Machine Learning HW7

- Python 3.8.5, Spyder
- Packages: numpy, scipy.sptial, matplotlib, imageio, os, re
  - matplotlib for visualization
  - imageio for exporting gif
  - scipy.spatial for calculating euclidean distance
  - PIL for reading image and resize it

Kernel Eigenfaces

code with detailed explanations

Data

In this assignment, we are going to use the **Yale Face Database.zip** contains *165 images of 15 subjects*(subject01, subject02, etc.). There are 11 images per subject, one for each of the following facial expressions or configurations: center-light, w/glasses, happy, left-light, w/no glasses, normal, right-light, sad, sleepy, surprised, and wink.

These data are separated into training dataset(135 images) and testing dataset(30 images). We could resize the images for easier implementation.

Global variable

```
K = [1, 3, 5, 7]
subjectNum = 15

#RESIZE = (231, 195)
#RESIZE = (100, 100)
RESIZE = (60, 50)
```

Read image

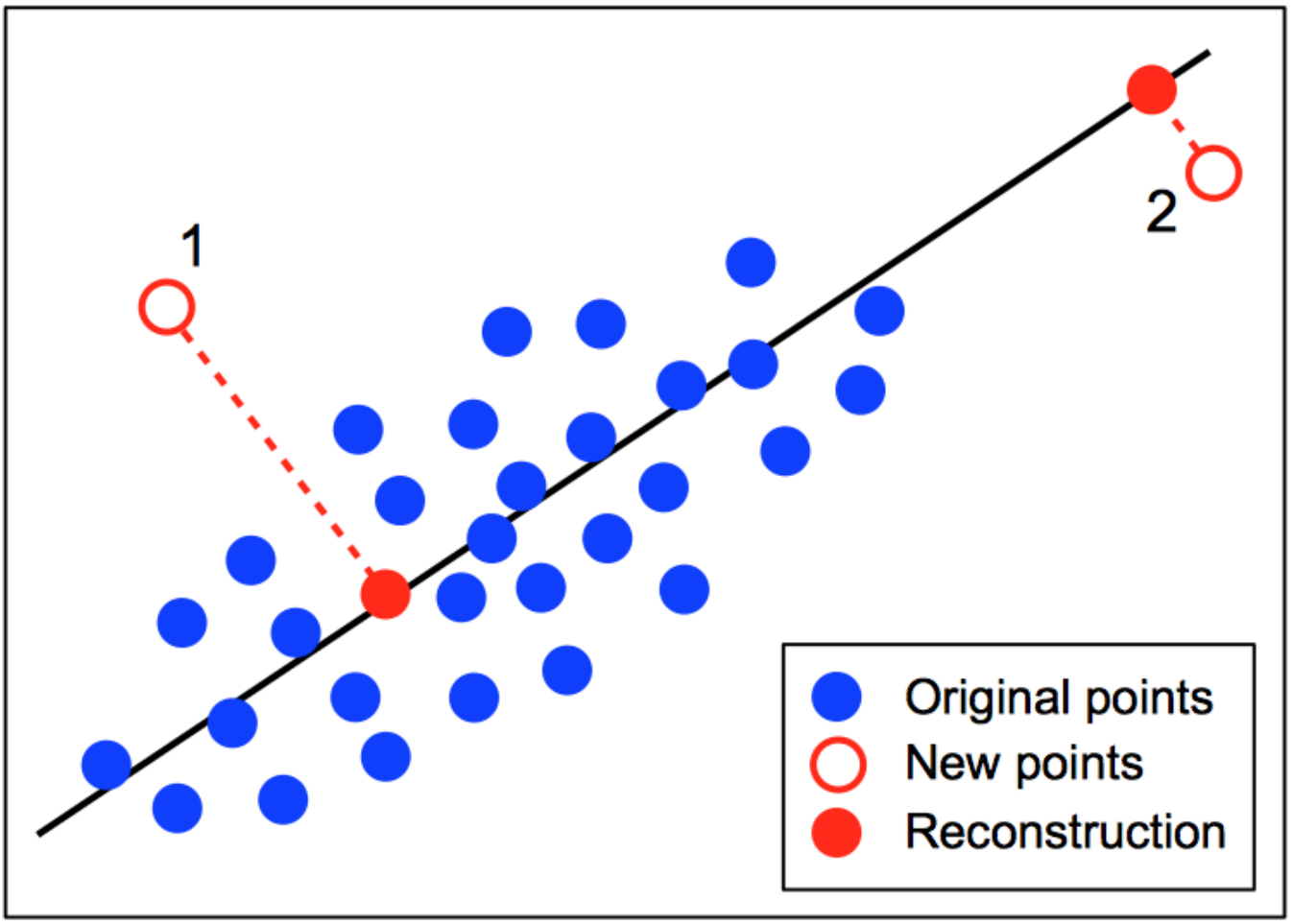
```
from PIL import Image

def readPGM(file):
    with open(file, "rb") as f:
        img = Image.open(f)
        img = img.resize(RESIZE, Image.ANTIALIAS)
        imgArray = np.array(img)
        return imgArray.flatten()

def readFile(filePath):
    filename = []
    label = []
    img = []
    for file in os.listdir(filePath):
        filename.append(file)
        label.append(file[:file.find(".")])
        img.append( readPGM( filePath + "/" + file ) )
    return np.array(filename), np.array(label), np.array(img)
```

function readPGM
Here we use the package PIL to read Image, resize it and return a numpy array.
In the process of resizing, it depends on the global variable - *RESIZE*. The shape of original data image is (231, 195). Also, we would choose the parameter Image.ANTIALIAS for resizing. (ANTIALIAS a high-quality downsampling filter).
After resizing, shape of imgArray would be (width, height). Then we flatten it to be (width*height).
function readFile
Because training data and testing data is inside different folder, the readFile function would read all the file inside the filePath and return all the filename, img and label.

Implementation of PCA and LDA

PCA (Principal componnent analysis)

The goal of PCA is to find an orthogonal projection W (the black line) in the following graph, so that the projected data point (y= Wx) could have maximum variance. (minimum mean square error MSE)




```
def PCA(train, K):
    # average face
    avgFace = np.mean(train, axis = 1)

    # transformation - compute the covariance (every image - average face)
    X = (train.T - avgFace).T
    cov_ = X.T @ X

    # do the eigen decomposition, eigh => eigenValue would be ascending
    eigenValue, eigenVector = np.linalg.eigh(cov_)

    # transformation - train @ eigenvector
    eigenVector = X @ eigenVector

    # normalize the eigenVector => ||w|| = 1
    for i in range(eigenVector.shape[1]):
        eigenVector[ :, i] = eigenVector[ :, i] / np.linalg.norm( eigenVecto

    # select the largest K eigenVector
    W = eigenVector[: , -K:]
    return avgFace, W
```

Parameter train would be the image array.
Paramter K would be used to select K largest eigenvectors.

Step:

1. Compute the average face. (mean of image data)
2. Let X equal to be the difference between train data and average face
3. Compute the covariance.
4. Do the eigen decomposition of covariance.
    - Here we use some trick because the dimension of image would be huge and hard to calculate. For example a face image (256, 256), the shape of covariance matrix would become (66536, 66536). What's more, if there is M persons. the covariance would be computed by the (66536, M) @ (M, 66536). It would be hard and time-consuming to do the eigen decomposition.
    $A^TAv_i = \mu_i v_i$
    $\Rightarrow AA^TAv_i = \lambda_i Av_i$
    $\Rightarrow u_i = Av_i \ and \ \lambda_i = \mu_i$
    - original data would be A^TA shape would be (66536, 66536), then we transform it by both multiply A. So we could use the AA^T whose shape is (135, 135). (number of training images). Then do the eigen decomposition and get the eigevector. Then we multipy the eigenvector by A. So we could get the original eigen vector.
5. Normalize the eigenvector. (so that $||w|| = 1$)
6. extract the K largest eigenvectors.

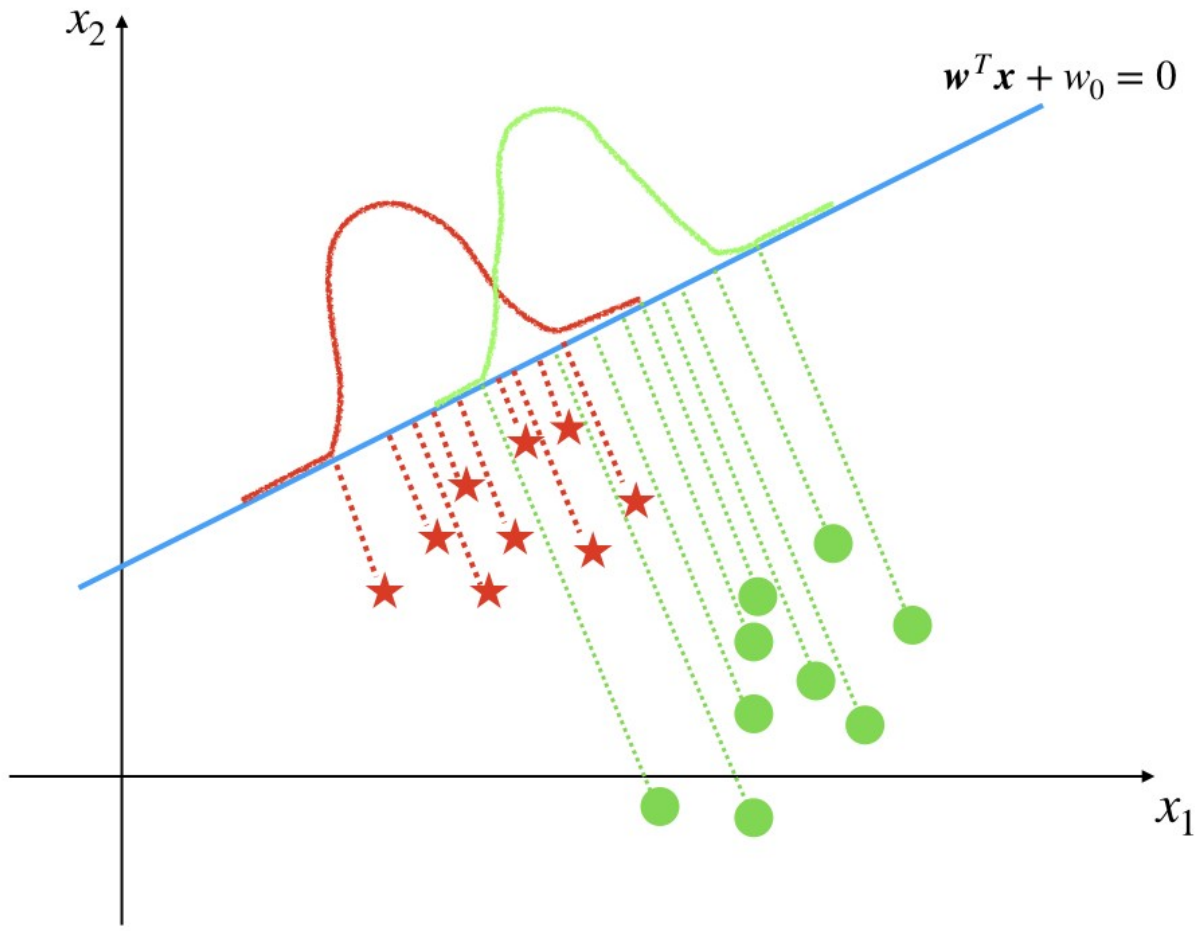
LDA (Linear Disciminative Analysis)

In LDA we want maximize between-class scatter and minimize within-class scatter.

we want to project data onto a line parameterized by a unit vector w: $y = w^Tx$ such that projected data of C1 is maximally separated from projected data of C2

$S_{between-classes} = S_b = \sum_{i=1}^{c} N_i(\mu_i - \mu)(\mu_i - \mu)^T$
$S_{within-classes} = S_w = \sum_{i=1}^{c} \sum_{x_j \in X_c}(x_j - \mu_i)(x_j - \mu_i)^T$

between-class scatter means how these classes are separated in low D.
within-class sactter means the variance in each class.


x_2
$\boldsymbol{w}^T\boldsymbol{x} + w_0 = 0$
x_1

```
def LDA(train, K):
    # how many cluster
    imgNum = train.shape[1] // subjectNum
    mean = np.zeros(( subjectNum, train.shape[0]))
    for i in range(subjectNum):
        mean[i] = np.mean(train[:, i * imgNum : (i+1) * imgNum], axis=1)

    overallMean = np.mean(train, axis=1)

    # compute the between-class scatter and within-class scatter
    Sw = np.zeros((train.shape[0], train.shape[0]))
    Sb = np.zeros((train.shape[0], train.shape[0]))

    for index, value in enumerate(np.unique(label)):
        xi = train[:, np.where(label == value)[0]].T
        dist1 = xi - mean[index]
        Sw += dist1.T @ dist1

        dist2 = (mean[index] - overallMean).reshape(1, -1)
        Sb += len(np.where(label == value)[0]) * dist2.T @ dist2

    # compute for the eigenVector
    fisherValue, fisherVector = np.linalg.eig(np.linalg.pinv(Sw) @ Sb)

    #select the largest K fisherVector
    index = np.argsort(fisherValue)[::-1]
    W = fisherVector[:, index[: K]].real
    return overallMean, W
```

Step:

1. imgNum for counting how many data belong to each cluster. In this assignment, imgNum equals to 9.
2. compute the mean of each cluster and overall mean.
   - μ: overall average face.
   - μ_i: average face of each cluster.
     $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$
     $\mu_i = \frac{1}{|X_i|}\sum_{x_j \in X_i} x_j, i \in \{1, \dots, c\}$
3. compute the between classes scatter S_b and within-class scatterS_w.
   - In the code xi would be training data which belong to cluster i.
   - dist1 = xi - cluster mean.
   - S_w : summation of dist1.T @ dist1.
   - dist2 = cluster average face - overall average face.
   - S_b : summation of dist2.T @ dist2 multiply N(number of data in cluster i) &.

4. do the eigen decomposition.
    - $arg\max_W \frac{|W^T S_B W|}{|W^T S_W W|} \Rightarrow S_W^{-1} S_B v_i = \lambda_i v_i$
5. extract the k largest eigenValue and eigenVector.
    - only get the real value

Implementation of kernel PCA and LDA

Here we would use 3 type of kernel:

1. linear $K(x, z) = x^T z$
2. polynomial: $K(x, z) = (\gamma x^T z + \gamma)^d, \gamma > 0$
3. radial basis function(RBF): $K(x, z) = e^{-\gamma||x-z||^2}$

kernel method

```
def linearKernel(X, Y):
    return X @ Y.T

def polynomialKernel(X, Y, gamma=1e-2, coef=0.1, degree=2):
    return np.power(gamma * (X @ Y.T) + coef, degree)

def rbfKernel(X, Y, gamma=1e-8):
    return np.exp(-gamma * scipy.spatial.distance.cdist(X, Y, 'sqeuclidean')
```

Kernel method apply to PCA and LDA.
referencce : Kernel Eigenfaces vs. Kernel Fisherfaces: Face Recognition Using Kernel Methods (https://www.csie.ntu.edu.tw/~mhyang/papers/fg02.pdf)

Kernel PCA

```
def KernelPCA(train, K, method):
    # compute kernel
    kernel = method(train.T, train.T)
    eigenValue, eigenVector = np.linalg.eigh(kernel)

    index = np.argsort(eigenValue)[::-1]
    # already sorted due to the implementation of eigh
    W = eigenVector[:, index[:K]].real
    return W, kernel
```

Eigenvalue problem of covariance matrix in feature space:
$K\alpha = \lambda N\alpha$ **assume centered already.**
If the K is not centered already, we need to do some calculation.
$K^c = K - 1_N K - K1_N + 1_N K 1_N$
1_N is NxN matrix with every element 1/N.

1. In the code, parameter method is for kernel type. After the method, it would return Kernel Gram Matrix.
2. Because this gram matrix is not centered already, we would transform it by applying the above calculation.
3. Then again, do the eigen decomposition of centered kernel gram matrix.
4. get the k largest eigenvector.

kernel LDA

```
def KernelLDA(train, K, method):
    _kernel = method(train.T, train.T)

    # compute the between-class scatter and within-class scatter
    Z = np.full((_kernel.shape[0], _kernel.shape[0]), 1 / _kernel.shape[0])
    Sb = _kernel @ Z @ _kernel
    Sw = _kernel @ _kernel

    # compute for the eigenVector
    fisherValue, fisherVector = np.linalg.eig(np.linalg.pinv(Sw) @ Sb)

    #select the largest K fisherVector
    index = np.argsort(fisherValue)[::-1]
    W = fisherVector[:, index[: K]].real
    return W, _kernel
```

Eigenvalue problem in feature spaces would be:
Refer to the paper above. (kernel fisherfaces)

> $\lambda KK\alpha = KZK\alpha$
> $S_b = KZK, Sw = KK$
> Z is $l_t \times l_t$ matrix with terms all equal to $\frac{1}{l_t}$.

In the implementation:

1. get the gram matrix
2. compute the Z and then get the S_b and S_w.
3. do the eigen decomposition of $(S_w)^{-1}(S_b)$.
4. return the k largest eigen vector and the kernel gram matrix.

Part I show the eigenfaces, fisherfaces and reconstruction.

show the first 25 eigenfaces and fisherfaces, and randomly pick 10 images to show their reconstruction.

```
# PCA
avgFace, eigenFace = PCA(train, 25)
randIndex = np.random.choice(train.shape[1], 10, replace=False)

# ploting
drawEigenFace(eigenFace, 25, "PCA/PCA EigenSpace")
drawReconstructFace(avgFace, eigenFace, train, filename, randIndex, "PCA/PCA

# # LDA
avgFace, fisherFace = LDA(train, 25)
randIndex = np.random.choice(train.shape[1], 10, replace=False)

# ploting
drawEigenFace(fisherFace, 25, "LDA/LDA fisherSpace1")
drawReconstructFace(avgFace, fisherFace, train, filename, randIndex, "LDA/LD
```

Here we would visualize eigenfaces and fisherfaces. And do the face reconstructions of 10 randomly choosen train images.

function drawEigenFace

```
def drawEigenFace(eigenFace, K, title):
    # for ploting multiple figures
    fig, ax = plt.subplots(5, 5, figsize=(8, 8), squeeze=False)
    fig.tight_layout(pad = 3.0)

    fig.suptitle(title, fontsize=16)
    for i in range(K):
        # original image
        r, c = i // 5, i % 5
        ax[r][c].imshow(eigenFace[:, i].reshape(RESIZE[1], RESIZE[0]), cmap=
        ax[r][c].set_title(title[3:] + "_" + str(i))

    plt.subplots_adjust(top=0.9)
    plt.savefig(title)

```

plt.subplots would be used because we would like to show multiple figures in a image. Because at the first we flatten the images to be vectors, here if we want to visualize the image vectors. It would be reshaped.

function drawReconstructFace

```
def drawReconstructFace(avgFace, eigenFace, train, filename, randIndex, titl
    # for ploting multiple figures
    fig, ax = plt.subplots(4, 5, figsize=(12, 8), squeeze=False)
    fig.tight_layout(pad = 3.0)
    fig.suptitle( title + ' Face Reconstruct', fontsize=16)

    for index, value in enumerate(randIndex):
        img = train[:, value]
        projection = eigenFace.T @ (img - avgFace)
        reFace = eigenFace @ projection + avgFace

        r, c = index // 5 * 2, index % 5
        ax[r][c].imshow(train[:, value].reshape(RESIZE[1], RESIZE[0]), cmap=
        ax[r][c].set_title("original")

        ax[r+1][c].imshow(reFace.reshape(RESIZE[1], RESIZE[0]), cmap="gray")
        ax[r+1][c].set_title(filename[value])
    plt.subplots_adjust(top=0.9)
    plt.savefig(title + "_Face_Reconstruct.png")
```

Iterate the index of 10 images and reconstruct it. First computed projection of each face and reconstruct it.
project the X to low dimension:
$Y = W^T(X - \mu)$
Reconstruct it: low dimension -> high dimension
$X' = WY\mu$

Part II do face recognition and compute the performance.

Use PCA and LDA to do face recognition, and compute the performance. You should use k nearest neighbor to classify which subject the testing image belongs to.

```
# PCA
avgFace, eigenFace = PCA(train, 25)
train_proj = eigenFace.T @ (train.T - avgFace).T
test_proj = eigenFace.T @ (test - avgFace).T
print("-"*10, "PCA", "-"*10)
predictFaceRecong(train_proj, label, test_proj, testLabel)

# LDA
avgFace, fisherFace = LDA(train, 25)
train_proj = fisherFace.T @ (train.T - avgFace).T
test_proj = fisherFace.T @ (test - avgFace).T
print("-"*10, "LDA", "-"*10)
predictFaceRecong(train_proj, label, test_proj, testLabel)
```

train_proj: project training image to low dimension.
test_proj: project testing image to low dimension.
Do the face recognition and output the performance of prediction.

function predictFaceRecong

```
def predictFaceRecong(train_proj, label, test_proj, testLabel):
    dist = scipy.spatial.distance.cdist(test_proj.T, train_proj.T, 'euclidea

    # different k -> k for nearest neighbor
    for k in K:
        truePredict = 0
        for i in range(dist.shape[0]):
            row = dist[i, :]
            sortIndex = np.argsort(row)[: k]
            neighbor = label[sortIndex]
            face, count = np.unique(neighbor, return_counts=True)
            predict = face[np.argmax(count)]
            if predict == testLabel[i]:
                truePredict += 1
            #print("predict : ", predict, "// True : ", testLabel[i])
        print(f'K neighbors={k}, Accuracy: {truePredict / test_proj.shape[1]
```

- First we compute the euclidean distance between test (after projection) and train (after projection).
  - K is the global variable for runing different value of k nearest neighbor.
- iterate each test image and get the smallest k distance. (nearest neighbors)
- count the label of each k nearest points. get the largest value to be the predicted label.
- compare the predicted label and the true label. If the value is equal, truePredict add 1. Paramter truePredict is the total times of ture predicting.

Part III Use kernel PCA and kernel LDA to do face recognition.

```
# Kernel PCA and LDA
kernel = [linearKernel, polynomialKernel, rbfKernel]

# Before doing kernel -> make the data centered!!!
avgFace = np.mean(train, axis=1)
train = (train.T - avgFace).T
test = test - avgFace

for method in kernel:
    # PCA
    eigenFace, GramMatrix = KernelPCA(train, 25, method)
    train_proj = GramMatrix @ eigenFace

    testGramMatrix = method(test, train.T)
    test_proj = testGramMatrix @ eigenFace
    print("-"*10, "Kernel PCA - ", str(method), "-"*10)
    predictFaceRecong(train_proj.T, label, test_proj.T, testLabel)

for method in kernel:
    # LDA
    fisherFace, GramMatrix  = KernelLDA(train, 25, method)
    train_proj = (GramMatrix) @ fisherFace

    testGramMatrix = method(test, train.T)
    test_proj = testGramMatrix @ fisherFace
    print("-"*10, "Kernel LDA - ", str(method), "-"*10)
    predictFaceRecong(train_proj.T, label, test_proj.T, testLabel)
```

In the Kernel PCA, we would get the eigenFace and kernel gram matrix.

1. Before mapping the training data to feature space, we need to **centralize the training data. train = train - avgFace, test = test - avgFace**
2. Use the Gram Matrix dot eigenFace to get the projection of training data in feature space to low dimension.
3. get the Gram Matrix between test data and training data. And then project them.
4. predict them.

For computation

In the Kernel LDA, we would get the eigenFace and kernel gram matrix, too.

1. follow the same step in PCA, and predict it.

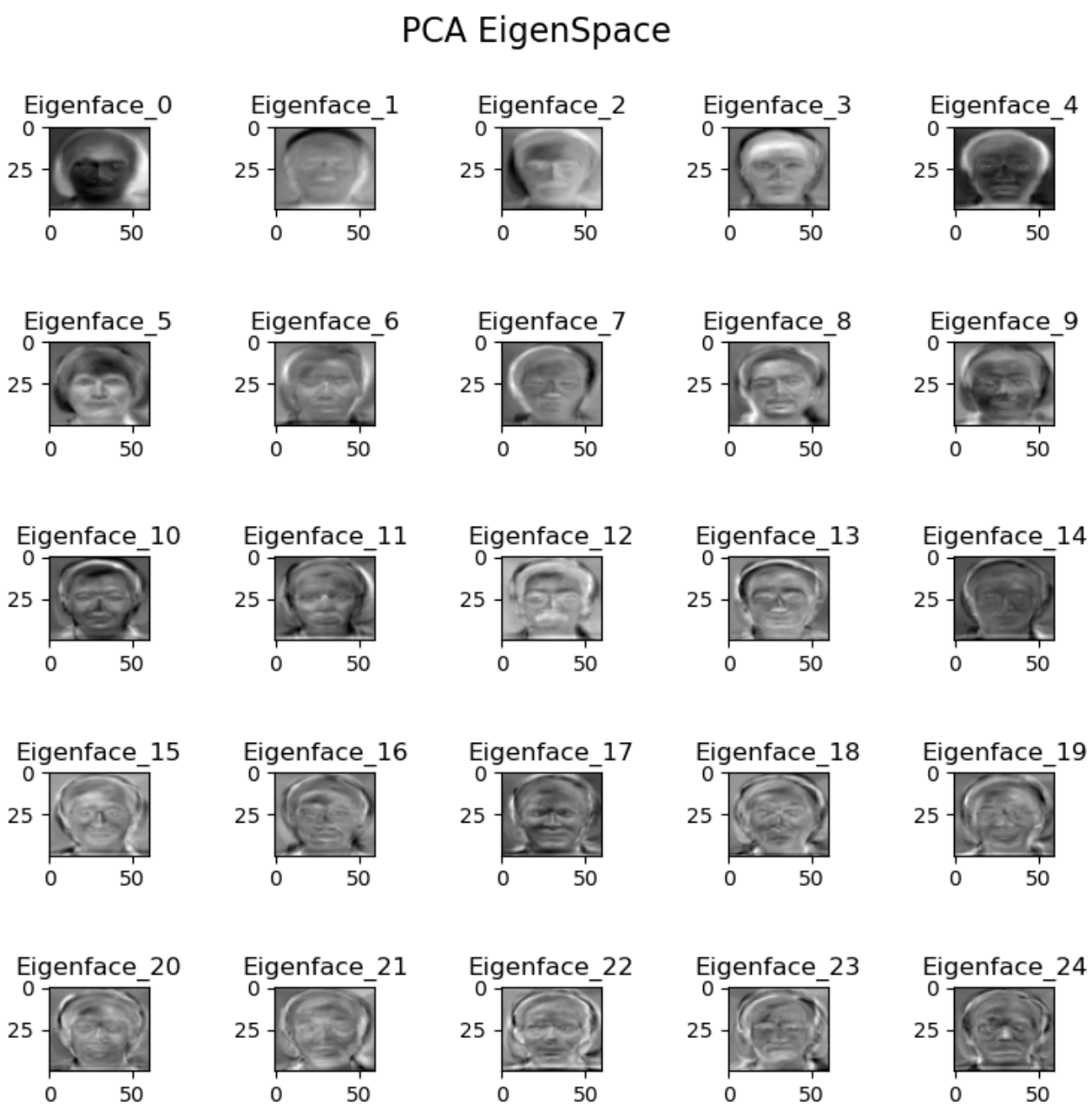
Experiments settings and results & discussion

Part I show the eigenfaces, fisherfaces and reconstruction.

take K=25 largest eigenfaces.

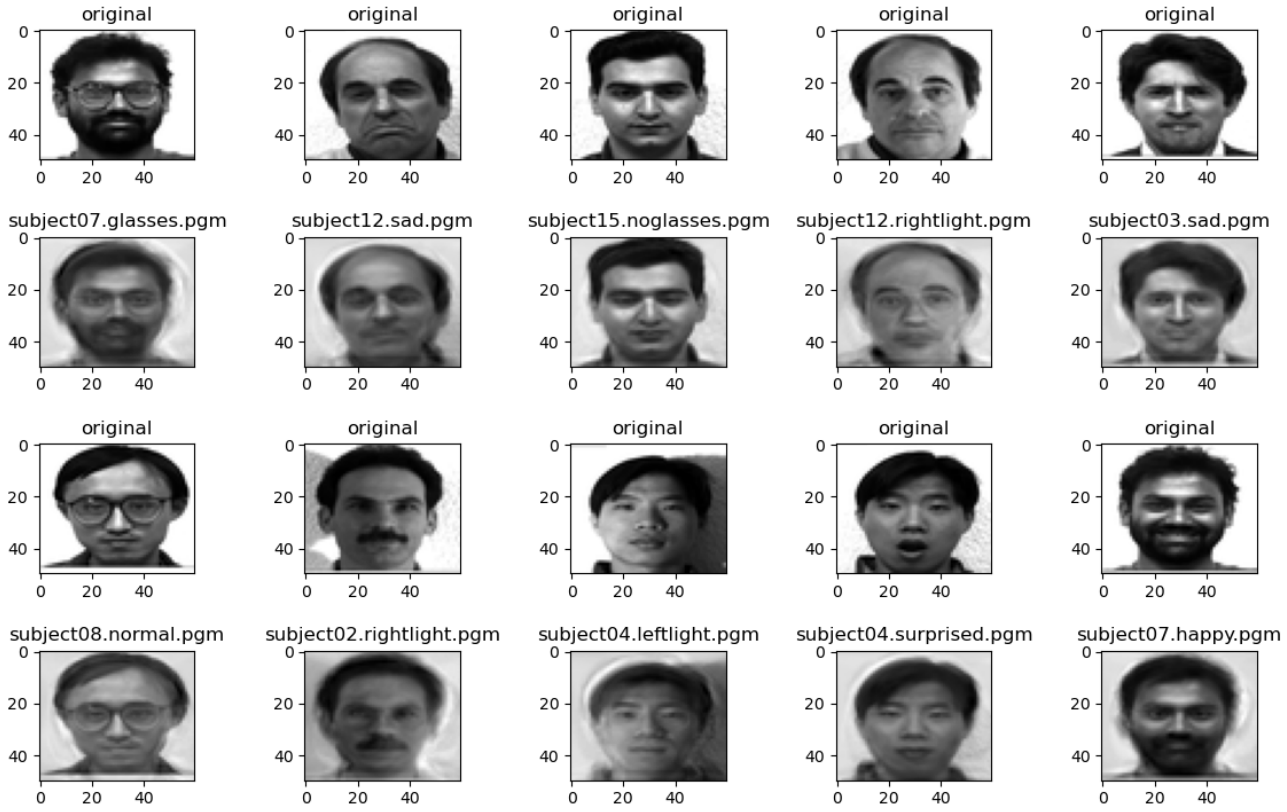
PCA

Eigen Face



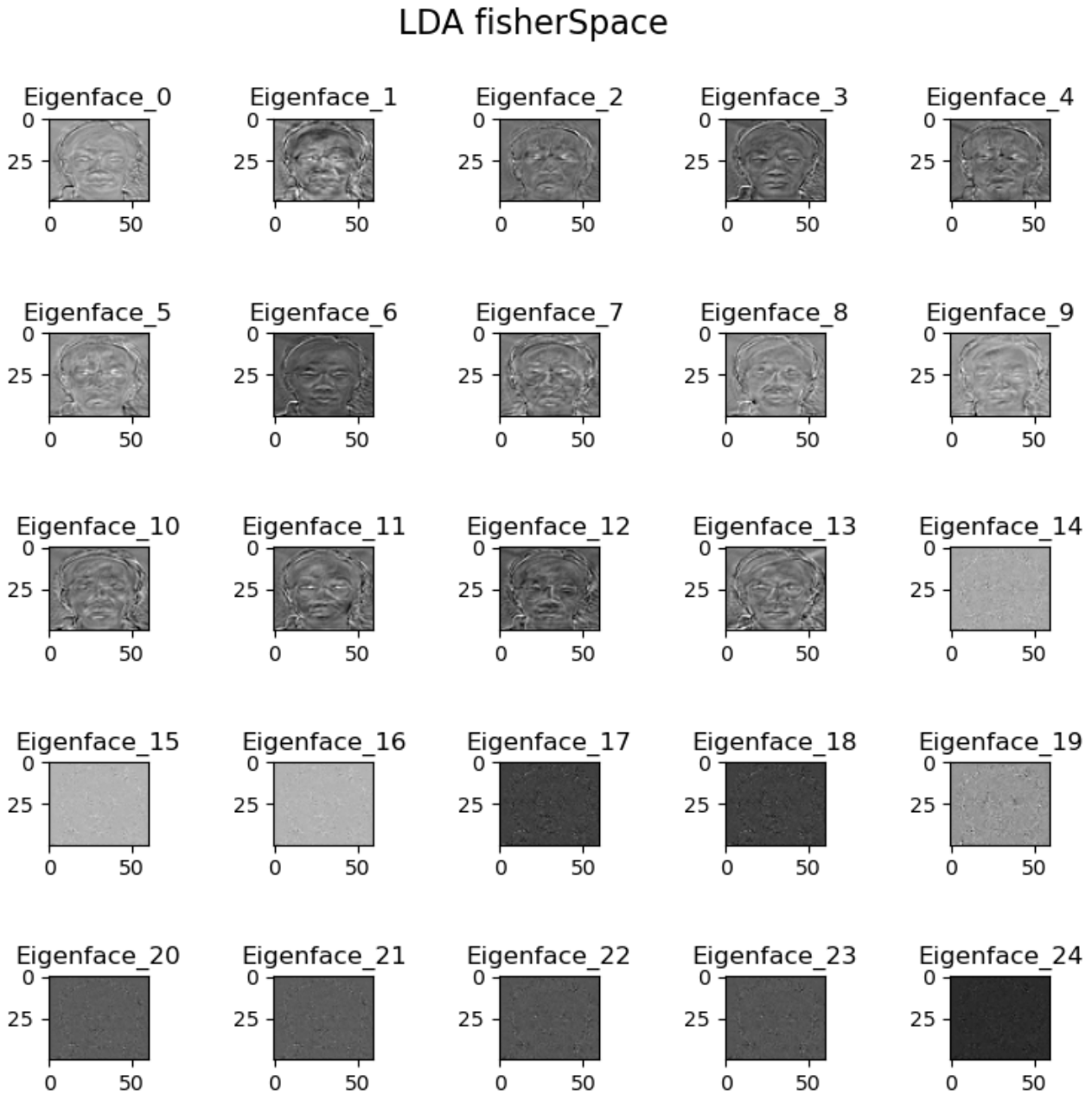
Face reconstruction

PCA/PCA Face Reconstruct

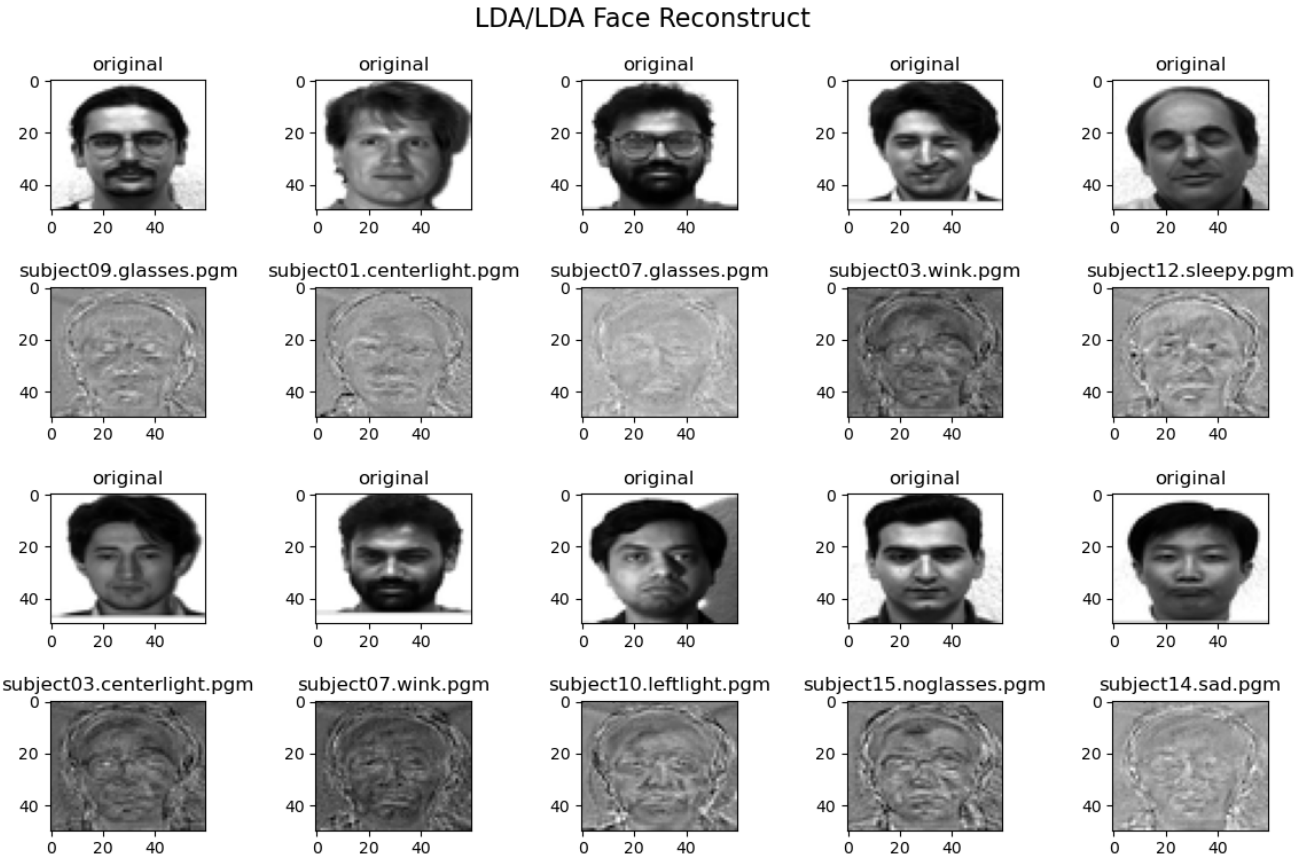


LDA

Fisher Face



Face reconstruction



Part II do face recognition and compute the performance.

```
---------- PCA ----------
K neighbors=1, Accuracy: 0.833 (25/30)
K neighbors=3, Accuracy: 0.833 (25/30)
K neighbors=5, Accuracy: 0.867 (26/30)
K neighbors=7, Accuracy: 0.900 (27/30)
---------- LDA ----------
K neighbors=1, Accuracy: 0.967 (29/30)
K neighbors=3, Accuracy: 0.967 (29/30)
K neighbors=5, Accuracy: 0.967 (29/30)
K neighbors=7, Accuracy: 0.967 (29/30)
```

Part III Use kernel PCA and kernel LDA to do face recognition.

Then compare the difference between simple LDA/PCA and kernel LDA/PCA, and the difference between different kernels.

hyperparameter of kernel:

- linear: none
- polynomial: gamma=1e-2, coef=0.1, degree=2
- gamma=1e-8

Kernel PCA:

```
---------- Kernel PCA -  <function linearKernel at 0x00000249918EE1F0> ----------
K neighbors=1, Accuracy: 0.800 (24/30)
K neighbors=3, Accuracy: 0.833 (25/30)
K neighbors=5, Accuracy: 0.800 (24/30)
K neighbors=7, Accuracy: 0.800 (24/30)
---------- Kernel PCA -  <function polynomialKernel at 0x00000249918ED280> ----------
K neighbors=1, Accuracy: 0.833 (25/30)
K neighbors=3, Accuracy: 0.867 (26/30)
K neighbors=5, Accuracy: 0.867 (26/30)
K neighbors=7, Accuracy: 0.833 (25/30)
---------- Kernel PCA -  <function rbfKernel at 0x00000249918EDCA0> ----------
K neighbors=1, Accuracy: 0.833 (25/30)
K neighbors=3, Accuracy: 0.767 (23/30)
K neighbors=5, Accuracy: 0.800 (24/30)
K neighbors=7, Accuracy: 0.733 (22/30)
```

Kernel LDA:

```
---------- Kernel LDA -  <function linearKernel at 0x00000249918EE1F0> ----------
K neighbors=1, Accuracy: 0.800 (24/30)
K neighbors=3, Accuracy: 0.800 (24/30)
K neighbors=5, Accuracy: 0.800 (24/30)
K neighbors=7, Accuracy: 0.667 (20/30)
---------- Kernel LDA -  <function polynomialKernel at 0x00000249918ED280> ----------
K neighbors=1, Accuracy: 0.700 (21/30)
K neighbors=3, Accuracy: 0.667 (20/30)
K neighbors=5, Accuracy: 0.600 (18/30)
K neighbors=7, Accuracy: 0.600 (18/30)
---------- Kernel LDA -  <function rbfKernel at 0x00000249918EDCA0> ----------
K neighbors=1, Accuracy: 0.767 (23/30)
K neighbors=3, Accuracy: 0.800 (24/30)
K neighbors=5, Accuracy: 0.800 (24/30)
K neighbors=7, Accuracy: 0.767 (23/30)
```

discussion

PCA vs. LDA

k = 1 ~ 7, LDA is all higher than all the PCA.
LDA precisely predict. (29/30, 0.967%)
PCA performs well too. (24/30, 0.800%)
But for drawing the

PCA, LDA vs. Kernel PCA, LDA

In my experiment the original one perform better.
Actually, in my assumption the kernel version should outperform than the original one.
But the result is not unexpected. Maybe the reason might due to not proper hyperparameter or the data is not dealed well to be center. (I centeralize the data before mapping to the feature space.)

different Kernel

In PCA, the polynomial kernel perform better than others.
In LDA, the rbf Kernel perform better than others.

t-SNE

code with detailed explanations

data

Data & reference code: https://lvdmaaten.github.io/tsne/code/tsne_python.zip (https://lvdmaaten.github.io/tsne/code/tsne_python.zip),

- mnist2500_X.txt: contains 2500 feature vectors with length 784, for describing 2500 mnist images.
- mnist2500_labels.txt: provides corresponding labels
- tsne.py (http://tsne.py): reference code

Part1: Try to modify the code a little bit and make it back to symmetric SNE.

(explain the difference between symmetric SNE and t-SNE in the report (e.g. point out the crowded problem of symmetric SNE).)

The major difference between symmetric SNE and t-SNE is that the **t-SNE would use the Student t-disb to turn distances in low dimension into probability** instead of Gaussian disb.

same:distance in high Dimension **all Gaussian distribution**

$$p_{ij} = \frac{exp(-||x_i - x_j||^2/(2\sigma^2))}{\sum_{k \neq l} exp(-||x_i - x_j||^2/(2\sigma^2))}$$

difference t-SNE: **Student t distribution in low D**

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

difference symmetric SNE: **Gaussian distribution in low D**

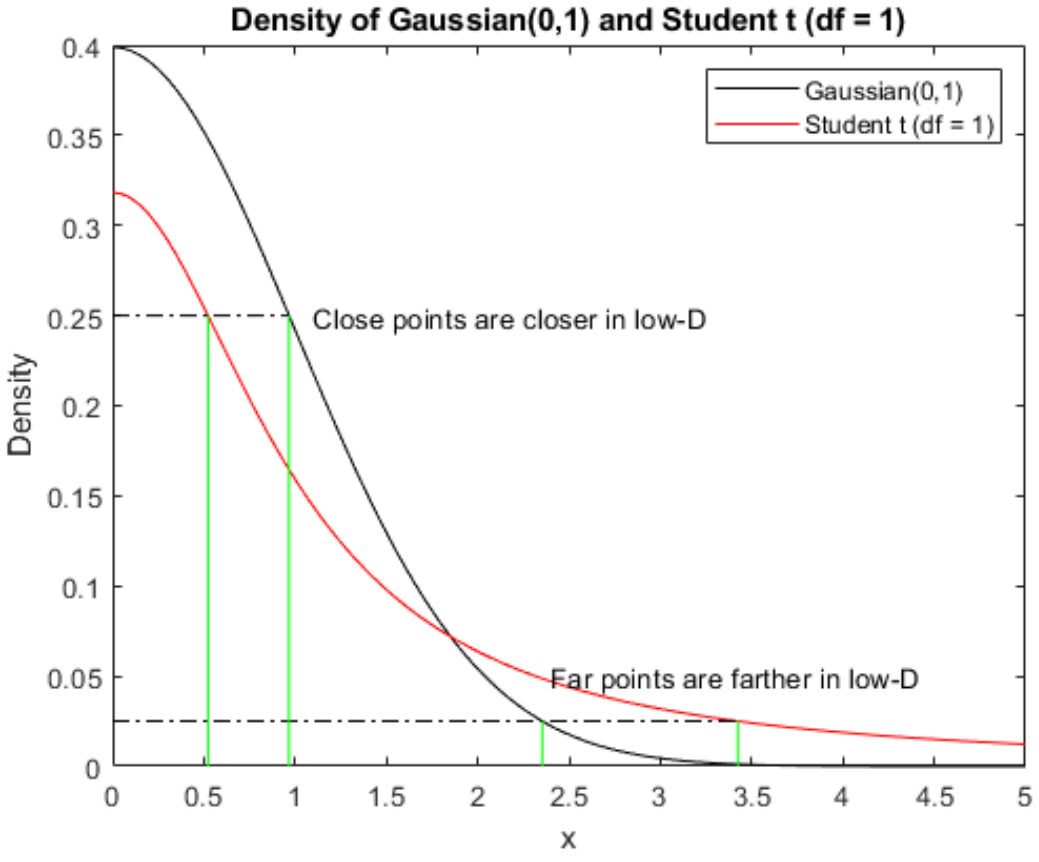
$$q_{ij} = \frac{exp(-||y_i - y_j||^2)}{\sum_{k \neq l} exp(-||y_i - y_j||^2)}$$

In symmetric SNE, it only solve the problem that **nearby points in high-D really want to be nearby in low-D.** But there woulb be another problem that **those faraway points might have change to push them, which would result in the crowded problem.** All the data points in low D would become very crowded.

(Reference from the textbook.)
Crowding problem: when the output dimensionality is smaller than the effective dimensionality of data on the input, the neighborhoods are mismatched:

- in a high-D space, points can have many close-by neighbors in different directions. In a 2D space, you essentially have to arrange close-by neighbors in a circle around the central point, which constrains relationships among neighbors.
- in a high-D space you can have many points that are equidistant from one another; in 2D, at most 3 points are equidistant from each other ‣ volume of a sphere scales as rd in d dimensions
    - on a 2D display, there is much less area available at radius r than the corresponding volume in the original space

t-SNE could alleviate crowding problem because **student t distribution would have longer-tail**. Such taht the data should be further away in low-D in order to achieve low probability.



```
def sne(X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0):
    """
        Runs symmetric-SNE on the dataset in the NxD array X to reduce its
        dimensionality to no_dims dimensions. The syntaxis of the function i
        `Y = tsne.tsne(X, no_dims, perplexity), where X is an NxD NumPy arra
    """

    # Check inputs
    if isinstance(no_dims, float):
        print("Error: array X should have type float.")
        return -1
    if round(no_dims) != no_dims:
        print("Error: number of dimensions should be an integer.")
        return -1

    # Initialize variables
    X = pca(X, initial_dims).real
    (n, d) = X.shape
    max_iter = 1000
    initial_momentum = 0.5
    final_momentum = 0.8
    eta = 500
    min_gain = 0.01
    Y = np.random.randn(n, no_dims)
    dY = np.zeros((n, no_dims))
    iY = np.zeros((n, no_dims))
    gains = np.ones((n, no_dims))

    # Compute P-values
    P = x2p(X, 1e-5, perplexity)
    P = P + np.transpose(P)
    P = P / np.sum(P)
    P = P * 2.  # early exaggeration - change!
    P = np.maximum(P, 1e-12)

    # Run iterations
    for iter in range(max_iter):

        # Compute pairwise affinities
        sum_Y = np.sum(np.square(Y), 1)
        num = -2. * np.dot(Y, Y.T)
        num = -(np.add(np.add(num, sum_Y).T, sum_Y)) # change it!!
        num = np.exp(num) # add this !!
        num[range(n), range(n)] = 0.
        Q = num / np.sum(num)
        Q = np.maximum(Q, 1e-12)

        # Compute gradient
        PQ = P - Q
        for i in range(n):
            dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] -

```

```
        # Perform the update
        if iter < 20:
            momentum = initial_momentum
        else:
            momentum = final_momentum
        gains = (gains + 0.2) * ((dY > 0.) != (iY > 0.)) + \
                (gains * 0.8) * ((dY > 0.) == (iY > 0.))
        gains[gains < min_gain] = min_gain
        iY = momentum * iY - eta * (gains * dY)
        Y = Y + iY
        Y = Y - np.tile(np.mean(Y, 0), (n, 1))

        # 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))

        # Stop lying about P-values
        if iter == 100:
            P = P / 4.

        if iter % 50 == 0:
            title = "symmetric SNE Embedding " + \
                    "perplexity=" + str(perplexity) \
                     + " iter=" + str(iter)
            visualize(Y, labels, title, "2/SNE2", "SNE_" + str(iter) +".png"

    # Return solution
    return Y, P, Q
```

In code, we need to change the calculation of gradient.
The reason is that because tsne use the Student t distribution in low D , while symmetric sne use the gaussian distribution in low D. Hence, the gradient would change too.

Gradient in t-SNE:

$\frac{\delta C}{\delta y_i} = 4\sum_j(p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1}$

The correspoding code:

```
P = P * 4.  # early exaggeration
```

```
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))
num[range(n), range(n)] = 0.
Q = num / np.sum(num)
Q = np.maximum(Q, 1e-12)
```

Gradient in symmetric SNE:
$\frac{\delta C}{\delta y_i} = 2\sum_j (p_{ij} - q_{ij})(y_i - y_j)$
The main difference from tsne is that:

```
33  P = P * 2.     # early exaggeration - change!
```

```
43  num = -(np.add(np.add(num, sum_Y).T, sum_Y)) # change it!!
44  num = np.exp(num) # add this !!
```

Part2: Visualize the embedding of both t-SNE and symmetric SNE.

Details of the visualization:

- Project all data onto 2D space and mark the data points into different colors respectively. The color of the data points depends on the label.
- Use GIF images to show the optimize procedure.

visualize the data onto 2D space.

```
1  def visualize(Y, labels, title, folder, filename):
2      plt.clf()
3      # visualize
4      scatter = plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
5      plt.legend(*scatter.legend_elements(), \
6                           loc="lower left", title="Classes")
7      plt.title(title)
8      plt.savefig(folder + "/" + filename)
```

In visualization, the package matplotlib.pyplot would be use.
Y: data onto 2D space.
plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
Here, Parameter 20 means the size. Parameter labels means to specify the color of points due to class of labels.

Make GIF

In util.py (http://util.py)

```
import imageio
import re
import os

# for sorting the sequence of file like 1, 2, 3.... 10, 11
def sorted_alphanumeric(data):
    convert = lambda text: int(text) if text.isdigit() else text.lower()
    alphanum_key = lambda key: [ convert(c) for c in re.split('([0-9]+)', ke
    return sorted(data, key=alphanum_key)

def exportGif(filePath, imgName):
    images = []
    for filename in sorted_alphanumeric(os.listdir(filePath)):
        if imgName in filename:
            images.append(imageio.imread( filePath +"/" +filename))

    imageio.mimsave(filePath +"/" + imgName + ".gif", images, duration= 0.5)
```

First we need to **list all the file** in the filePath and sort it by using the function **sorted_alphaaumeric**.

The reason why we need this is because the default sorting of our operating system would be like image1, image10, image11, image2, image20, image3, image30. But that's not what we want when exporting the gif.

So after the function sorted_alphaaumeric, it would return a sorted file as what we want, then we check whether the imageName is in the filename such as "image2" in "image2_1.jpg". If true, we append to the images list.

Then again, we check whether the directory exists or not. If not, create ones and save the gif.

Part3: Visualize the distribution of pairwise similarities

Similarity in both highdimensional space and low-dimensional space, based on both t-SNE and symmetric SNE.

```
def plotSimilarity(P, Q, labels, subtitle, filepath, filename):
    # for ploting
    idx = np.argsort(labels)
    sortedP = -np.log(P[:, idx][idx])
    sortedQ = -np.log(Q[:, idx][idx])

    fig, axes = plt.subplots(1, 2, figsize=(8, 4), squeeze=False)
    fig.suptitle(subtitle, fontsize=16)
    fig.tight_layout()

    im1 = axes[0][0].imshow(sortedP, cmap="RdBu_r")
    axes[0][0].set_title("In High dimension")


    im2 = axes[0][1].imshow(sortedQ, cmap="RdBu_r")
    axes[0][1].set_title("In Low dimension")

    fig.colorbar(im2, ax = axes.ravel().tolist())
    plt.show()
    plt.savefig(filepath + "/" + filename)
```

In visualization of pairwise similarities, we would use the heatmap to see the relationship.
Here there are something we need to note.

1. got the index of sorted labels (2, 4, 6, 8 ...) -> (0, 0, 0, ... 9, 9, 9) which means the same cluster are arranged together.
2. P and Q is rearrange.
3. Most important of all, take **-log** transformation because first we calculating the P and Q by Gaussian or Student t distribution. The effect of that is like exponential them. **Instead of take -log**, the value of P and Q is too small to see their relationship and hard to transform except for taking log...

Part4: play with different perplexity values. Observe the change in visualization and explain it in the report.

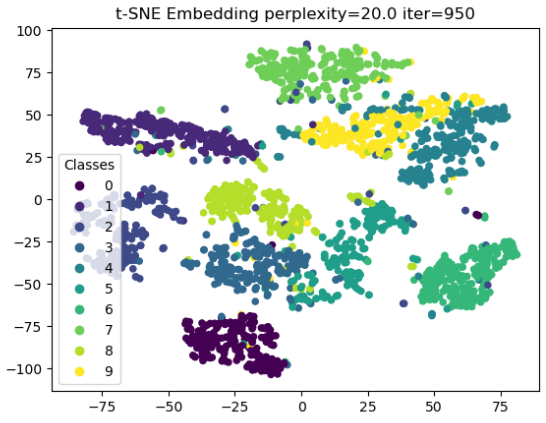
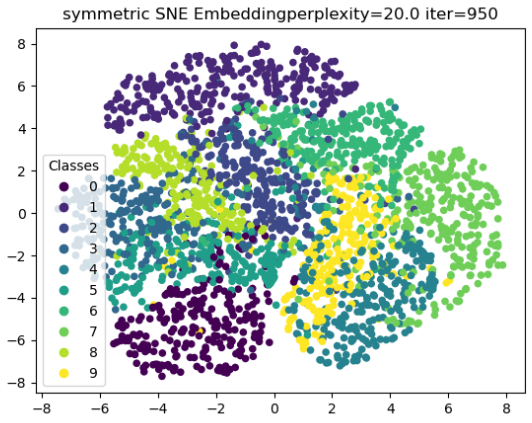
```
Perplexity = [5, 10, 50, 100]

for p in Perplexity:
    filePath = "2/tSNE_" + "perplexity_" + str(p)
    if not os.path.exists(filePath):
        os.makedirs(filePath)

    # t-sne
    Y, P, Q = tsne(X, labels, 2, 50, p, True, filePath)
    ....
```

Define a Perplexity array and iterate it to see those result with different perplexity.

experiments settings and results & discussion

Visualize the embedding of both t-SNE and symmetric SNE.

setting : perplexity=20

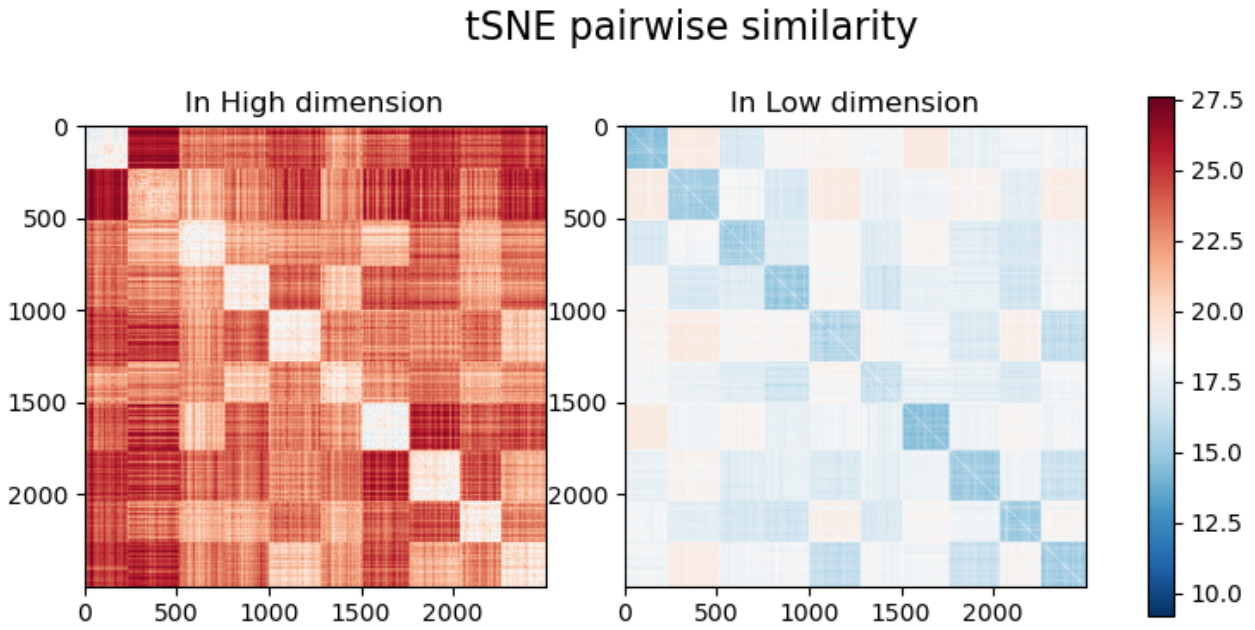
tSNE	symmetric SNE
	

We could see the **crowding problem of symmetric SNE.** There isn't clear boundary bewteen different cluster. All the data points are mixed together. Also the range of x and y axis in symmetric SNE is (-8, +8). The range of **tSNE's** is (-75, 75), which is **much larger than symmetric SNE.** Embedding in tSNE could span more widely.

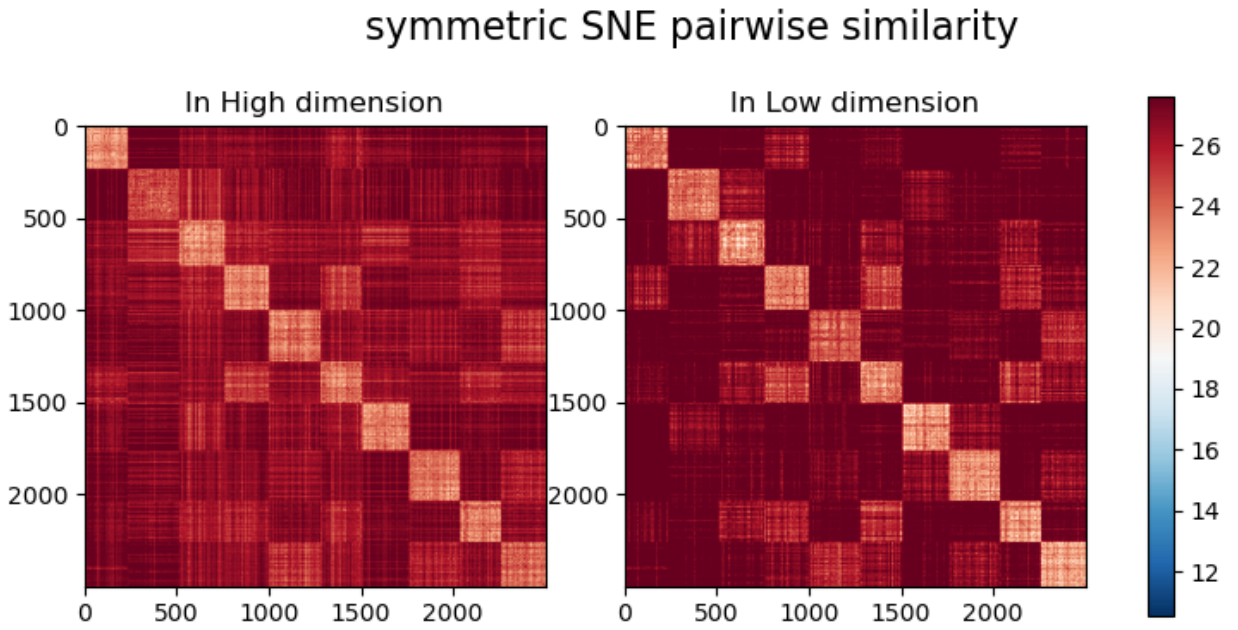
Visualize the distribution of pairwise similarities

setting : perplexity = 20

tSNE



Symmetric SNE

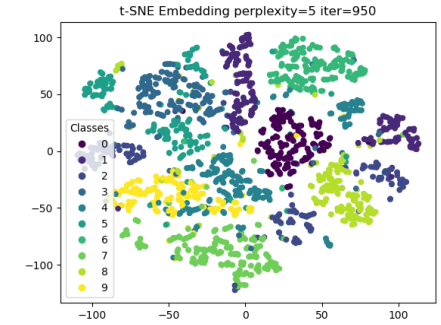
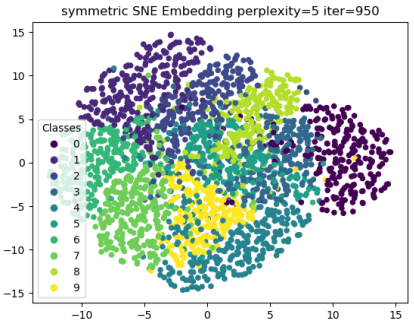
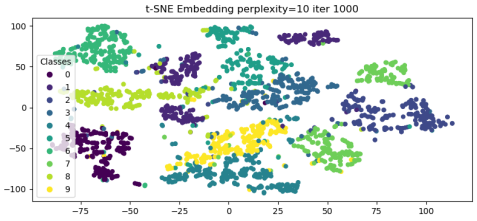
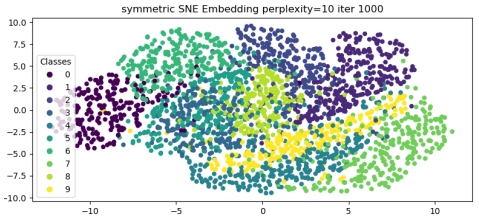
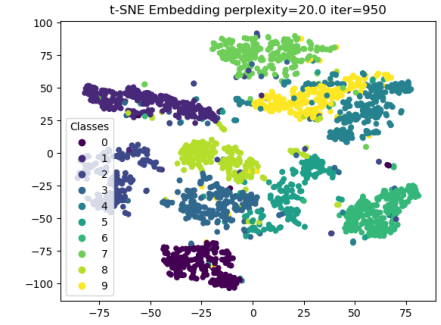
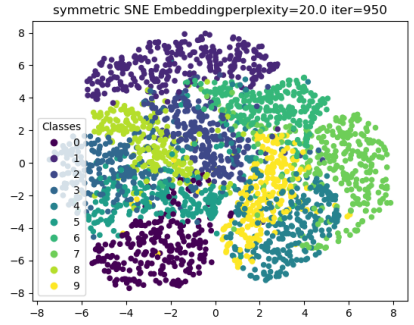
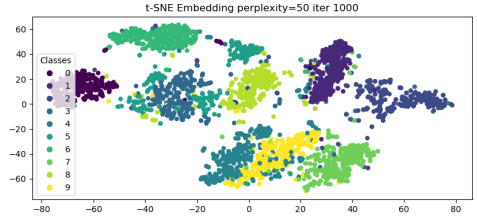
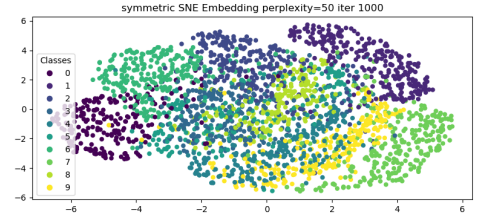
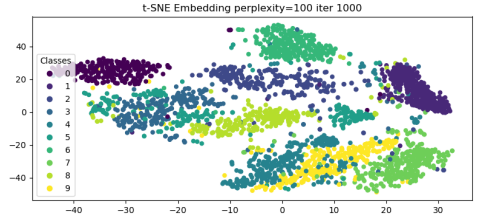
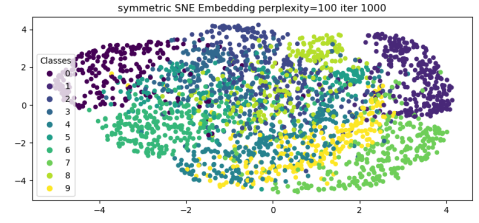


By comparing the figure in high dimension and low dimension, we could see **similarity in tSNE is more obvious.** Also noted that those blue cell means that there are less relative. We could see that the **red cell in high dimension in tSNE is correspond to the transparent one in low dimension.**
What's more, *for those less similar*, like **transparent cell in high dimension is also correspond to the blue cell in low dimension.**

As for the **symmetric SNE**, the result figure only indicate those less relative cell (the diagonal) and does not give other information. Because all **the similarity in low dimension doesn't make too much of a difference except for the diagonal.**

with different Perplexity

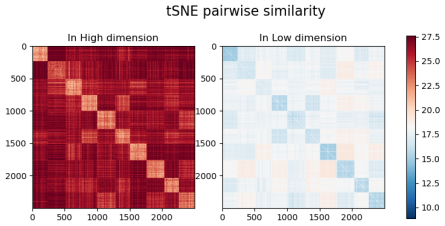
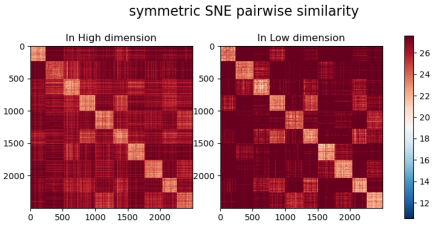
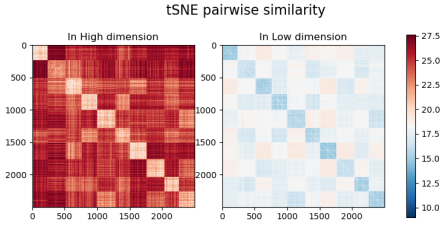
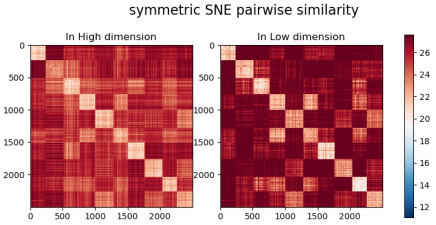
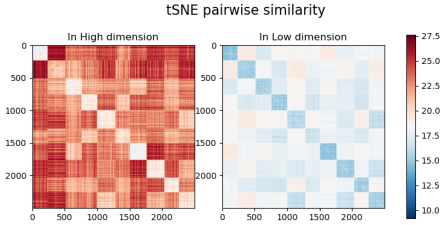
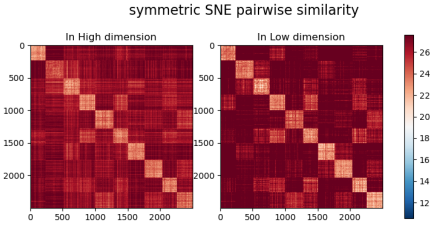
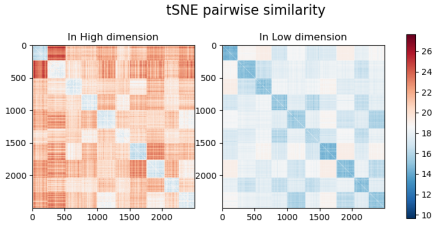
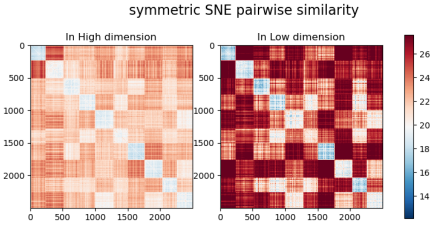
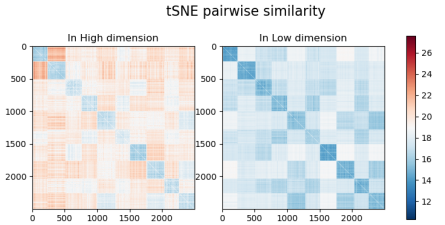
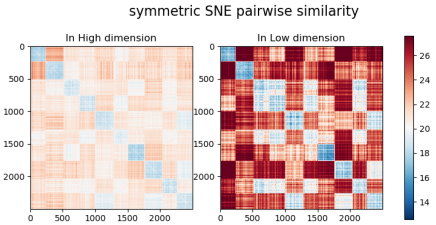
setting : fixed the initial Y

Perplexity	tSNE	symmetric SNE
perplexity=5		
perplexity=10		
perplexity=20		
perplexity=50		
perplexity=100		

With *smaller perplexity*, the boundary **in both tSNE and symmetric SNE** between different clusters is **more specific and obvious.** On the other side, with *larger perplexity*, the boundary between different clusters is **not clear and vague.** Also with

larger perplexity like 50 and 100, there are **some clusters glued to each other.** Even with different perplexity, symmetric SNE still have the crowding problem.

setting : fixed the initial Y

Perplexity	tSNE	symmetric SNE
perplexity=5		
perplexity=10		
perplexity=20		
perplexity=50		
perplexity=100		

Figures in the **left** hand sidie is **High Dimension**.
Figures in the **right** hand sidie is **Low Dimension**.

With increase in perplexity, we could see that the similarity in both high D (the left side) is more and more vague and small. That is because the result of not clearly figuring out each cluster.

In the paper, the author mentions that usually the perplexity would be aroung 5 to 50. In some situation, the perplexity would be above 100. **Generally speaking, large dataset would need large perplexity.**
The meaning of perplexity would be a measure of the effective number of neighbors, defined as the two to the power of Shannon entropy.
Larger perplexity means that the number of near neighbor would be more, which also means that local area would be less sensitive.
To recap:

- low perplexity: only little neighbor would have impact. Also it might divide the same cluster to different cluster.
- high perplexity: global structure would be more specific. But it might be hard to distinguish cluster and cluster.

So the author choose 20 to be the value of perplexity. I think 20 is the most suitable value with comparison to others.

Overall observations and discussion

In this assignment, we implemented the dimension reduction method like PCA, LDA, Kernel PCA and Kernel LDA to do image reconstruction and face recognition.

The visualization of fisherface look weird starting from face 14, while all the visualization of eigenface represent well. But the performance of LDA is much better than PCA.

As for the Kernel PCA and LDA, I am wondering that if training data minus average face before mapping them to the feature space. Does that really make the training data centered? I had tried the centeralized way in feature space both in train and test data but that result turns out that the performance is much lower and weird. Maybe there are related to the setting of hyperparamter. But in my assumption, the kernel version should outperform the original one. Because by mapping them to the feauture space would provide a non-linear clustering way.

For face recognition, the knn is implemented. The lower k we set for low dimensional space, the lower accuarcy is.

tSNE could truly solve the crowding problem of symmetric SNE. With different perplexity we could see how each cluster changes. But there’s one thing I am wondering. That is the similarity with respect to different perplexity. In that figures, should I conclude that higher perplexity would give to lower pairwise similarity? There are still lots of thing I could do to figure out those problem.

File

PCA LDA -> KernelEigenfaces.py (http://KernelEigenfaces.py)
Images of PCA and LDA would be put to folder - PCA_LDA.

tSNE and symmetric SNE -> tsne.py (http://tsne.py)
Images and Gifs of tSNE and symmetric SNE would be put to folder - tSNE_SNE.

Reference

t-SNE (https://lvdmaaten.github.io/tsne/)

人脸识别经典算法三：Fisherface（LDA） (https://blog.csdn.net/smartempire/article/details/23377385)

LDA原理与fisherface实现 (https://blog.csdn.net/chenaiyanmie/article/details/8001095)

比較LDA演算法原理及matlab實現 (https://www.itread01.com/content/1548303672.html)

資料降維與視覺化：t-SNE 理論與應用 (https://mropengate.blogspot.com/2019/06/t-sne.html)