have a simpler cost function based on a single KL divergence between a joint probability distribution

$$C = KL(P||Q) = \sum_i \sum_{j \neq i} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

t-SNE (t-Distributed SNE)

The idea of t-SNE is same as SNE. The different between them is t-SNE use **student's t distribution probability function** in low dimensionality space(q) instead of gaussian distribution.

High dimension (p) $p_{ij} = \frac{\exp(-\,||\,x_i - x_j\,||^2\,/(2\sigma^2))}{\sum_{k \neq l} \exp(-\,||\,x_l - x_k\,||^2\,/(2\sigma^2))}$

Low dimensional (q) $q_{ij} = \frac{(1+\,||\,y_i - y_j\,||^2)^{-1}}{\sum_{k \neq l}(1+\,||\,y_i - y_j\,||^2)^{-1}}$

By using the student's t distribution, while retaining local structure, t-SNE also strives to maintain the data's overall structure as well. t-SNE is an advanced version of Symmetric-SNE in preserving the data structure.

1. **My modification**

```
sum_Y = np.sum(np.square(Y), 1)
num = -2. * np.dot(Y, Y.T)
num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y)) # t_SNE
```

```
dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

Base on the understanding of the Symmetric-SNE and t-SNE and the difference of the formula. I did the modification for the code by changing the low dimensional (q) from gaussian distribution probability function to student's t distribution probability function.

2. **Visualization**

- To save distribution result for each 10 iterations in both t_SNE and Symmetric-SNE.
- Read and generate the gif graph and save it after all the results finished

```
def image_save(Y, labels, lim, output_dir, iter):
    pylab.clf()
    pylab.xlim(lim)
    pylab.ylim(lim)
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    pylab.savefig(output_dir+f"{iter}.png")
```

```
def gif_save(output_dir):
    gif = []
    for i in range(100):
        gif.append(Image.open(output_dir+f"{i+1}0.png"))
    gif[0].save(output_dir+f"result.gif",
                format='GIF',
                save_all=True,
                append_images=gif[1:],
                duration=400, loop=0)
```

```
# Compute current value of cost function
if (iter + 1) % 10 == 0:
    C = np.sum(P * np.log(P / Q))
    print("Iteration %d: error is %f" % (iter + 1, C))

    if mode == "t_sne":
        saveImage(Y, labels, [-120, 120], f"./t_sne_output/{int(perplexity)}/", iter+1)
    elif mode == "sym_sne":
        saveImage(Y, labels, [-10, 10], f"./sym_sne_output/{int(perplexity)}/", iter+1)
```

```
X = np.loadtxt("./tsne_python/mnist2500_X.txt")
labels = np.loadtxt("./tsne_python/mnist2500_labels.txt")
for mode in ["sym_sne", "t_sne"]:
    for perplexity in [10., 20., 30., 40., 50.]:
        Y = sne(mode, X, 2, 50, perplexity)
        save_path = "./" + mode + f"_output/{int(perplexity)}/"
        gif_save(save_path)
```

3. **Pairwise similarity plot**

```
def Similarity_save(P, Q, mode, perplexity):
    pylab.subplot(2,1,1)
    pylab.title(mode+" high-dim")
    pylab.hist(P.flatten(),bins=40,log=True)
    pylab.subplot(2,1,2)
    pylab.title(mode+" low-dim")
    pylab.hist(Q.flatten(),bins=40,log=True)
    pylab.savefig("./"+mode+f"_output/similarities_{int(perplexity)}.png")
```

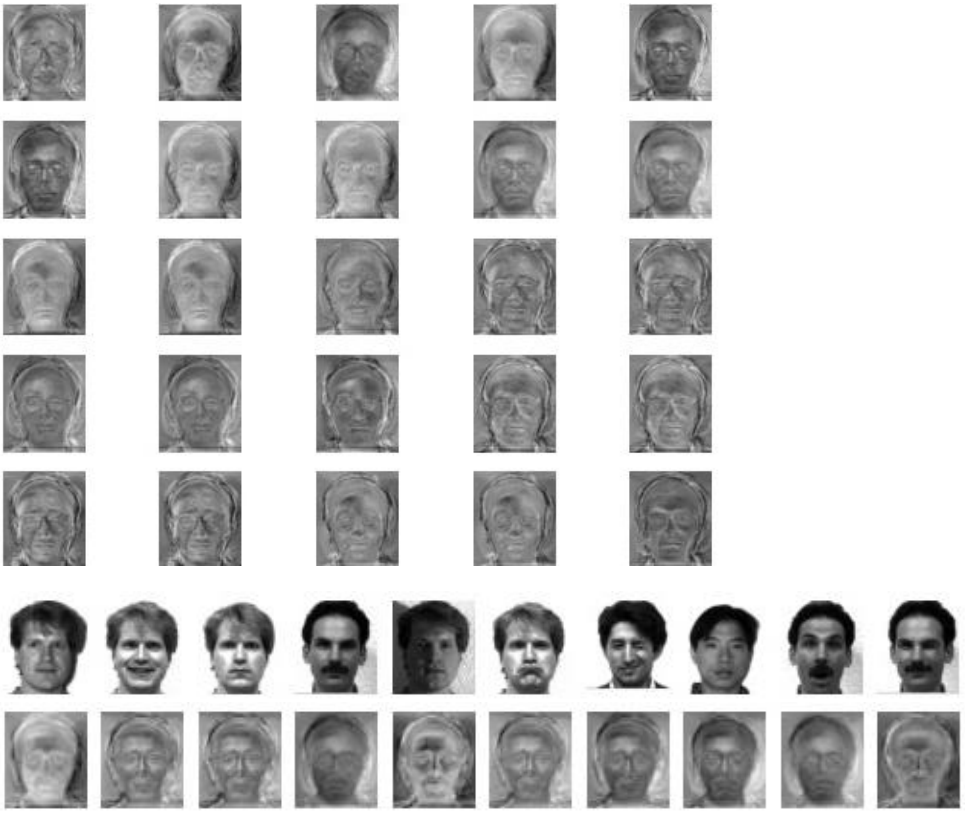
To visualize the similarities in P(high dimension) and Q(low dimension), we use histogram to show the value in each of the perplexity level and save as graph.

b. experiments settings and results

a. Results LDA & PCA

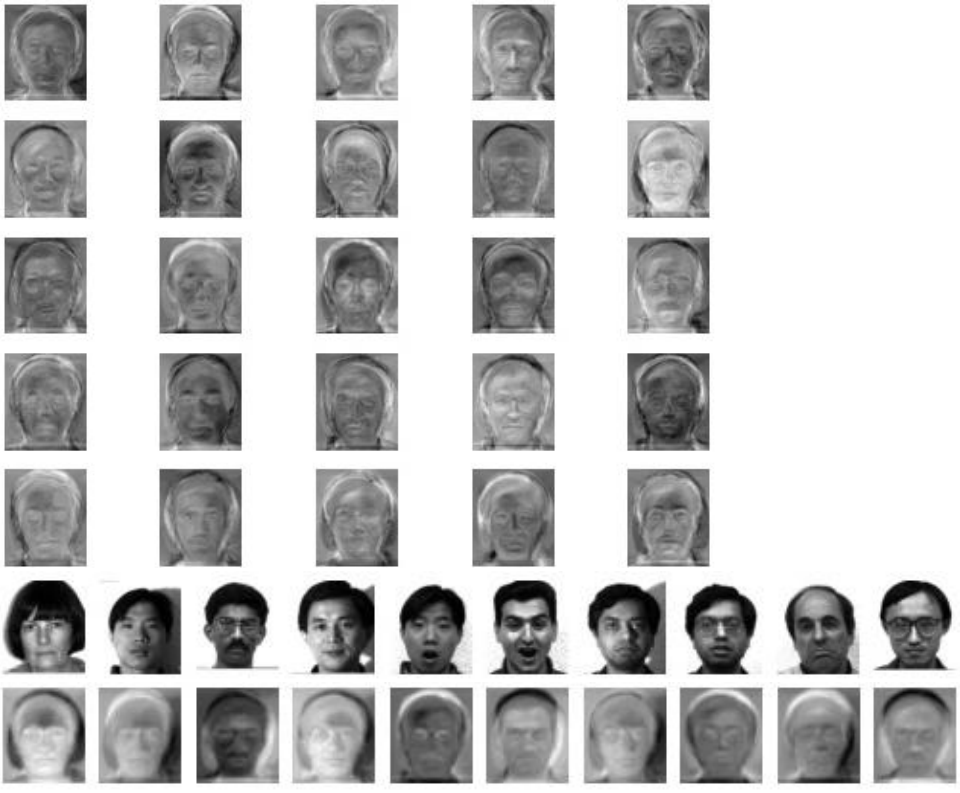
PCA: Eigenface and Reconstruct (10 origin image and 10 reconstruct image):

LDA: Fisherfaces and Reconstruction (10 origin image and 10 reconstruct image):



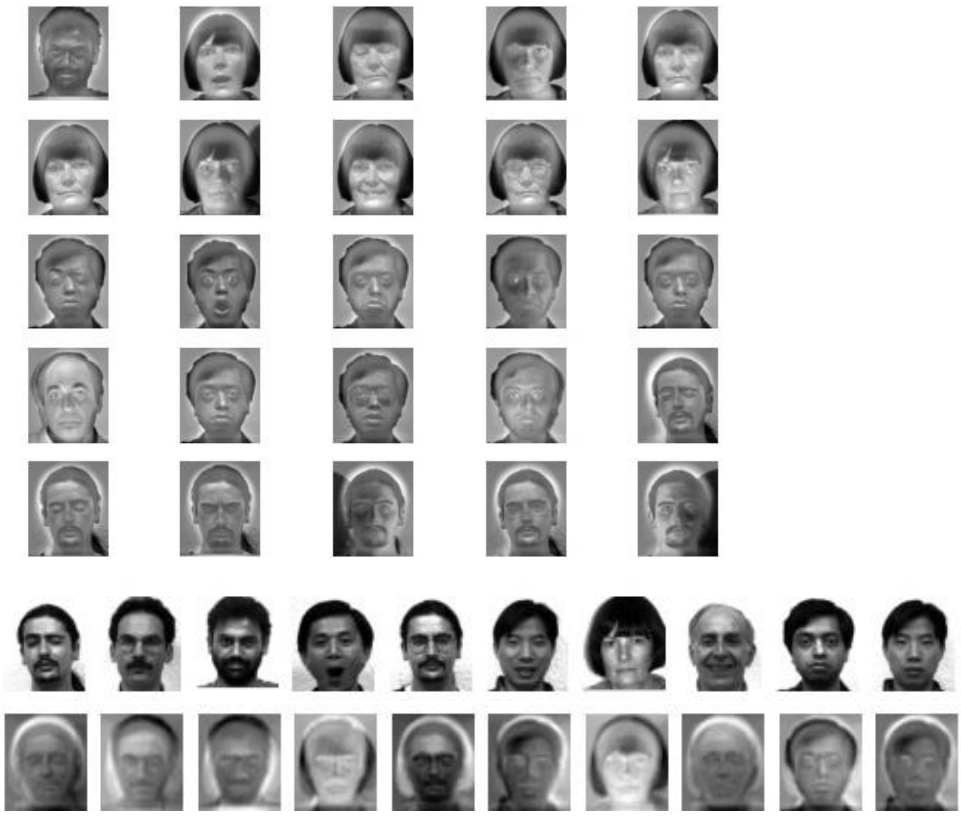
Kernel PCA: Eigenface and Reconstruct (10 origin image and 10 reconstruct image):

Linear



Polynomial

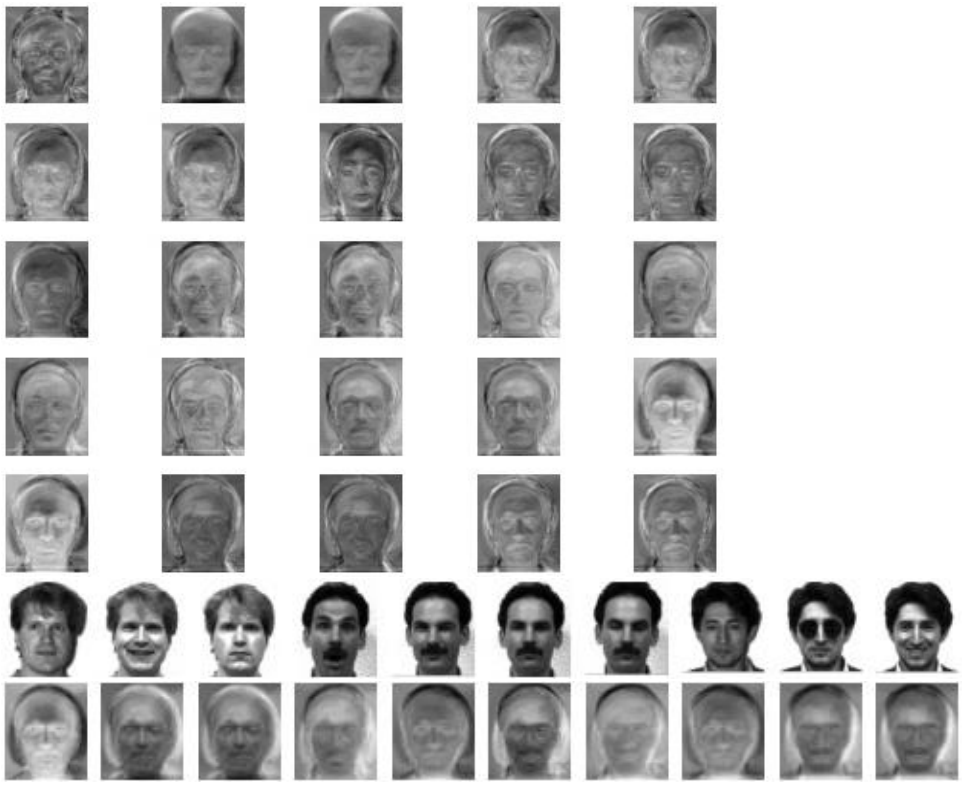
RBF



Kernel LDA: Fisherfaces and Reconstruction (10 origin image and 10 reconstruct image):

Linear

Polynomial



RBF

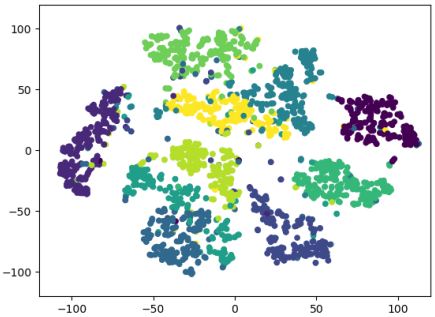
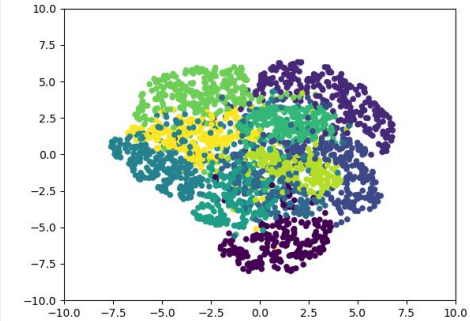
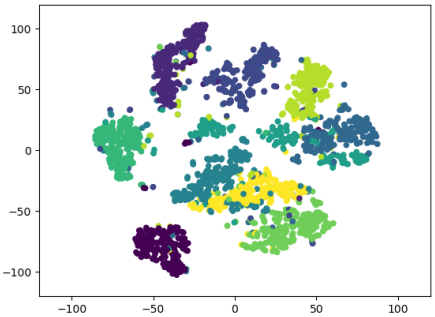
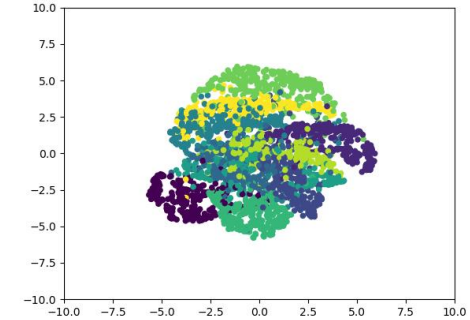
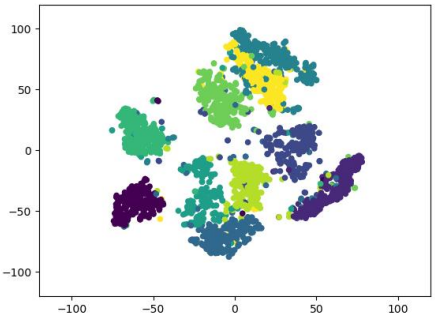
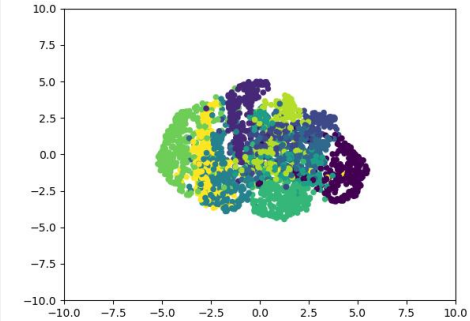
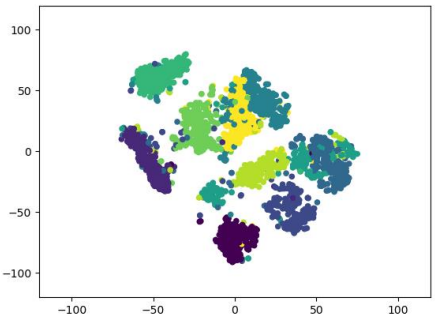
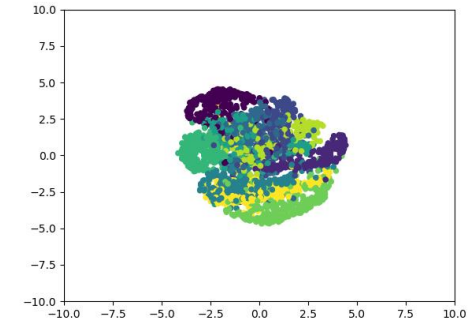
Testing accuracy using different kernel:

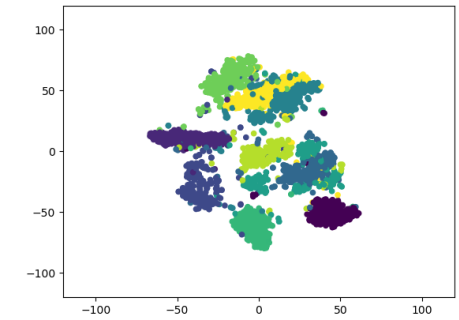
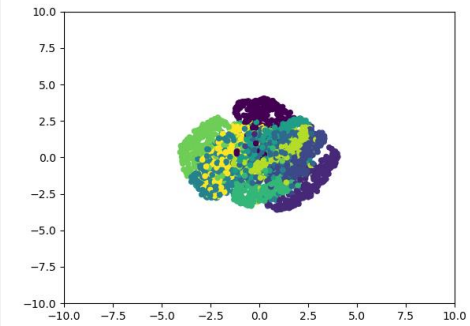
K = 3	K = 4	K = 5
kernel: none PCA acc: 83.33% LDA acc: 80.00% kernel: linear kernel PCA (linear) acc: 80.00% kernel LDA (linear) acc: 76.67% kernel: polynomial kernel PCA (polynomial) acc: 83.33% kernel LDA (polynomial) acc: 66.67% kernel: RBF kernel PCA (RBF) acc: 83.33% kernel LDA (RBF) acc: 76.67%	kernel: none PCA acc: 83.33% LDA acc: 80.00% kernel: linear kernel PCA (linear) acc: 83.33% kernel LDA (linear) acc: 70.00% kernel: polynomial kernel PCA (polynomial) acc: 83.33% kernel LDA (polynomial) acc: 73.33% kernel: RBF kernel PCA (RBF) acc: 83.33% kernel LDA (RBF) acc: 80.00%	kernel: none PCA acc: 83.33% LDA acc: 76.67% kernel: linear kernel PCA (linear) acc: 83.33% kernel LDA (linear) acc: 80.00% kernel: polynomial kernel PCA (polynomial) acc: 83.33% kernel LDA (polynomial) acc: 63.33% kernel: RBF kernel PCA (RBF) acc: 83.33% kernel LDA (RBF) acc: 76.67%

Observations for LDA & PCA

1. It demonstrates that PCA acc are all good regardless of the kernel we deploy. However, linear and polynomial kernels do not perform well with LDA ACC.
2. For human eyes, the eigenface with a higher eigenvalue appears "clearer."
3. Reconstructed faces appear to be all blurry looking.
4. Fisherfaces and Eigenfaces don't resemble the parts of a face. In other words, they are not mustaches, glasses, hairstyles, or any other accessories. However, it makes sense because neither PCA nor LDA restricts the requirement that all values be positive. It may therefore have negative values. Images may now add and subtract objects. As a result, all fisher faces and eigenfaces resemble actual faces.

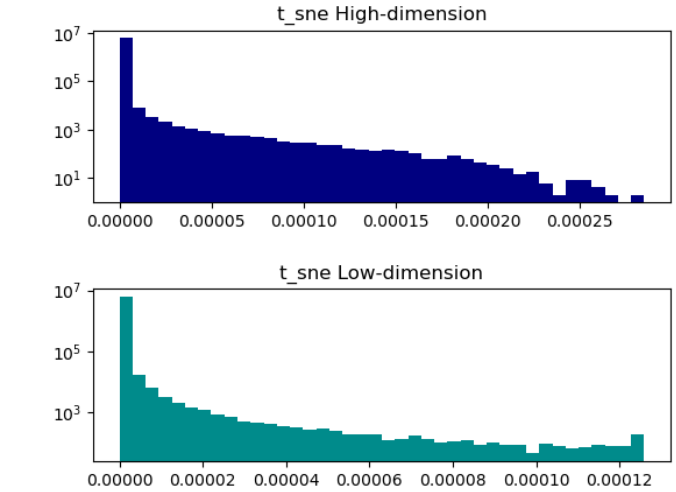
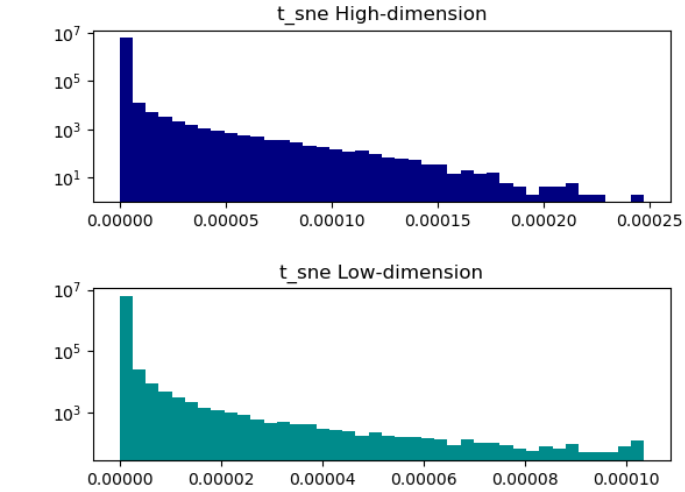
b. Results SNE and t-SNE

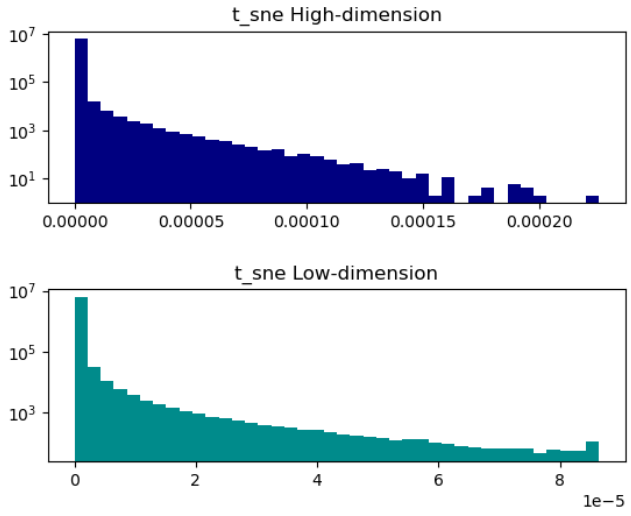
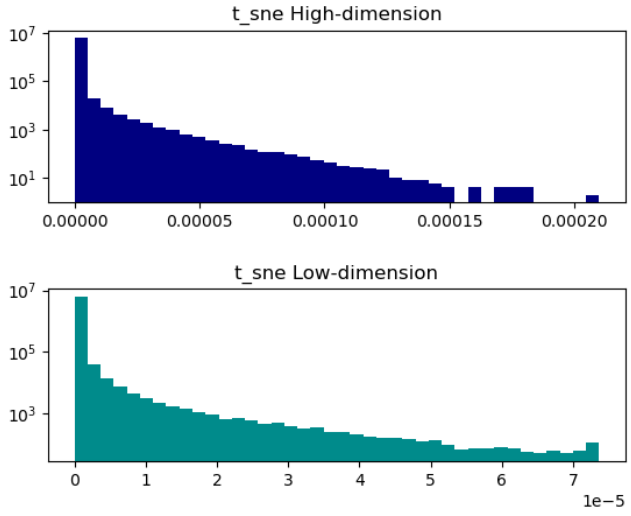
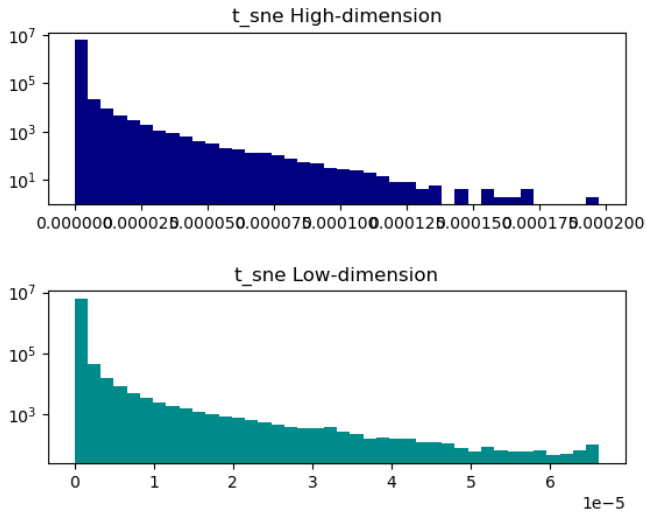
Perplexity	t-SNE	Symmetric SNE
10		
20		
30		
40		

Perplexity	t-SNE	Symmetric SNE
50		

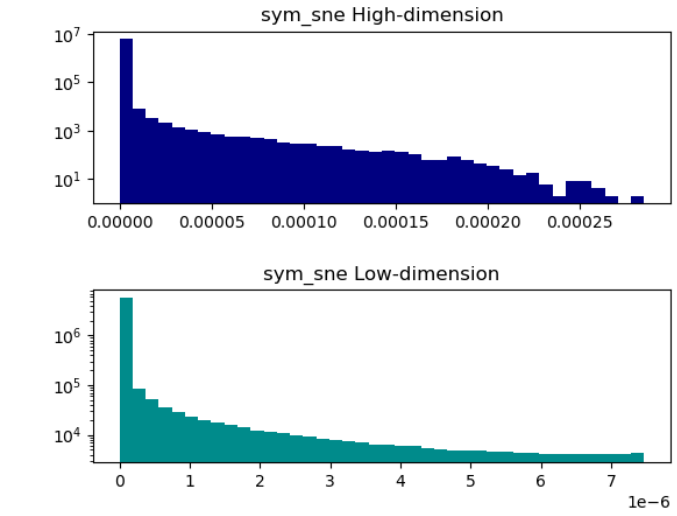
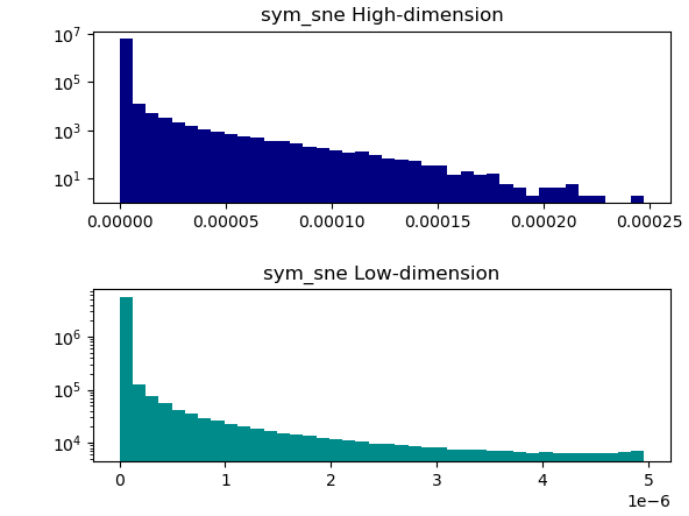
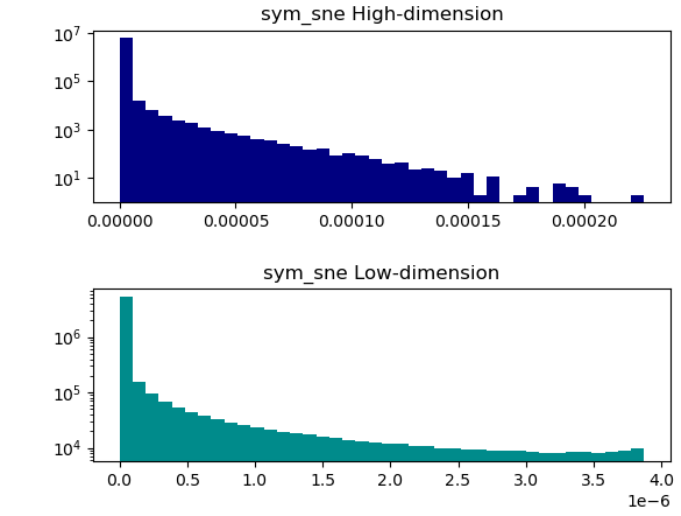
The compassion of similarity in different perplexity values:

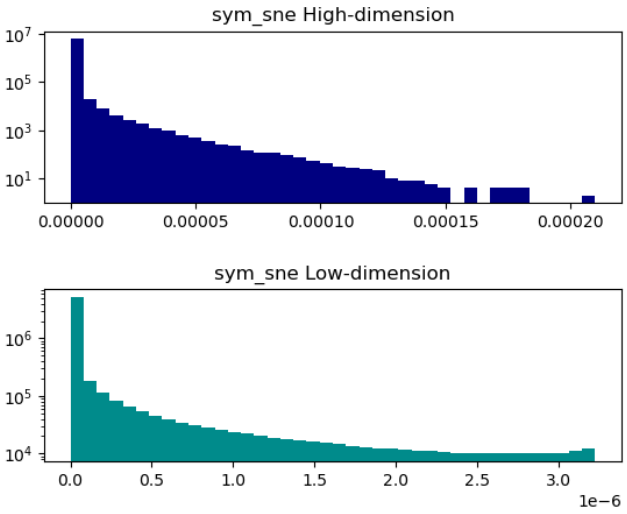
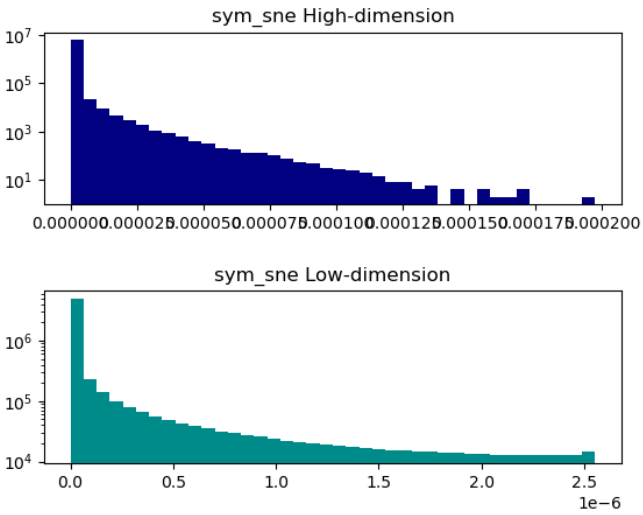
t-SNE:

Perplexity	t-SNE
10	
20	

Perplexity	t-SNE
30	
40	
50	

Symmetric SNE:

Perplexity	Symmetric SNE
10	
20	
30	

Perplexity	Symmetric SNE
40	
50	

Observations for SNE and t-SNE

1. Outcome comparison:
    a. S-SNE results are crowded and packed, suggesting that t-SNE can classify the data more accurately.
    b. High-perplexity outcomes are more congested in both s-SNE and t-SNE results than low-perplexity results.
2. Pairwise similarity comparison:
    a. In both s-SNE and t-SNE, the pairwise similarity is independent in shape with perplexity.
    b. S-SNE results have a wider range of results than t-SNE. so that the crowd issue at t-SNE is reduced.

3. It is clear from the results that symmetric SNE is constrained by crowded problems. It is exceedingly difficult to tell one class from another without color labels. We convert the low-dimension distribution to the t-distribution for t-SNE. Pairwise points that are far apart in high-dimensional space can become

even more distant in low-dimensional space with the aid of this technique. This yields an obvious grouping outcome.

4. Using a gif, we can demonstrate how training groups the data. t-SNE quickly separates several classes into distinct locations in 2D before changing distribution more comprehensively. Compared to SNE, symmetric SNE marginally shifts data in low-dimensional space. The distribution is then improved with a small modification.

c. observations and discussion

The meaning of eigenfaces

- The idea is that anyone's face can be reconstructed from a suitable linear combination of eigenfaces. Someone else's face would have a different combination of the same eigenfaces, whereas your face would be made up of 7% Face A, 3.4% Face B, and so on.Training on a set of genuine faces results in the creation of Eigenfaces. Mathematically identifying "features" and giving each one a strength rating for a specific face is feasible through matrix manipulation. The objective is to produce the fewest eigenfaces necessary to accurately represent the full training set (hence it is a type of Principal Component Analysis). The resulting set of eigenfaces should be able to represent all faces if the training set is sufficiently diverse. The term "eigenfaces" is analogous to "eigenvalues" and "eigenvectors," which are orthonormal basis sets used to, among other things, describe oscillatory system motion. Any periodic system's motion can be explained as the superposition of its eigenvectors.

Crowded problem of symmetric SNE

- For instance, in 10 dimensions, it is feasible to have 11 datapoints that are equally spaced apart, yet a two-dimensional map cannot accurately represent this.
- We thus encounter the following "crowding problem" if the datapoints are somewhat uniformly distributed in the area surrounding I on the 10-dimensional manifold and we attempt to model the distances from I to the other datapoints in the 2-dimensional map.
- 11 datapoints cannot be included in the area of the 2-dimensional map.
- We may resolve this in the low-dimensional map by converting distances into probabilities using a probability distribution that has considerably thicker tails than a Gaussian.
- As a result, a significantly smaller distance in the high-dimensional space may be accurately described.

Anything you want to discuss

for LDA & PCA

1. LDA performs better than PCA when comparing reconstruction results.
2. Reconstruction faces can be used to identify fake faces with ease. However, it can still accurately estimate which subject they fall under. PCA and LDA hold the most crucial information in reserve, particularly LDA. LDA is aware of labels beforehand and is aware of which images are intended for projection into low-dimensional space.

3. As expected, the kernel's performance is superior to that of the non-kernel. This is the same outcome as the prior task; the kernel is a clever trick. The reconstruction appears to use kernel approaches with better detail. The use of linear or RBF has minimal differences. The gamma of the RBF must be chosen, though, or otherwise, everything will turn out gray.

for SNE and t-SNE

1. In both symmetric SNE and t-SNE, perplexity is a significant hyperparameter. It takes the number of neighbors into account. Larger data typically require a higher number for confusion. Data may have more neighbors and wouldn't be as susceptible to tiny groups, thanks to this.
2. It is evident from the outcomes of both symmetric and t-symmetric SNE. Only a data's immediate neighbor can affect it when perplexity decreases. Contrary to previously stated, a structure may become more obvious as complication increases.
3. Because symmetric SNE has a crowded problem, perplexity has a higher impact on t SNE than symmetric SNE does.