a. code with detailed explanations (40%)

> **Kernel Eigenfaces**
>
> > Part1, part2, and part3 is controlled by the mode variable in main body of the code.
> >
> > The readfiles function reads Yale_Face_Database's data and returns (images, labels) pair, which is captured by (trainX,trainY) and (testX, testY) pair in main.

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
import os
import re

def readfiles(dirpath,shape):
 images = []
 labels = []
 #read picture with format of PGM
 for pgm in os.listdir(dirpath): #for each pgm picture
     #deal with individual pgm picture
     filepath = f'{dirpath}/{pgm}'
     image = Image.open(filepath) #use PIL.Image module to read the pgm picture
     image = image.resize(shape, Image.ANTIALIAS) #use PIL.Image module's
filter to resize the pgm picture in order to prevent excessive calculations in
LDA
     image = np.array(image) #turn the pgm picture to numerical array
     #plt.imshow(image, cmap='gray') #take a look how the picture looks like
now
     #plt.show() #you can see the picture even if in debug mode
     image = image.flatten() #flatten the image into 1-D array
     label = int(re.search(r'subject([0-9]+)', pgm).group(1)) #use regular
expression to find out the number cancated after the string "subject"

     #concate every individual pictures
     images.append(image)
     labels.append(label)

 #turn list to array
 images = np.asarray(images,dtype=np.float64) #important!!!!default type=unit8
will lead to kernel computation's overflow!!!!!
 labels = np.asarray(labels)
 return images,labels

if __name__ == '__main__':
 shape = (65,77)#(195,231) #(width,height)
 #readfiles
 trainX, trainY = readfiles("./Yale_Face_Database/Training",shape)
```

```
 testX, testY = readfiles("./Yale_Face_Database/Testing",shape)

 MODE = [1,2,3]
 for mode in MODE:
     if mode == 1:
         print("==================")
         print("Part1:Use PCA and LDA to show the first 25 eigenfaces and
fisherfaces, and randomly pick 10 images to show their reconstruction")
         #randomly pick 10 images
         randIdx = np.random.choice(len(trainX), 10)
         samplesX = trainX[randIdx]

         #PCA
         pcaW, meanX = PCA(trainX,25) #find the projection matrix that can
project the original(higher) data space to first 25(lower) eigenfaces
         showResult(shape, samplesX, pcaW, meanX) #show the eigenfaces and
reconstruction

         #LDA
         ldaW = LDA(trainX,trainY,25) #find the projection matrix that can
project the original(higher) data space to first 25(lower) fisherfaces
         showResult(shape, samplesX, ldaW, None) #show the eigenfaces and
reconstruction

     if mode == 2: #reuse the result of part1, so here is not elif
         print("\n==================")
         print("Part2: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
         ptrainX = (trainX-meanX) @ pcaW #project the training data using PCA
that present the 25 pricipal components(from higher dimensional spce into lower
dimensional space) of that data
         ptestX = (testX-meanX) @ pcaW #project the testing data using PCA that
present the 25 pricipal components(from higher dimensional spce into lower
dimensional space) of that data
         performance = KNN(ptrainX,trainY,ptestX,testY,5) #k nearest neighbor,
assume k=5
         print(f"KNN's performace under PCA = {performance:>.3f}") #show
results

         #LDA
         ptrainX = trainX @ ldaW
         ptestX = testX @ ldaW
         performance = KNN(ptrainX,trainY,ptestX,testY,5)
         print(f"KNN's performace under LDA = {performance:>.3f}")

     if mode == 3:
         print("\n==================")
```

```
        print("Part3:Use kernel PCA and kernel LDA to do face recognition, and compute the performance. (You can choose whatever kernel you want, but you should try different kernels in your implementation.) Then compare the difference between simple LDA/PCA and kernel LDA/PCA, and the difference between different kernels.")
        kernels = [linearKernel,rbfKernel] #function
        for kernel in kernels: #for all kernel functions
            #kernel PCA
            kpcaW = kernelPCA(trainX,25,kernel)
            ptrainX = kernel(trainX,trainX).T @ kpcaW #using kernel projection matrix W, project training data from higher dimensional feature space into lower dimensional feature space, refer to PPT p.128
            ptestX = kernel(trainX,testX).T @ kpcaW #using kernel projection matrix W, project testing data from higher dimensional feature space into lower dimensional feature space, refer to PPT p.128
            performance = KNN(ptrainX,trainY,ptestX,testY,5)
            print(f"KNN's performace under kernel PCA using {kernel.__name__} = {performance:>.3f}")

            #kernel LDA
            kldaW = kernelLDA(trainX,trainY,25,kernel)
            ptrainX = kernel(trainX,trainX).T @ kldaW
            ptestX = kernel(trainX,testX).T @ kldaW
            performance = KNN(ptrainX,trainY,ptestX,testY,5)
            print(f"KNN's performace under kernel LDA using {kernel.__name__} = {performance:>.3f}")
```

Part1: Use PCA and LDA to show the first 25 eigenfaces and fisherfaces, and randomly pick 10 images to show their reconstruction (10%)

> Both PCA and LDA are the method about dimensionality reduction, while the former relates to unsupervised manner, and the latter relates to supervised manner.
>
> The showResult function shows the eigenspace(fisherspace) and the reconstructed result, and the procedure will be discuss in third and final step in PCA.

```
def showResult(shape, samplesX, W, meanX):
    #show eigenfaces or fisherfaces
    dirpath = "./result"
    os.makedirs(dirpath, exist_ok=True)
    if meanX is None: #LDA
        meanX = np.zeros(samplesX.shape[1])
        alogType = 'LDA'
        faceType = 'Fisherfaces'
    else:  #PCA
        alogType = 'PCA'
        faceType = 'Eigenfaces'
```

```
    plt.suptitle(f"{alogType}: {faceType}")
    for i in range(W.shape[1]): #W.shape[1] == 25
        plt.subplot(5,5,i+1)
        plt.imshow(W[:, i].reshape(shape[::-1]), cmap='gray')
        plt.axis('off')
    plt.savefig(f'{dirpath}/part1_{alogType}_{faceType}.png')
    plt.show()

    #show original (defalut) 10 samples
    plt.suptitle(f"{alogType}: Original samples")
    for i in range(len(samplesX)): #len(samplesX) == 10
        plt.subplot(2,5,i+1)
        plt.imshow(samplesX[i].reshape(shape[::-1]),cmap='gray')
        plt.axis('off')
    plt.savefig(f'{dirpath}/part1_{alogType}_original_samples.png')
    plt.show()

    #show reconstructed (defalut) 10 samples, refer to PPT p.121
    projectX = (samplesX-meanX) @ W #use projection matrix W to project data
in higher dimensional space into lower dimensional space
    reconstructX = projectX @ W.T + meanX #W can project data from high-D
into low-D, so in oppsite, W.T can project data from low-D onto high-D
    plt.suptitle(f"{alogType}: Reconstructed samples")
    for i in range(len(reconstructX)): #len(reconstructX) == 10
        plt.subplot(2,5,i+1)
        plt.imshow(reconstructX[i].reshape(shape[::-1]),cmap='gray')
        plt.axis('off')
    plt.savefig(f'{dirpath}/part1_{alogType}_reconstructed_samples.png')
    plt.show()
```

The **PCA** function returns the projection matrix, and the returned matrix will be use in showResult function and part2.

```
def PCA(X,k):
    #build the covariance matrix S, refer to PPT p.119
    meanX = np.mean(X, axis=0)
    covariaceX = (X-meanX) @ (X-meanX).T

    #find the orthogonal projection matrix W containing k principal
components, refer to PPT p.120
    eigenValues, eigenVectors = np.linalg.eigh(covariaceX)
    if np.all(eigenValues.imag < 1e-3): #if the imaginary part is close to 0
        eigenValues = eigenValues.real
        eigenVectors = eigenVectors.real
    kLargestIdx = np.argsort(eigenValues)[::-1][:k] #[::-1] means to revert
an array, hence this line code means to find the k "largest" eigenvalues
```

```
    W = (X-meanX).T @ eigenVectors[:, kLargestIdx] #project X onto the k
principal component's axes
    W = W / np.linalg.norm(W,axis=0) #normalize, because ||w||=1

    return W, meanX
```

Let's see what PCA does.

First, build the covaraicne matrix S which refers to the formula in PPT p.119.

$$S = [\frac{1}{N}\sum_{x}(x-\bar{x})(x-\bar{x})^T]$$

Second, construct orthogonal projection matrix W which is composed of k first largest normalized eigenvectors ($||w^2|| = 1$) of covariance matrix of x, refer to PPT p.120. The projection matrix W can project the origianl high dimensional data into lower k dimensional space. Here k is defined as 25 for the problem requirement. That is, there are 25 columns in W, and the each column of W represents one dimension of Eigenspace.

Third, project data into lower dimensional space by the formula in PPT p.121.

$$projectX = Wx$$

Finally, reconstruct data from low dimensional space onto high dimensional space by the formula in PPT p.121.

$$reconstructX = xWW^T$$

The **LDA** function returns the projection matrix, and the returned matrix will be use in showResult function and part2.

```
def LDA(X,Y,k):
    #build within-class scatter Sw and between-class scatter Sb ,refer to PPT
p.179
    meanX = np.mean(X, axis=0)
    classes = np.unique(Y)
    Sw = np.zeros((X.shape[1], X.shape[1]))
    Sb = np.zeros((X.shape[1], X.shape[1]))
    for c in classes:
        meanXj = np.mean(X[Y==c],axis=0)
        #within-class scatter Sw
        Sj = (X[Y==c] - meanXj).T @ (X[Y==c] - meanXj)
        Sw += Sj

        #between-class scatter Sb
        Sbj = np.sum(Y==c) * ((meanXj-meanX).T @ (meanXj-meanX))
        Sb += Sbj

    #build the projection matrix W, refer to PPT p.181
```

```
    eigenValues, eigenVectors = np.linalg.eig(np.linalg.pinv(Sw) @ Sb)
    if np.all(eigenValues.imag < 1e-3):
        eigenValues = eigenValues.real
        eigenVectors = eigenVectors.real
    kLargestIdx = np.argsort(eigenValues)[::-1][:k]
    W = eigenVectors[:, kLargestIdx]
    return W
```

Let's see what LDA does.

First, build within-class scatter S_W and between-class scatter S_B which refers to PPT p.179.

$$S_W = \sum_{j=1}^{k} S_j, \text{ where } S_j = \sum_{i \in C_j} (x_i - m_j)(x_i - m_j)^T, \text{ and } m_j = \frac{1}{n_j} \sum_{i \in C_j} x_i$$

$$S_B = \sum_{j=1}^{k} S_{B_j}, \text{ where } S_B = \sum_{j=1}^{k} n_j (m_j - m)(m_j - m)^T, \text{ and } m = \frac{1}{n} \sum x$$

Second, construct the projection matrix which is composed of k largest eigenvectors of $\frac{S_B}{S_W}$, referring to PPT p.181. Again, the projection matrix W can project the origianl high dimensional data into lower k=25 dimensional space which is also known as Fisherspace, and each dimension of the Fisherspace is the column of W.

The third step and final step are same as third step and final step in PCA.

Part2: 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. (5%)

The KNN function does the k nearest neighbor algorithm, and returns the performance.

The KNN algorithm labels test data as the k shortest train data's majority label.

```
def KNN(trainX,trainY,testX,acTestY,k): # k nearest neighbor
    distances = cdist(testX,trainX,'sqeuclidean') #compute each train and
test's pairwise distance
    predTestY = np.zeros(len(acTestY))
    for i in range(len(testX)): #for each testing data, predict their labels
        kNearestIdx = np.argsort(distances[i])[:k] # find k nearest training
data for i-th testing data => compute the k "shortest" distance between i-th
testing data and all the training data
        candidates = trainY[kNearestIdx] #possible k predicted labels
        candidates,counts = np.unique(candidates,return_counts=True)
#possible unique predicted labels with their counts
        predTestY[i] = candidates[np.argmax(counts)] #major vote
    performance = np.sum(predTestY == acTestY) / len(acTestY)
    return performance
```

> The input parameters' dimension will be reduced to 25 by the projection matrix produced by PCA or LDA in main before putting into KNN function.

Part3: Use kernel PCA and kernel LDA to do face recognition, and compute the performance. (You can choose whatever kernel you want, but you should try different kernels in your implementation.) Then compare the difference between simple LDA/PCA and kernel LDA/PCA, and the difference between different kernels. (10%)

> The kernel functions I used includes linear kernel and RBF kernel.

$$\text{linear kernel: } k(x_1, x_2) = x_1 \cdot x_2^T$$
$$\text{RBF kernel: } k(x_1, x_2) = \exp(-\gamma||x_1 - x_2||^2)$$

```
def linearKernel(x1,x2):
    return x1 @ x2.T

def rbfKernel(x1, x2, gamma=1e-10): #gamma=1's performance is worse
    return np.exp(-gamma * cdist(x1, x2, 'sqeuclidean'))
```

> The **kernelPCA** function returns the projection matrix W in feature space.

```
def kernelPCA(X,k,kernel): #parameter kerenl is a function
    #implement kernel PCA, refer to PPT p.128
    K = kernel(X,X)
    oneN = np.ones((len(X), len(X))) / len(X)
    Kcov = K - oneN@K - K@oneN + oneN@K@oneN #covariance matrix in the
feature space
    eigenValues, eigenVectors = np.linalg.eigh(Kcov)
    if np.all(eigenValues.imag < 1e-3):
        eigenValues = eigenValues.real
        eigenVectors = eigenVectors.real
    kLargestIdx = np.argsort(eigenValues)[::-1][:k] #find the k "largest"
eigenvalues
    W = eigenVectors[:, kLargestIdx]
    W = W / np.linalg.norm(W,axis=0) #normalize
    return W
```

> After computting the projection matrix W, I use it to project original data from higher dimensional feature space into lower dimensional feature space and do the KNN algorithm to do the classificaiton. The code is written in main.

> Let's see what kernel PCA does.

> First, construct covariance matrix K^c in the feature space referring to PPT p.128

$$K^c = K - 1_N K - K 1_N + 1_N K 1_N, \text{ where } 1_N \text{ is NxN matrix with every element } 1/\text{N}$$

Second, find the k corresponding eigenvectors with k largest eigenvalues to construct projection matrix W.

Finally, project the data in feature space into k dimensional space using the formula in PPT p.128.

$$W\Phi(x_{new}) = \sum_i \alpha_i K(x_i, x_{new}),$$

where α_i is the eigenvector with i-th largest eigenvalue of covariance matrix K^c in the feature space,
i.e. i-th column in projection matrix W that project the data from original higher dimensional feature space into lower k dimensional feature space.
$K(x_i, x_{new})$ is the similarity of data x_i and data x_{new} in feature space.

The **kernelLDA** function returns the projection matrix W in feature space.

```
def kernelLDA(X,Y,k,kernel):
    #build the projection matrix W, refer to formula 15 from [Kernel
Eogenfaces vs. kernel fisherfaces: face recognition usgin kernel methods,
https://www.csie.ntu.edu.tw/%7Emhyang/papers/fg02.pdf]
    #refer to wiki Multi-class KFD,
https://en.wikipedia.org/wiki/Kernel_Fisher_discriminant_analysis#Multi-
class_KFD
    K = kernel(X,X)

    meanK = np.mean(K, axis=0)
    classes = np.unique(Y)
    N = np.zeros((len(X), len(X)))
    M = np.zeros((len(X), len(X)))
    for c in classes:

        meanKj = np.mean(K[Y==c], axis=0)
        lj = np.sum(Y==c)
        onelj = np.ones((lj, lj)) / lj
        #within-class scatter N in the feature space
        N += K[Y==c].T @ (np.eye(lj)-onelj) @ K[Y==c]

        #between-class scatter M in the feature space
        M += lj * (meanKj-meanK).T @ (meanKj-meanK)

    #build the projection matrix W, same logic as PPT p.181
    eigenValues, eigenVectors = np.linalg.eig(np.linalg.pinv(N) @ M)
    if np.all(eigenValues.imag < 1e-3):
        eigenValues = eigenValues.real
        eigenVectors = eigenVectors.real
    kLargestIdx = np.argsort(eigenValues)[::-1][:k]
    W = eigenVectors[:, kLargestIdx]
    return W
```

Again, after computting the projection matrix W, I use it to project original data from higher dimensional feature space into lower dimensional feature space and do the KNN algorithm to do the classificaiton. The code is also written in main.

> Let's see what kernel LDA does, and the method refers to wiki Multi-class KFD, https://en.wikipedia.org/wiki/Kernel_Fisher_discriminant_analysis#Multi-class_KFD
>
> First, construct within-class scatter N in the feature space and between-class scatter M in the feature space.
>
> $$M = \sum_{j=1}^{c} l_j (M_j - M_*)(M_j - M_*)^T,$$
>
> where M_j's i-th element is $(M_j)_i = \frac{1}{l_j} \sum_{k=1}^{l_j} k(x_i, x_k^j)$, l_j is the number of datapoints in class j,
>
> M_*'s i-th element is $(M_*)_i = \frac{1}{l} \sum_{k=1}^{l} k(x_i, x_k)$, l is the number of datapoints in whole dataset.
>
> $$N = \sum_{j=1}^{c} K_j (I - 1_{l_j}) K_j^T$$
>
> where $K_j$is the similarity of datapoint in class j in feature space,
> 1_j is NxN matrix with every element $1/l_j$.
>
> Second, build the projection matrix W, whose logic is same as in PPT p.181 and is constructed by k largest eigenvectors of $\frac{M}{N}$.
>
> The final step of projection is same as kernel PCA.

t-SNE

> Here is the modified code of t-SNE, all the rewritten area has the #modify comment. These code inclueds both algorithm of t-SNE and symmetric SNE which is seperate by the if-else statement.

```
#
#  sne.py
#
# Implementation of t-SNE and symmetric-SNE in Python. The implementation was
tested on Python
# 2.7.10, and it requires a working installation of NumPy. The implementation
# comes with an example on the MNIST dataset. In order to plot the
# results of this example, a working installation of matplotlib is required.
#
# The example can be run by executing: `ipython sne.py`
#
#
#  Created by Laurens van der Maaten on 20-12-08.
#  Copyright (c) 2008 Tilburg University. All rights reserved.

import numpy as np
import pylab
import imageio
import io
import os
```

```
from scipy.spatial.distance import cdist

def Hbeta(D=np.array([]), beta=1.0):
 """
     Compute the perplexity and the P-row for a specific value of the
     precision of a Gaussian distribution.
 """

 # Compute P-row and corresponding perplexity
 P = np.exp(-D.copy() * beta)
 sumP = sum(P)
 H = np.log(sumP) + beta * np.sum(D * P) / sumP
 P = P / sumP
 return H, P


def x2p(X=np.array([]), tol=1e-5, perplexity=30.0):
 """
     Performs a binary search to get P-values in such a way that each
     conditional Gaussian has the same perplexity.
 """

 # Initialize some variables
 print("Computing pairwise distances...")
 (n, d) = X.shape
 sum_X = np.sum(np.square(X), 1)
 D = np.add(np.add(-2 * np.dot(X, X.T), sum_X).T, sum_X)
 P = np.zeros((n, n))
 beta = np.ones((n, 1))
 logU = np.log(perplexity)

 # Loop over all datapoints
 for i in range(n):

     # Print progress
     if i % 500 == 0:
         print("Computing P-values for point %d of %d..." % (i, n))

     # Compute the Gaussian kernel and entropy for the current precision
     betamin = -np.inf
     betamax = np.inf
     Di = D[i, np.concatenate((np.r_[0:i], np.r_[i+1:n]))]
     (H, thisP) = Hbeta(Di, beta[i])

     # Evaluate whether the perplexity is within tolerance
     Hdiff = H - logU
     tries = 0
     while np.abs(Hdiff) > tol and tries < 50:
```

```
            # If not, increase or decrease precision
            if Hdiff > 0:
                betamin = beta[i].copy()
                if betamax == np.inf or betamax == -np.inf:
                    beta[i] = beta[i] * 2.
                else:
                    beta[i] = (beta[i] + betamax) / 2.
            else:
                betamax = beta[i].copy()
                if betamin == np.inf or betamin == -np.inf:
                    beta[i] = beta[i] / 2.
                else:
                    beta[i] = (beta[i] + betamin) / 2.

            # Recompute the values
            (H, thisP) = Hbeta(Di, beta[i])
            Hdiff = H - logU
            tries += 1

        # Set the final row of P
        P[i, np.concatenate((np.r_[0:i], np.r_[i+1:n]))] = thisP

    # Return final P-matrix
    print("Mean value of sigma: %f" % np.mean(np.sqrt(1 / beta)))
    return P


def pca(X=np.array([]), no_dims=50):
    """
        Runs PCA on the NxD array X in order to reduce its dimensionality to
        no_dims dimensions.
    """

    print("Preprocessing the data using PCA...")
    (n, d) = X.shape
    X = X - np.tile(np.mean(X, 0), (n, 1))
    (l, M) = np.linalg.eig(np.dot(X.T, X))
    Y = np.dot(X, M[:, 0:no_dims])
    return Y


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

    # 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))
    Ys = [] #modify original tsne to my tsne
    Ps = [] #modify, same as above
    Qs = [] #modify, same as above
    iters = [] #modify, same as above

    # Compute P-values
    P = x2p(X, 1e-5, perplexity)
    P = P + np.transpose(P)
    P = P / np.sum(P)
    P = P * 4.                  # early exaggeration
    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)

        #modify original tsne to my tsne and ssne
        #Part1: Try to modify the code a little bit and make it back to symmetric
SNE
        if algo == 'tSNE': #refer to PPT p.172
            #num = -2. * np.dot(Y, Y.T)
            #num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
            num = 1 / ( 1 + cdist(Y,Y,'sqeuclidean') ) #cdist(Y, Y, 'sqeuclidean')
means ||Y-Y||^norm2
        elif algo == 'sSNE': #refer to PPT p.167
            num = np.exp( -1 * cdist(Y,Y,'sqeuclidean') )
        num[range(n), range(n)] = 0. #diagonal
        Q = num / np.sum(num)
```

```
        Q = np.maximum(Q, 1e-12)

        # Compute gradient
        PQ = P - Q
        for i in range(n):
            #modify original tsne to my tsne and ssne
            if algo == 'tSNE': #refer to PPT p.172
                dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T *
(Y[i, :] - Y), 0)
            elif algo == 'sSNE': #refer to PPT p.167
                dY[i, :] = np.dot( PQ[i,:] , Y[i,:]-Y ) #sum up all j(hidden in :)


        # 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))
            Ys.append(Y) #modify original tsne to my tsne ; to record the the
convergence process and do the visualization
            Ps.append(P) #modify, same as above
            Qs.append(Q) #modify, same as above
            iters.append(str(iter)) #modify, same as above

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

    # Return solution
    return Ys,Ps,Qs,iters #modify original tsne to my tsne

def visualization(algo, Ys, Ps, Qs, iters, perplexity=30.0): #modify original
tsne to my tsne
    #part 2 : Visualize the embedding of both t-SNE and symmetric SNE. Details of
the visualization
    #Project all your data onto 2D space and mark the data points into different
colors respectively. The color of the data points depends on the label.
    #Use videos or GIF images to show the optimize procedure.
```

```
 dirpath = f'./result/part2_{algo}_2Dprojection_{perplexity}perplexity'
 bufferMode = True
 if not bufferMode: os.makedirs(dirpath, exist_ok=True)
 images = []
 for i,Y in zip(iters,Ys):
     pylab.title(f"{algo} 2D projection with {perplexity} perplexity")
     pylab.scatter(Y[:, 0], Y[:, 1], s=20, c=labels) #s means the marker size,
c means the marker's color;
     if bufferMode:
         buffer = io.BytesIO()
         pylab.savefig(buffer, format='png')
         buffer.seek(0)
         pylab.show()
         image = pylab.imread(buffer, format='png')
         #buffer.truncate(0) #buffer.tell() #buffer.getvalue() #buffer.close()
         buffer.close()
     else:
         pylab.savefig(f'{dirpath}/iteration{i}.png', format='png')
         pylab.show()
         image = pylab.imread(f'{dirpath}/iteration{i}.png', format='png')
     images.append(image)
 imageio.mimsave(f'./result/part2_{algo}_2Dprojection_{perplexity}perplexity.gi
f', images) #save as gif

 #part 3 : Visualize the distribution of pairwise similarities in both high-
dimensional space and low-dimensional space, based on both t-SNE and symmetric
SNE
 Dimensionalities = ['high','low']
 Data = [Ps[-1],Qs[-1]]
 for dim,data in zip(Dimensionalities,Data):
     pylab.suptitle(f"in {dim} dimensional space with {perplexity} perplexity")
     pylab.subplot(1,2,1)
     pylab.title('the distribution of similarity') #the log distribution of
pairwise similarities in high-D
     pylab.xlabel("pairwise similarity")
     pylab.ylabel("log probability")
     pylab.hist(data.flatten(),bins=100,log=True)
     pylab.subplot(1,2,2)
     pylab.title('the similarity heat matrix') #the pairwise similarities heat
matrix in high-D
     pylab.xlabel("data points")
     pylab.ylabel("data points")
     pylab.imshow(data, cmap='binary', interpolation='nearest')

pylab.savefig(f'./result/part2_{algo}_similarity_{dim}D_{perplexity}perplexity.
png', format='png')
     pylab.show()
```

```
if __name__ == "__main__":
 print("Run Y = sne.sne(X, no_dims, perplexity) to perform t-SNE and symmetric-SNE on your dataset.")
 print("Running example on 2,500 MNIST digits...")
 X = np.loadtxt("./SNE_Database/mnist2500_X.txt") #modify original tsne to my tsne; X has 784 dimensions
 labels = np.loadtxt("./SNE_Database/mnist2500_labels.txt") #modify original tsne to my tsne; Y has 2 dimensions
 perplexities = [10.0, 20.0, 30.0, 40.0, 50.0, 100, 300, 500, 700, 900] #refer to PPT p.163
 algos = ['tSNE','sSNE'] #sSNE means symmetric SNE
 for algo in algos:
     #Part4:Try to play with different perplexity values. Observe the change in visualization and explain it in the report
     for perplexity in perplexities:
         Ys,Ps,Qs,iters = sne(algo, X, 2, 50, perplexity) #modify original tsne to my tsne
         visualization(algo, Ys,Ps,Qs,iters,perplexity) #modify original tsne to my tsne
```

Part1: Try to modify the code a little bit and make it back to symmetric SNE. You need to first understand how to implement t-SNE and find out the specific code piece to modify. You have to explain the difference between symmetric SNE and t-SNE in the report (e.g. point out the crowded problem of symmetric SNE).

> This part's code has the comment of #Part1: Try to modify the code a little bit and make it back to symmetric SNE.
>
> The main difference between t-SNE and symmetric-SNE is the similarity of data points in low dimensionality $q_{j|i}$ and gradient step.
>
> In symmetric SNE, the q_{ij} and $\frac{\partial C}{\partial y_i}$ is defined in PPT p.165 and p.166. This algorithm has the crowded problem because SNE uses KL divergence as the cost function which is asymmetric. The asymmetric property causes the loss function's value different when q is larger than p and when p is larger than q. In other words, symmetric SNE only cares that nearby data points in high dimensional space should be also nearby in low dimensional space, but the far away data points in high dimensional space are not enforced to be also far away in low dimensional space. It gives the chance that far away data points in high dimensional space may be close to each other in low dimensional space.
>
> $$q_{ij} = \frac{exp(-||y_i - y_j||^2)}{exp(-||y_l - y_k||^2)}$$
> $$\frac{\partial C}{\partial y_i} = \sum_{j \neq i}(p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j})(y_i - y_j)$$

> In t-SNE, the q_{ij} and $\frac{\partial C}{\partial y_i}$ is defined in PPT p.172. To alleviate the crowded problem in (symmetric) SNE, t-SNE uses student-t distribution in low dimensional space which gives small(large) penalty to the loss function's value when p is larger(smaller) than q. This trick somehow mitigate KL divergence's asymmetric problem.
>
> $$q_{ij} = \frac{(1+||y_i - y_j||^2)^{-1}}{\sum_{k \neq l}(1+||y_i - y_j||^2)^{-1}}$$
> $$\frac{\partial C}{\partial y_i} = 4\sum_j (p_{ij} - q_{ij})(y_i - y_j)(1+||y_i - y_j||^2)^{-1}$$

part 2 : Visualize the embedding of both t-SNE and symmetric SNE. Details of the visualization. Project all your data onto 2D space and mark the data points into different colors respectively. The color of the data points depends on the label. Use videos or GIF images to show the optimize procedure.

> This part's code is written on the upper part of visualization function. The projection results stored in the variable Y, and I record each iteration's Y into variable Ys. The first and second column of Y represents first and second dimension of 2D space respectively.

part 3 : Visualize the distribution of pairwise similarities in both high-dimensional space and low-dimensional space, based on both t-SNE and symmetric SNE

> This part's code is written on the lower part of visualization function. Here I also records every P and Q value in each iteration as variable Ps and Qs, and use the final result of P and Q as the visualized target. I use two ways to do the visuialization, one is the heat matrix of pairwise similarity, and the other is the histogram which records each similarity value to represents the similarity distribution.

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

> This part's code is with the comment #Part4: Try to play with different perplexity values. Observe the change in visualization and explain it in the report. The perplexity values I used are [10.0, 20.0, 30.0, 40.0, 50.0, 100, 300, 500, 700, 900].

b. experiments settings and results (35%) & discussion (15%)

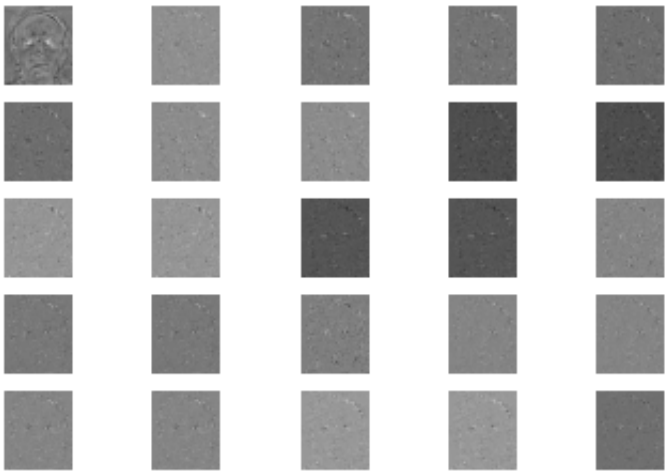
> Kernel Eigenfaces
>
> > **Part1 (5%)**

PCA: Eigenfaces

PCA: Original samples

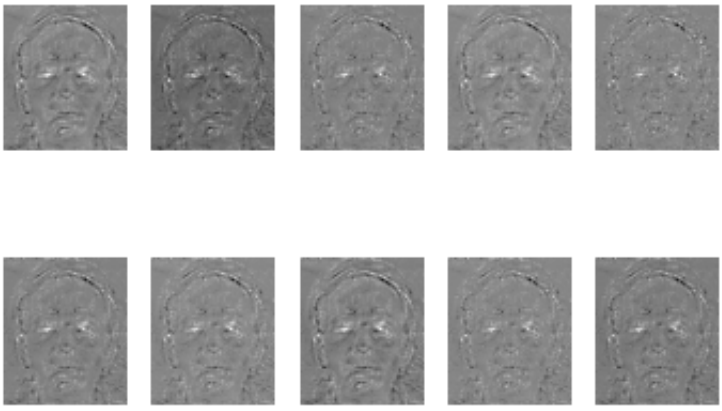
PCA: Reconstructed samples

LDA: Fisherfaces



LDA: Original samples

LDA: Reconstructed samples



Part2 (5%)

> The KNN function's default k is 5.
>
> ```
> ====================
> Part2: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.
> KNN's performace under PCA = 0.900
> KNN's performace under LDA = 0.700
> ```

Part3 (5%) & (5%)

> **[result]**
>
> I use linear kernel and RBF kernel (gamma=1e-10) here, and the KNN function's default k is also 5.
>
> ```
> ====================
> Part3:Use kernel PCA and kernel LDA to do
> face recognition, and compute the
> performance. (You can choose whatever
> kernel you want, but you should try
> different kernels in your implementation.)
> Then compare the difference between simple
> LDA/PCA and kernel LDA/PCA, and the
> difference between different kernels.
> KNN's performace under kernel PCA using
> linearKernel = 0.833
> KNN's performace under kernel LDA using
> linearKernel = 0.733
> KNN's performace under kernel PCA using
> rbfKernel = 0.800
> KNN's performace under kernel LDA using
> rbfKernel = 0.700
> ```
>
> **[discussion]**
>
> The result shows that in this example, linear kernel is better than RBF kernel and PCA is better than LDA. The information of labels doesn't give any help for classification.

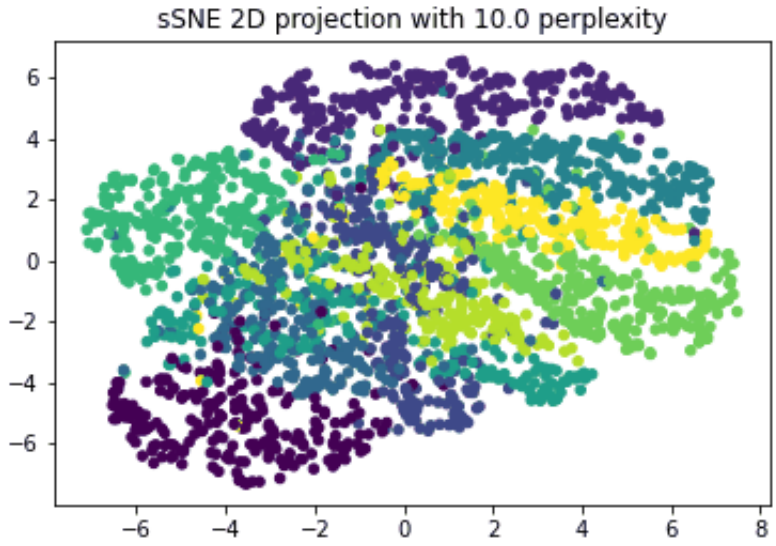
t-SNE

> **Part1 (5%) & (5%)**
>
> > **[result]**
> >
> > The results are in the result folder.
> >
> > **[discussion]**

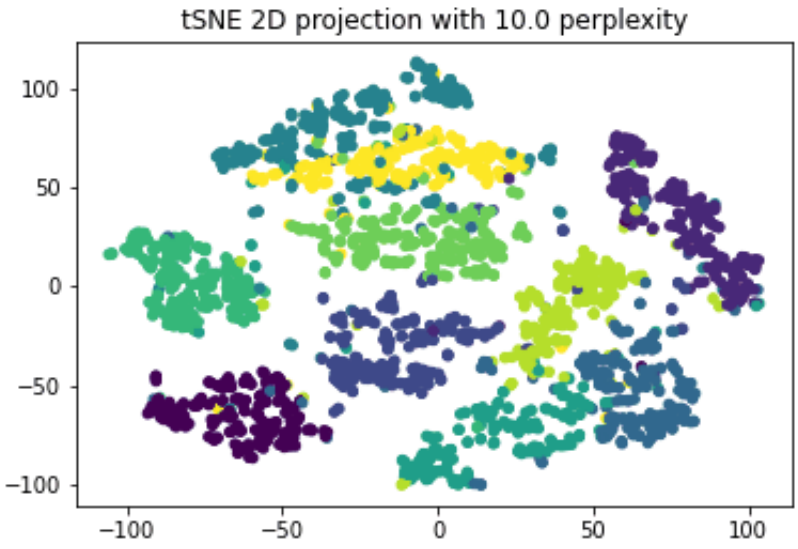
Obviously, there is a crowded problem in symmetric-SNE, and the reason has been mentioned in part a. In brief, symmetric-SNE uses asymmetric loss function, KL divergence, with no adjustment like adding different penalties. It gives two points that are far apart in higher dimensions the opportunity to become close in lower dimensions. Hence, the data in low dimensional space looks crowded.

Part2 (5%)

For symmetric SNE with 10 perplexity,

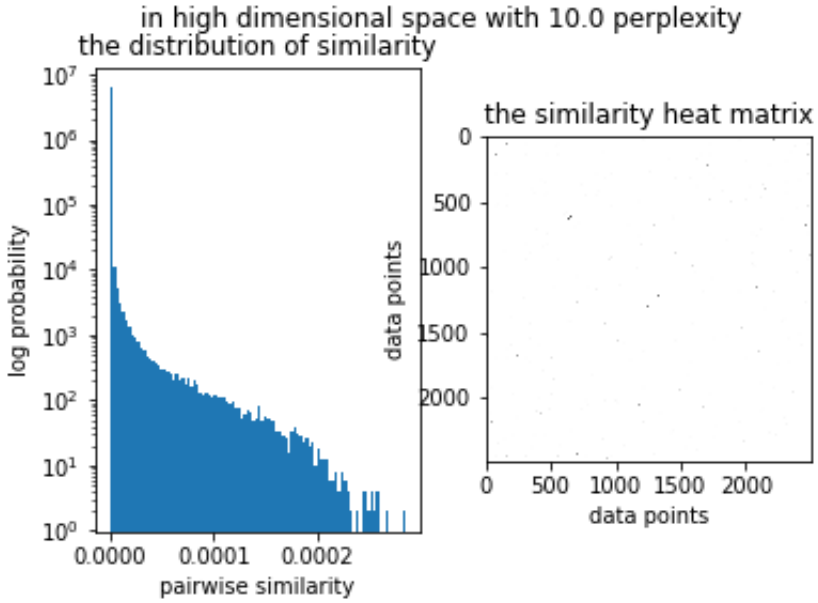


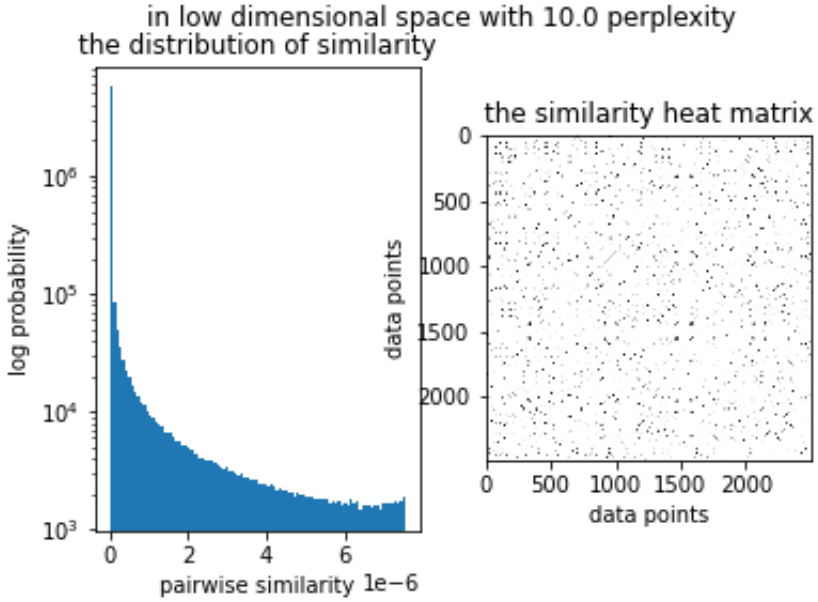
For t-SNE with 10 perplexity,



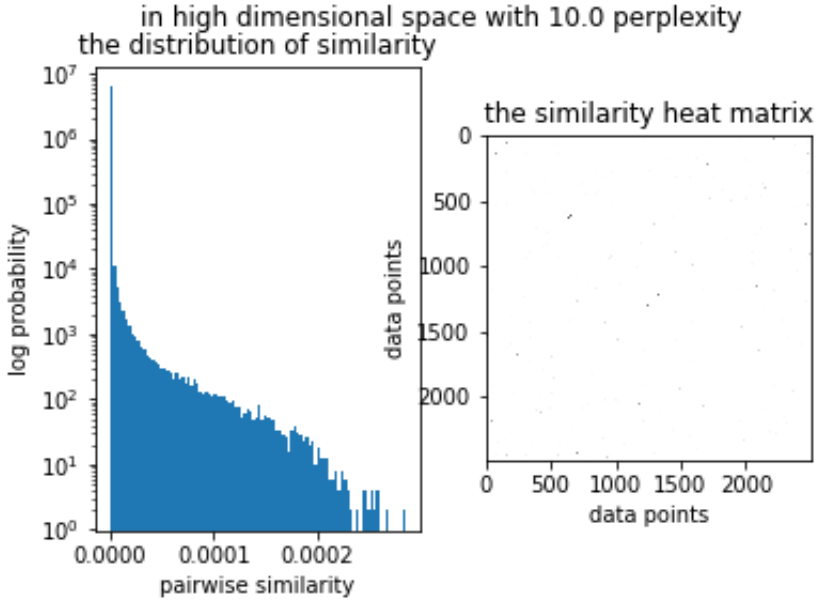
Part3 (5%)

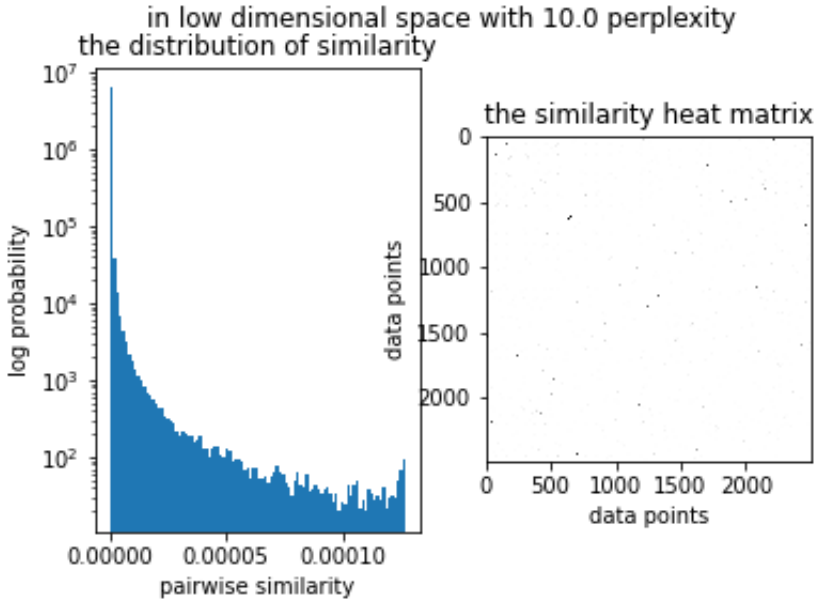
For symmetric-SNE with 10 perplexity,





For t-SNE with 10 perplexity,





Part4 (5%) & (5%)

> **[reuslt]**
>
> The other results with different perplexity are in the result folder. The embedding visualization has the name of part2_xSNE_2Dprojection_xxxperplexity, the pairwise similarity in high dimensional space has the name of part2_xSNE_similarity_highD_xxxperplexity, and the pairwise similarity in low dimensional space has the name of part2_xSNE_similarity_lowD_xxxperplexity.
>
> **[discussion]**
>
> 1. It seems that the result in both symmetric-SNE and t-SNE with 10 perplexity has the best classification result which seperate each group with different colored labels the most clearly. When the perplexity increase, the classification results become worse.
> 2. The similarity heat matrix in low dimesional space becomes more distinct as the perplexity's value raise.
> 3. The similarity distribution is sharper in both high or low dimensional space as the perplexity grows up.

c. observations and discussion (10%)

> Eigenface can be seen as the basic component of how human face is constructed. Any image of a human face can be considered a combination of these standard faces.