⚠️ **The gif animations and some markdown notations just won't work on pdf file! So please *refer to my hackmd or gif files in the zip file handed in to check the animations!***
my hackmd:
https://hackmd.io/@witty27818a/r1Ufpx-u9

(https://hackmd.io/@witty27818a/r1Ufpx-u9)

Code with detailed explanations

Kernel Eigenfaces

Preparations and Settings

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

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--input", help = 'the input path of Yale Face database
                        folders. 2 folders "Training" and "Testing" containing som
                        pgm files in the path', type = str, default = "D:\\HW7_ML_
    parser.add_argument("--output", help = "the path of output root folder",
                        type = str, default = ".\\output")
    parser.add_argument("--size", help = "the size to used in opening images.
                        size x size images.", type = int, default = 50)
    parser.add_argument("--dim", help = "number of principal components to be used
                        or the dimension after mapping in LDA", type = int,
                        default = 25)
    parser.add_argument("--seed", help = "random seed for reproducing results",
                        type = int, default = 100)
    args = parser.parse_args()
    np.random.seed(args.seed)

    train_path = os.path.join(args.input, 'Training')
    test_path = os.path.join(args.input, 'Testing')
    if not os.path.exists(train_path):
        raise OSError("Cannot open or find the training folder!")
    if not os.path.exists(test_path):
        raise OSError("Cannot open or find the testing folder!")
    if not os.path.exists(args.output):
        os.mkdir(args.output)

    size = args.size # 50, so (50, 50) images
    train_data,  train_settings, train_labels = read_files(train_path, size)
    test_data, test_settings, test_labels = read_files(test_path, size)
    # combining all data, setting names and labels in training and testing
    all_data = np.vstack((train_data, test_data)) #(165, 2500)
    all_settings = np.hstack((train_settings, test_settings)) #(165,)
    all_labels = np.hstack((train_labels, test_labels)) #(165,)
```

Firstly, import required packages and set some flags of arguments.

1. **–input**: the input path of the repossitory storing data of **Yale Face dataface**.
2. **–output**: the output path of root folder containing output result files and folders.
3. **–size**: the side length s of input images. Each image will be sized to $s \times s$.
   Default value = ***50***
4. **–dim**: in ***PCA*** or ***kernel PCA***, it denotes the number of principal components to be used, while in ***LDA*** or ***kernel LDA***, it denotes the dimension of projection space.
   Default value = ***25*** to align requests in **Part 1**.
5. **–seed**: the random seed. set seed for reproducing my results.
   Default value = ***100***

Try to validify the input paths, and create the output repository if it does not exist yet.
After that, we will read the *pgm* image files by the function **read_files()**, which will be introduced below. We will also record the ***settings*** and the ground truth ***labels*** for all files.
The ***settings*** refer to the 11 different facial expressions or configurations: *center-light, with glasses, happy, left-light, without glasses, normal, right-light, sad, sleepy, surprised,* and *wink,* with the subject number. As for the ***labels***, they refer to the subject number, from *1* to *15*.
To prepare for dimensionality reduction in part 1, we will combine the training data and testing data into `all_data`, `all_settings`, `all_labels`.

function: read_files()

```python
def read_files(path, size):
    # images array, settings names array, labels(integer) array
    data, settings, labels = [], [], []
    for pgm in os.listdir(path):
        image = Image.open(os.path.join(path, pgm))
        image = image.resize((size, size), resample = Image.ANTIALIAS)
        image = np.array(image).ravel().astype(np.float64)

        label = int(re.search(r"subject(\d+)", pgm, flags = re.I).group(1))

        data.append(image)
        settings.append(pgm.split('.')[1]+'_'+str(label))
        labels.append(label)
    return (np.asarray(data), np.asarray(settings), np.asarray(labels))
```

For each *pgm* image files, we open it and resize it to the $s \times s$, where s is the preset size, **50** in this implementation. We will use the ***ANTIALIAS*** filter when resizing. This is a kind of optical low-pass filter that is used when mapping multiple input pixels as an output pixel.
The pixels are then flattened into a $(2500,)$ numpy float type array.
As for label, which is just the subject number, can be taken directly from the file name by regular expression. We set `flags=re.I` in case the file name started in a capitalized letter: **Subject...**.
The settings are configurations + labels, extracted directly from file names.

Part 1

the main structure

```python
'''part 1: use PCA and LDA to show first 25 eigenfaces/fisherfaces.
Randomly pick 10 images to show their reconstruction.'''
random_idx = np.random.choice(all_data.shape[0], 10)
data_picked = all_data[random_idx] # data_picked = (10, 2500)
settings_picked = all_settings[random_idx] # (10,)

'''PCA: eigen faces'''
pc_number = args.dim # 25
mu, W = PCA(all_data, pc_number) # mu = (2500,) W = (2500, 25)
visualization(data_picked, settings_picked,
              os.path.join(args.output, 'eigenfaces'), size, W, mu)

'''LDA: fisher faces'''
q = args.dim # 25
W = LDA(all_data, all_labels, q)
visualization(data_picked, settings_picked,
              os.path.join(args.output, 'fisherfaces'), size, W)
```

In this part, we use both PCA and LDA to reduce the dimensionality, and pick out the first 25 eigenfaces/fisherfaces. Also, we randomly picked 10

images to show their reconstructions.
We first randomly pick 10 images in all data. Also, record the settings of these 10 images, for naming the visualization outputs.

In ***PCA*** mode, we set up the number of principal components `pc_number` as the value in argument *dim*. **25** in this implementation.
And then, we call the function **PCA()** to conduct principal component analysis, and get the projection matrix W and the mean vector $\vec{\mu}$, stored in `W` and `mu`, respectively.
Details about the processes will be given in the section **function: PCA()** below.
After that, we will use the 10 randomly picked images, as well as the projection matrix W and mean vector $\vec{\mu}$ we just derived, to visualize the 25 eigenfaces, as well as the 10 reconstructed faces, by the function **visualization()**. Again, I will explain the visualization details later in the section **function: visualization()**.

In ***LDA*** mode, we set up the dimension of the projection space, `q` as the value in argument *dim*. **25** in this implementation.
And then, we call the function **LDA()** to conduct principal component analysis, and get the projection matrix W, stored in `W`.
Details about the processes will be given in the section **function: LDA()** later.
After that, we will use the 10 randomly picked images, as well as the projection matrix W we just derived, to visualize the 25 fisherfaces, as well as the 10 reconstructed faces, by the function **visualization()**.
Again, I will explain the visualization details later in the section **function: visualization()**.

function: PCA()

In this function, we conduct the principal component analysis.

```
def PCA(data, PCnum):
    mu = np.mean(data, axis = 0) # mean vector, dim=size*size=50*50=2500
    demeaned = data - mu
    cov = np.cov(demeaned.T)

    eigvalues, eigvectors = np.linalg.eigh(cov)
    # normalization
    for i in range(eigvectors.shape[1]):
        eigvectors[:, i] /= np.linalg.norm(eigvectors[:, i])
    np.save("D:\\HW7_ML_params\\eigval_pca.npy", eigvalues)
    np.save("D:\\HW7_ML_params\\eigvec_pca.npy", eigvectors)
    eigvalues = np.load("D:\\HW7_ML_params\\eigval_pca.npy")
    eigvectors = np.load("D:\\HW7_ML_params\\eigvec_pca.npy")

    maxK_idx = np.argsort(eigvalues)[::-1][:PCnum] # max 25 eigvalues
    W = eigvectors[:, maxK_idx].real # discard imaginary part

    return mu, W

```

The data inpu `data` has the shape of $(165, 2500)$, where 165 is the number of all data, with each image has $50 \times 50 = 2500$ pixels.
First, we can take average over these 165 images, to get a mean vector $\vec{\mu}$, stored in the variable `mu`. `mu` has the shape of $(2500,)$.
Note that before conducting **PCA**, one important assumption is that the data must be ***demeaned (centered)***. So, we need to subtract the mean from the data first. Let the original input data be X and the mean vector be $\vec{\mu}$, then the centered data matrix would then be

$$X_c = X - \begin{bmatrix} \vec{\mu}^T \\ \vec{\mu}^T \\ \vdots \\ \vec{\mu}^T \end{bmatrix}$$

After that, we can compute the covariance matrix $Cov(X_c)$ of the demeaned data X_c, as

$$Cov(X_c) = X_c^T X_c$$

The result covariance matrix is a $(2500, 2500)$ matrix, since we have 2500 pixels, each can be deemed as a dimension. We will store the matrix in the variable `cov`. Note that we

don't divide the sample size $N = 165$ here for simplicity, because it does not affect the eigenvectors we get (we will do normalization).
Our goal is to find an orthogonal projection matrix W, such that the data after projection $Z = X_cW$ will have *maximum variance*. If we solve the optimization problem, we can find out that W is composed of the **25** first largest eigenvectors of the covariance matrix $Cov(X_c)$. So, we do the eigen decomposition on `cov`, by the help of `numpy.linalg.eigh()`.
After solving the eigenpairs, we will **normalize** these eigenvectors to unit length, which is also a constraint in PCA optimization problem.
As we say earlier, the projection matrix W consists of the eigenvectors corresponding to the largest k eigenvalues, where k is the reduced dimension of the projection space, a.k.a. the number of principal components taken `PCnum`. $k = 25$ in this implementation.

```
maxK_idx = np.argsort(eigvalues)[::-1][:PCnum] # max 25 eigvalues
W = eigvectors[:, maxK_idx].real # discard imaginary part
```

Finally, we return the projection matrix W and the mean vector $\vec{\mu}$. Both of these will be used while reconstructing images.

function: LDA()

In this function, we conduct the linear discriminant analysis.

```
def LDA(data, labels, dim):
    d = data.shape[1] # dimension in data space, 2500
    classes = np.unique(labels) # subject numbers
    mu = np.mean(data, axis = 0) # mean vector, dim=2500
    # initialize within-class scatter Sw and between-class scatter Sb
    Sw = np.zeros((d, d), dtype = np.float64)
    Sb = np.zeros_like(Sw, dtype = np.float64)

    # calculate Sw and Sb, by summating over each class
    for c in classes:
        data_c = data[np.where(labels == c)[0], :]
        mu_c = np.mean(data_c, axis = 0) # mean vector of the class c
        Sw += np.matmul((data_c - mu_c).T, (data_c - mu_c))
        Sb += data_c.shape[0] * np.matmul((mu_c - mu).T, (mu_c - mu))

    # since Sw might be non-invertible if the data size,
    # we use the pseudo inverse, by "pinv"
    # solve the eigen problem of matrix Sw^(-1)*Sb.
    eigvalues, eigvectors = np.linalg.eigh(np.matmul(np.linalg.pinv(Sw), Sb))
    # normalization
    for i in range(eigvectors.shape[1]):
        eigvectors[:, i] /= np.linalg.norm(eigvectors[:, i])
    np.save("D:\\HW7_ML_params\\eigval_lda.npy", eigvalues)
    np.save("D:\\HW7_ML_params\\eigvec_lda.npy", eigvectors)
    eigvalues = np.load("D:\\HW7_ML_params\\eigval_lda.npy")
    eigvectors = np.load("D:\\HW7_ML_params\\eigvec_lda.npy")

    maxq_idx = np.argsort(eigvalues)[::-1][:dim] # max 25 eigvalues
    W = eigvectors[:, maxq_idx].real # discard imaginary part

    return W
```

Firstly, the dimension of original data space is $d = 2500$ in this case.
Then, we will count the unique subject numbers. **Each subject will be treated as a single class.**
Therefore, there are **15** classes in total.
Then, we calculate the overall mean vector, stored in `mu`:

$$\vec{m} = \frac{1}{n}\sum \vec{x} = \frac{1}{165}\sum_{i=1}^{165} \vec{x}_i$$

```
mu = np.mean(data, axis = 0)
```

And then, we initialize 2 $d \times d = 2500 \times 2500$ matrixes for the ***within-class scatter*** and the ***between-class scatter***, S_W and S_B. These 2 matrixes can be calculated as follow.
For the ***within-class scatter***, S_W,

$$S_W = \sum_{j=1}^{15} (X_j - M_j)^T (X_j - M_j)$$

where X_j is data matrix composed of those belonging to class j, and M_j is a matrix having the same shape as X_j, with $\vec{m}_j$ being its every rows. n_j is the size of class j.

$$\vec{m}_j = \frac{1}{n_j} \sum_{i \in class\ j} \vec{x}_i$$

As for the ***between-class scatter***, S_B,

$$S_B = \sum_{j=1}^{15} n_j (M_j - M)^T (M_j - M)$$

where M has every rows equaled to $\vec{m}$. It is just a matrix from $\vec{m}$ broadcast to the shape of M_j.

```
for c in classes:
    data_c = data[np.where(labels == c)[0], :]
    mu_c = np.mean(data_c, axis = 0) # mean vector of the class c
    Sw += np.matmul((data_c - mu_c).T, (data_c - mu_c))
    Sb += data_c.shape[0] * np.matmul((mu_c - mu).T, (mu_c - mu))
```

After that, one can find that the projection matrix W into the target space with dimension $q = 25$ is a $d \times q = 2500 \times 25$ matrix. And its just composed of the eigenvectors corresponding to the $q = 25$ largest eigenvalues of $S_W^{-1} S_B$. So, we need to solve the eigenpairs for this matrix.
Note that we use `numpy.linalg.pinv()` here while finding S_W^{-1}. This is because that S_W becomes **non-invertible** when the size of the data is far less than the dimension of data space

$$n < D$$

which is true in this case ($165 < 2500$). So, we use **pseudo inverse** of S_W when the matrix is singular. Another way to avoid this is to conduct **PCA** first before doing LDA, but we won't do that here in this implementation. Because we want to check the performance generated purely on LDA.
The remaining processes are very similar as those in ***PCA***. After solving the eigenpairs, we will **normalize** these eigenvectors to unit length. Then, since the dimension of projected space is $q = 25$, we will pick the eigenvectors corresponding to the largest 25 eigenvalues to consist W.

Finally, we return the projection matrix W.

function: visualization()

This function is to draw out the 25 eigenfaces(or fisherfaces), as well as to reconstruct the 10 randomly chosen images after dimensionality reduction.

```
def visualization(data, settings, output_dir, size, W, mu = None):
    # data = (10, 2500)
    if mu is None:
        mu = np.zeros(data.shape[1]) # 2500
    demeaned = data - mu
    proj = np.matmul(demeaned, W) # (10, 25), each row a proj vector.
    reconstr = np.matmul(proj, W.T) + mu # (10, 2500), each row a reconstructed im
    if not os.path.exists(output_dir):
        os.mkdir(output_dir)
    dim = W.shape[1] # 25

    # 25 eigenfaces/fisherfaces
    if int((dim ** 0.5) + 0.5) ** 2 == dim:
    # if the PC number in PCA or dimension in LDA is a square number
        plt.clf()
        for i in range(dim): # 25
            plt.subplot(int(dim ** 0.5 + 0.5), int(dim ** 0.5 + 0.5), i+1) # 5 * 5
            plt.imshow(W[:, i].reshape((size, size)), cmap = 'gray')
            plt.axis('off')
        plt.savefig(os.path.join(output_dir, 'all.png'), bbox_inches = 'tight')
    for i in range(dim): # 25
        plt.clf()
        plt.imshow(W[:, i].reshape((size, size)), cmap = 'gray')
        plt.savefig(os.path.join(output_dir, '{}.png'.format(i+1)))

    # 10 reconstructed faces
    if reconstr.shape[0] == 10:
        plt.clf()
        for i in range(data.shape[0]):
            plt.subplot(2, 5, i+1)
            plt.imshow(reconstr[i].reshape((size, size)), cmap = 'gray')
            plt.axis('off')
        plt.savefig(os.path.join(output_dir, 'reconstructions.png'),
        bbox_inches = 'tight')
    for i in range(reconstr.shape[0]):
        plt.clf()
        plt.imshow(reconstr[i].reshape((size, size)), cmap = 'gray')
        plt.savefig(os.path.join(output_dir, settings[i] + "_reconstructed.png"))
```

We randomly picked 10 images from all data, each with 2500 pixels. So, the `data` has the shape of $(10, 2500)$. In **PCA**, we need to ***demean*** these data, by subtracting them from the mean vector of all data $\vec{\mu}$ we pass in, `mu` . However, in **LDA**, we don't need to do that. So, we set the mean vector as $\vec{\mu} = \vec{0}$ instead. Let X be the data matrix and X_c be the demeaned data matrix.

$$X_c = X - \vec{\mu} \qquad for\ PCA$$
$$X_c = X \qquad for\ LDA$$

$\vec{\mu}$ is broadcasted to the shape of X.
The next step, we will project these 10 images (centered) from original data space with dimension 2500, to the target space with dimension equaled to 25, by multiplying the X_c by the projection matrix W:

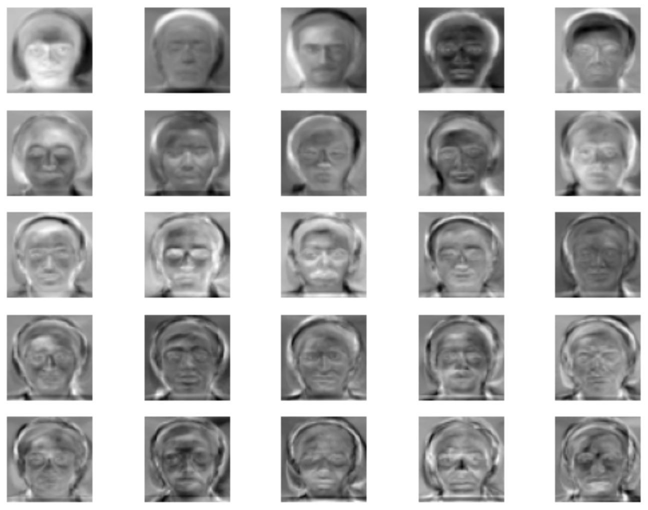
$$X_{proj} = X_c W$$

Now, X_{proj} has the shape of $(10, 25)$. And then, we reconstructed these images by multiplying the **inverse** of the orthogonal matrix W, which is just its transpose W^T. Also, remember that we demean our data at first in PCA, so we need to **add the mean vector** to shift the data back to the right place. We don't need to do this in LDA though.

$$\hat{X} = X_{proj} W^T + \vec{\mu}$$

Again, $\vec{\mu}$ is broadcasted to the shape of $\hat{X}$, which is $(10, 2500)$. These are the processes of reconstructing the images.
As for showing the **25** eigenfaces/fisherfaces, we just need to extract ***the 25 column eigenvectors of matrix*** W derived both in PCA and in LDA. These 25 eigenvectors are of *2500* dimensionality.
We can use them to draw 50×50 images, which are eigenfaces in PCA, or fisherfaces in LDA.
We will also draw all the eigenfaces/fisherfaces in a figure. However, due to visualization reason, only when the dimension in target space is a square number we do this step.

EX: 25 = 5 x 5, so we can arrange them like this



Part 2

In this part, we will do the face recognition by **PCA** and **LDA**. Each testing subject is classified by K nearest neighbor (KNN) method.

the main structure

```
'''part 2: face recognition by PCA and LDA. use KNN to cluster images'''
'''PCA'''
pc_number = args.dim # 25
K = np.arange(1, 17, 2) # cluster numbers to be tried: 1, 3, 5, ..., 15
mu, W = PCA(all_data, pc_number) # mu = (2500,) W = (2500, 25)
# demean training data and testing data, and get their projections, respectively.
train_proj = np.matmul(train_data - mu, W)
test_proj = np.matmul(test_data - mu, W)
# do the face recognition and print+write out the accuracy
face_recognition(train_proj, train_labels, test_proj, test_labels, K,
args.output, 'PCA')

'''LDA'''
q = args.dim # 25
K = np.arange(1, 17, 2) # cluster numbers to be tried: 1, 3, 5, ..., 15
W = LDA(all_data, all_labels, q) # W = (2500, 25)
# demean training data and testing data, and get their projections, respectively.
train_proj = np.matmul(train_data, W)
test_proj = np.matmul(test_data, W)
# do the face recognition and print+write out the accuracy
face_recognition(train_proj, train_labels, test_proj, test_labels, K,
args.output, 'LDA')
```

In ***PCA*** mode, we set up the number of principal components `pc_number` as the value in argument *dim*. **25** in this implementation. And as we said earlier, we will use ***KNN*** method to classify each testing image. The candidate number of nearest neighbors, K, ranges from **1 to 15**, with **2** as a step, namely

$$K \in \{1, 3, 5, 7, 9, 11, 13, 15\}$$

Next, we call the function **PCA()** to conduct principal component analysis, and get the projection matrix W and the mean vector $\vec{\mu}$, stored in `W` and `mu` , respectively.
Details about the processes are given in the section **function: PCA()** in ***Part 1***.
After that, we will use the projection matrix W, to project both the training data and testing data, and stored them in `train_proj` and `test_proj` , respectively. Let X^c denotes the demeaned data, and Z denotes the projected data:

$$\begin{gathered} X_{train}^c = X_{train} - \vec{\mu} \\ X_{test}^c = X_{test} - \vec{\mu} \\ Z_{train} = X_{train}^c W \\ Z_{test} = X_{test}^c W \end{gathered}$$

Both X_{train}, X_{train}^c have the shape of $(135, 2500)$, where 135 is the training data size and 2500 is the number of pixels. Both X_{test}, X_{test}^c have the shape of $(30, 2500)$, where 30 is the testing data size and 2500 is the number of pixels. The projected training data Z_{train} has the shape of $(135, 25)$, while the projected testing data Z_{test} has the shape of $(30, 25)$. The orthogonal projection matrix W has the shape of $(2500, 25)$, where 25 is the number of principal components as well as the dimension of target space. Finally, $\vec{\mu}$ is the overall mean vector with the shape $(2500,)$. It will be broadcasted to $(135, 2500)$ or $(30, 2500)$ in the calculations.

In the last step, we will do the face recognition by the function **face_recognition()**, and compute as well as save the performance scores in a text file to the folder `output_dir`.
Details about this function will be described later in the section **function: face_recognition()**.

In ***LDA*** mode, we set up the dimension of the projection space, `q` as the value in argument *dim*. **25** in this implementation. **25** in this implementation. And as we said earlier, we will use ***KNN*** method to classify each testing image. The candidate number of nearest neighbors, K, ranges from **1 to 15**, with **2** as a step, namely

$$K \in \{1, 3, 5, 7, 9, 11, 13, 15\}$$

Next, we call the function **LDA()** to conduct linear discriminant analysis, and get the projection matrix W, stored in `W`.
Details about the processes are given in the section **function: LDA()** in ***Part 1***.
After that, we will use the projection matrix W, to project both the training data and testing data, and stored them in `train_proj` and `test_proj`, respectively. Let Z denotes the projected data:

$$Z_{train} = X_{train}W$$
$$Z_{test} = X_{test}W$$

X_{train} has the shape of $(135, 2500)$, where 135 is the training data size and 2500 is the number of pixels. X_{test} has the shape of $(30, 2500)$, where 30 is the testing data size and 2500 is the number of pixels. The projected training data Z_{train} has the shape of $(135, 25)$ while the projected testing data Z_{test} has the shape of $(30, 25)$. The orthogonal projection matrix W has the shape of $(2500, 25)$, where 25 is the dimension of target space.
In the last step, we will do the face recognition by the function **face_recognition()**, and compute as well as save the performance scores in a text file to the folder `output_dir`.
Details about this function will be described later in the section **function: face_recognition()**.

function: face_recognition()

In this function, we will use the projected training and testing data generated by either **PCA**, **LDA**, **kernel PCA** or **kernel LDA** method, as well as the training labels and testing labels, and the list of candidate number of nearest neighbors, `K`, to conduct the *face recognition* task.
In cases of **kernel PCA** and **kernel LDA**, `kernel` type used need to be additionally specified, too.

```
def face_recognition(train, train_labels, test, test_labels, K,
output_dir, method, kernel = None):
    file = open(os.path.join(output_dir, "{}.txt".format(method)), mode = 'a')
    if kernel is None:
        print("method used: {}, clustering by: KNN\n".format(method))
        file.write("method used: {}, clustering by: KNN\n\n".format(method))
    else:
        print("method used: {}, clustering by: KNN, kernel type: {}\n".
        format(method, kernel))
        file.write("method used: {}, clustering by: KNN, kernel type: {}\n\n".
        format(method, kernel))

    # train: (135, 2500) test: (30, 2500)
    distances = [] # initialize a matrix that stores tuples
    '''the (i,j) element in distances is (a,b), where a is the squared
    Euclidean distance for ith testing data
    and jth training data, and b is the label of the jth training data.'''
    for i in range(test.shape[0]): # 30
        test_dist = [] # distance tuples between this testing and all training dat
        for j in range(train.shape[0]): # 135
            sqdist = cdist(test[i].reshape(1, -1), train[j].reshape(1, -1),
            metric = 'sqeuclidean').item()
            test_dist.append((sqdist, train_labels[j]))
        test_dist.sort(key = lambda x: x[0]) # sort by sqdist, min first
        distances.append(test_dist)
    distances = np.asarray(distances)

    # do KNN and get prediction results as well as accuracy
    for k in K:
        correct = 0
        for i in range(test.shape[0]):
            test_dist = distances[i] # distances between the testing images
            # and all training images.
            # pick out the labels of KNN
            KNN_labels = np.asarray([x[1] for x in test_dist[:k]])
            # get counts of all unique labels in KNN
            candidate_labels, counts = np.unique(KNN_labels, return_counts = True)
            prediction = candidate_labels[np.argmax(counts)]
            if prediction == test_labels[i]:
                correct += 1
        print("K = {:>2}, accuracy = {:>.3f} ({}/{})".format(k, correct/test.shape
        , correct, test.shape[0]))
        file.write("K = {:>2}, accuracy = {:>.3f} ({}/{})\n".format(k,
        correct/test.shape[0], correct, test.shape[0]))
    print('\n')
    file.write('\n')
    file.close()
```

The first double for loop:
To do the face recognition on testing images, we will first build a ***similarity matrix***, named `distances` in our code. The (i, j)-th element of this matrix a 2-tuple (a, b), where a denotes the ***squared Euclidean distance*** between the i-

th testing data and j**-th training data**, and b denotes the ***groun truth label*** (i.e. *subject number*) of the j**-th training data**.
In other words, `distances` has its every *row* vectors corresponding to a *testing data* and its distances with respect to all training data as well as the ground truth labels of those training data.
The second double for loop
After this distance matrix is constructed, in testing session, we need to use ***KNN*** algorithm to classify testing images. We will try out all candidate number of nearest number k in the candidate list K. $k \in K = \{1, 3, 5, 7, 9, 11, 13, 15\}$. The reason we choose k up to 15 is because that there are **at most 15 sujects** in this database. So, set 15 as the maximum number of clusters might be an appropriate choice.
In the KNN testing processes, we retrieve the distances row vector of each testing image i. And then, we will pick out k training data which have the shortest distances between them and the testing data. ***KNN*** algorithm states that we can make a prediction to the label of the testing data i, by **voting** the most appeared labels in the k nearest neighbors.
After all that, we will count the correctly classified number of testing data, and compute the accuracy. Export the performance results in a text file.

Part 3

In this part, we will do the face recognition by **kernel PCA** and **kernel LDA**. Each testing subject is classified by K nearest neighbor (KNN) method.

the main structure

```
'''part 3: face recognition by kernel PCA and kernel LDA.
use KNN to cluster images'''
'''kernel PCA'''
pc_number = args.dim # 25
K = np.arange(1, 17, 2)
kernels = ['linear', 'polynomial', 'RBF'] # 3 different kernels
for kernel in kernels:
    all_kernel_proj = kernelPCA(all_data, pc_number, kernel)
    # training data
    train_kernel_proj = all_kernel_proj[:train_data.shape[0]]
    # testing data
    test_kernel_proj = all_kernel_proj[train_data.shape[0]:]

    face_recognition(train_kernel_proj, train_labels, test_kernel_proj, test_label
    K, args.output, 'kernel_PCA', kernel)

'''kernel LDA'''
q = args.dim # 25
K = np.arange(1, 17, 2)
kernels = ['linear', 'polynomial', 'RBF']
for kernel in kernels:
    all_kernel_proj = kernelLDA(all_data, all_labels, q, kernel)
    # training data
    train_kernel_proj = all_kernel_proj[:train_data.shape[0]]
    # testing data
    test_kernel_proj = all_kernel_proj[train_data.shape[0]:]

    face_recognition(train_kernel_proj, train_labels, test_kernel_proj, test_label
    K, args.output, 'kernel_LDA', kernel)
```

In ***kernel PCA*** mode, we set up the number of principal components `pc_number` as the value in argument *dim*. **25** in this implementation. And as we said earlier, we will use ***KNN*** method to classify each testing image. The candidate number of nearest neighbors, K, ranges from **1 to 15**, with **2** as a step, namely

$$K \in \{1, 3, 5, 7, 9, 11, 13, 15\}$$

Next, we call the function **kernelPCA()** to conduct the *kernel version* principal component analysis. This function will directly return all the ***projected*** data, stored in `all_kernel_proj` , which has the shape of $(165, 25)$. 165 is the size of all data and 25 the is number of principal components a.k.a. the dimension of the target space. Details about the processes will be given in the section **function: kernelPCA()** later.
After that, we will split out the projected training data `train_kernel_proj` and the projected testing data `test_kernel_proj` from all the projected data `all_kernel_proj` .
The projected training data `train_kernel_proj` has the shape of $(135, 25)$, where 135 is the training data size and 25 is the number of principal components. The projected testing data `test_kernel_proj` has the shape of $(30, 25)$, where 30 is the testing data size and 25 is the number of principal components.
In the last step, we will do the face recognition by the function **face_recognition()**, and compute as well as save the performance scores in a text file to the folder `output_dir` .
Details about this function are already described in the section **function: face_recognition()** in ***Part 2***.

In ***kernel LDA*** mode, we set up the dimension of the target space, `q` as the value in argument *dim*. **25** in this implementation. And as we said earlier, we will use ***KNN*** method to classify each testing image. The candidate number of nearest neighbors, K, ranges from **1 to 15**, with **2** as a step, namely

$$K \in \{1, 3, 5, 7, 9, 11, 13, 15\}$$

Next, we call the function **kernelLDA()** to conduct the *kernel version* linear discriminant analysis. This function will directly return all the ***projected*** data, stored in `all_kernel_proj`, which has the shape of $(165, 25)$. 165 is the size of all data and 25 the is number of principal components a.k.a. the dimension of the target space. Details about the processes will be given in the section **function: kernelPCA()** later.
After that, we will split out the projected training data `train_kernel_proj` and the projected testing data `test_kernel_proj` from all the projected data `all_kernel_proj`.
The projected training data `train_kernel_proj` has the shape of $(135, 25)$, where 135 is the training data size and 25 is the number of principal components. The projected testing data `test_kernel_proj` has the shape of $(30, 25)$, where 30 is the testing data size and 25 is the number of principal components.
In the last step, we will do the face recognition by the function **face_recognition()**, and compute as well as save the performance scores in a text file to the folder `output_dir`. Details about this function are already described in section **function: face_recognition()** in ***Part 2***.

Last but not least, we would like to try out different kernels in this part as well.
The candidate kernels are ***linear kernel, polynomial kernel,*** and ***RBF kernel***. I will try out all these 3 kernels as you can see in my code implementation above.
Details about these kernels are listed in the section **function: compute_kernel()** below.

function: compute_kernel()

```python
def compute_kernel(data, kernel_type, params = None):
    if kernel_type == 'linear':
        return np.matmul(data, data.T)
    elif kernel_type == 'polynomial':
        gamma = 5
        coeff = 10
        degree = 2
        if params is not None:
            gamma, coeff, degree = params
        return np.power(gamma * np.matmul(data, data.T) + coeff, degree)
    else:
        gamma = 1e-7
        if params is not None:
            gamma = params[0]
        return np.exp(-gamma * cdist(data, data, metric = 'sqeuclidean'))
```

There are 3 kernels we would like to try out in **Part 3**. They are ***linear kernel, polynomial kernel,*** and ***RBF kernel***.
Assume we have 2 data vectors $\vec{x}_i$ and $\vec{x}_j$, then the kernel function between these 2 vectors are:
Linear kernel

$$k(\vec{x}_i, \vec{x}_j) = \vec{x}_i^T \vec{x}_j$$

Polynomial kernel

$$k(\vec{x}_i, \vec{x}_j) = (\gamma \vec{x}_i^T \vec{x}_j + c_0)^d$$

where γ, d, c_0 are all hyper-parameters, denoted in variables `gamma`, `degree` and `coeff` in our code implementation, respectively.
RBF kernel

$$k(\vec{x}_i, \vec{x}_j) = e^{-\gamma \|\vec{x}_i - \vec{x}_j\|^2}$$

where γ is a hyper-parameter, denoted `gamma` in our code implementation.
All these hyper-parameters are set adjustable. We would like to tune these parameters in the ***Observations and Discussion*** section at the end of the report. But for now, the default values:

1. Polynomial kernel
   1.1 $\gamma = 5$
   1.2 $d = 2$
   1.3 $c_0 = 10$
2. RBF kernel
   2.1 $\gamma = 10^{-7}$

function: kernelPCA()

In this function, we would like to conduct the kernel PCA with the given kernel type. We will directly return the projected data.

```
def kernelPCA(data, PCnum, kernel):
    K = compute_kernel(data, kernel) # get the Gram matrix
    N = K.shape[0] # the total data size
    # center the data in feature space for eigen decomposition
    # square matrix of N order with every element = 1/N
    oneN = np.full((N, N), fill_value = 1 / N, dtype = np.float64)
    Kc = K - np.matmul(oneN, K) - np.matmul(K, oneN)
    + np.linalg.multi_dot([oneN, K, oneN])

    eigvalues, eigvectors = np.linalg.eigh(Kc)
    for i in range(eigvectors.shape[1]):
        eigvectors[:, i] /= np.linalg.norm(eigvectors[:, i])
    np.save("D:\\HW7_ML_params\\eigval_kernel_pca_" + kernel + ".npy", eigvalues)
    np.save("D:\\HW7_ML_params\\eigvec_kernel_pca_" + kernel + ".npy", eigvectors)
    eigvalues = np.load("D:\\HW7_ML_params\\eigval_kernel_pca_" + kernel + ".npy")
    eigvectors = np.load("D:\\HW7_ML_params\\eigvec_kernel_pca_" + kernel + ".npy"

    maxK_idx = np.argsort(eigvalues)[::-1][:PCnum]
    W = eigvectors[:, maxK_idx].real

    all_kernel_proj = np.matmul(Kc, W)

    return all_kernel_proj
```

This is the kernel version of PCA, so in the first step, we will compute the Gram matrix K by the aforementioned function **compute_kernel()**. The return matrix K has the size of $(N, N) = (165, 165)$, where $N = 165$ is the total data size.

However, note that while applying PCA, one need to make sure that the data are ***centered***. To ensure that our kernel matrix K is centered, we can use the formula from the handout slide:

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

where K^C is the centered kernel matrix, and 1_N is a $N \times N = 165 \times 165$ matrix with every element $\frac{1}{N} = \frac{1}{165}$.
After centering the Gram matrix, we can use K^C to do the PCA. Again, the best orthogonal projection matrix W we are looking for is composed of the eigenvectors corresponding to the 25 (number of principal components to be used) eigenvalues of the centered Gram matrix K^c.
The remaining steps are the same as those described in **function: PCA()** section in ***Part 1***.
The shape of W is $(165, 25)$.
Finally, this function will directly returned all the projected data. Our centered kernels are K^c, so the projected data will be

$$K_{proj} = K^c W$$

function: kernelLDA()

In this function, we would like to conduct the kernel LDA with the given kernel type. We will directly return the projected data.

```python
def kernelLDA(data, labels, dim, kernel):
    classes = np.unique(labels)
    K = compute_kernel(data, kernel)
    N = K.shape[0] # 165
    mu = np.mean(K, axis = 0) # mean vector, dim=165
    # within-class scatter kernel version ker_Sw and
    # between-class scatter kernel version ker_Sb
    ker_Sw = np.zeros((N, N), dtype = np.float64)
    ker_Sb = np.zeros_like(ker_Sw, dtype = np.float64)

    # calculate ker_Sw and ker_Sb
    for c in classes:
        K_c = K[np.where(labels == c)[0], :]
        mu_c = np.mean(K_c, axis = 0)
        Nk = K_c.shape[0] # size of the cluster c.
        I = np.identity(Nk)
        oneNk = np.full((Nk, Nk), fill_value = 1 / Nk, dtype = np.float64)
        # this part needs some derivations
        ker_Sw += np.linalg.multi_dot([K_c.T, I - oneNk, K_c])
        ker_Sb += Nk * np.matmul((mu_c - mu).T, (mu_c - mu))

    eigvalues, eigvectors = np.linalg.eigh(np.matmul(np.linalg.pinv(ker_Sw), ker_S
    for i in range(eigvectors.shape[1]):
        eigvectors[:, i] /= np.linalg.norm(eigvectors[:, i])
    np.save("D:\\HW7_ML_params\\eigval_kernel_lda_" + kernel + ".npy", eigvalues)
    np.save("D:\\HW7_ML_params\\eigvec_kernel_lda_" + kernel + ".npy", eigvectors)
    eigvalues = np.load("D:\\HW7_ML_params\\eigval_kernel_lda_" + kernel + ".npy")
    eigvectors = np.load("D:\\HW7_ML_params\\eigvec_kernel_lda_" + kernel + ".npy"

    maxq_idx = np.argsort(eigvalues)[::-1][:dim]
    W = eigvectors[:, maxq_idx].real

    all_kernel_proj = np.matmul(K, W)

    return all_kernel_proj
```

This is the kernel version of LDA, so in the first step, we will compute the Gram matrix K by the aforementioned function **compute_kernel()**. The return matrix K has the size of $(N, N) = (165, 165)$, where $N = 165$ is the total data size.
Again, we will count the unique subject numbers. **Each subject will be treated as a single class.**
Therefore, there are **15** classes in total.
Then, we calculate the overall mean vector of the kernel matrix K, stored in `mu`:

$$\vec{m} = \begin{bmatrix} \frac{1}{165}\sum_{j=1}^{165} k(\vec{x}_1, \vec{x}_j) \\ \frac{1}{165}\sum_{j=1}^{165} k(\vec{x}_2, \vec{x}_j) \\ \vdots \\ \frac{1}{165}\sum_{j=1}^{165} k(\vec{x}_{165}, \vec{x}_j) \end{bmatrix}$$

```python
mu = np.mean(K, axis = 0)
```

`mu` has the shape of $(165,)$.

And then, we initialize 2 $N \times N = 165 \times 165$ matrixes for the ***within-class scatter*** and the ***between-class scatter***, S_W^{ker} and S_B^{ker}. These 2 matrixes can be calculated as follow.

For the ***between-class scatter***, S_B^{ker}, it is very similar to that in ordinary PCA version:

$$S_B^{ker} = \sum_{j=1}^{15} n_j (M_j - M)^T (M_j - M)$$

where M has every rows equaled to $\vec{m}^T$. It is just a matrix from $\vec{m}^T$ broadcast to the shape of M_j. And M_j is a matrix with $\vec{m}_j^T$ being its every rows. n_j is the size of class j.

$$\vec{m}_j = \begin{bmatrix} \frac{1}{n_j} \sum_{k \in class\ j} k(\vec{x}_1, \vec{x}_k) \\ \frac{1}{n_j} \sum_{k \in class\ j} k(\vec{x}_2, \vec{x}_k) \\ \vdots \\ \frac{1}{n_j} \sum_{k \in class\ j} k(\vec{x}_{n_j}, \vec{x}_k) \end{bmatrix}$$

As for the ***within-class scatter***, S_W^{ker}, it can be shown that the matrix is actually:

$$S_W^{ker} = \sum_{j=1}^{15} K_j^T (I - 1_{n_j}) K_j$$

where K_j is the matrix containing only kernel functions related to *class* j. Let n_j denotes the size of class j, then K_j has the shape of $(n_j, 165)$. I is the identity matrix of order n_j. And finally, 1_{n_j} is a $n_j \times n_j$ matrix with every element equaled to $\frac{1}{n_j}$.

For the detailed derivations, please refer to this article:
https://zhuanlan.zhihu.com/p/92359921

(https://zhuanlan.zhihu.com/p/92359921).

```python
for c in classes:
    K_c = K[np.where(labels == c)[0], :]
    mu_c = np.mean(K_c, axis = 0)
    Nk = K_c.shape[0] # size of the cluster c.
    I = np.identity(Nk)
    oneNk = np.full((Nk, Nk), fill_value = 1 / Nk, dtype = np.float64)
    # this part needs some derivations
    ker_Sw += np.linalg.multi_dot([K_c.T, I - oneNk, K_c])
    ker_Sb += Nk * np.matmul((mu_c - mu).T, (mu_c - mu))
```

After that, one can find that the projection matrix W into the target space with dimension $q = 25$ is a $N \times q = 165 \times 25$ matrix. And its just composed of the eigenvectors corresponding to the $q = 25$ largest eigenvectors of $(S_W^{ker})^{-1} S_B^{ker}$. So, we need to solve the eigenpairs for this matrix.
Again, we use `numpy.linalg.pinv()` here while finding $(S_W^{ker})^{-1}$ for preventing the **non-invertibility** of S_W^{ker}.
The remaining processes are very similar as those in **LDA()** in ***Part 1***. After solving the eigenpairs, we will **normalize** these eigenvectors to unit length. Then, since the dimension of projected space is $q = 25$, we will pick the eigenvectors corresponding to the largest 25 eigenvalues to consist W.
W has the shape of $(165, 25)$.
Finally, this function will directly returned all the projected data. Our overall kernels are K, so the projected data will be

$$K_{proj} = KW$$

t-SNE

Part 1

In this part, we would like to modify the code given: ***tsne.py*** (http://tsne.py), so that it can conduct ***symmetric SNE*** as well.
Given a **perplexity**, we can compute the distribution P in the original ***high dimensional space***, as the following function does. We don't modify this function.

```python
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
```

The following code is the function **tsne()** with some modification done.

```python
def tsne(X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0, method = 'ts
    """
        Runs t-SNE on the dataset in the NxD array X to reduce its
        dimensionality to no_dims dimensions. The syntaxis of the function is
        `Y = tsne.tsne(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))
    drawout = 50

    # 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
        '''in tsne and ssne, the only difference happened in qij, i.e. the distrib
        if method == 'tsne':
            num = 1 / (1 + cdist(Y, Y, metric = 'sqeuclidean'))
        else: # ssne
            num = np.exp(-cdist(Y, Y, metric = 'sqeuclidean'))
        # 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)

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

        '''part 2: draw out a result image every drawout=50 iterations'''
        if (iter + 1) % drawout == 0:
            visualization(Y, labels, iter + 1, method, perplexity)

        # 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.

    # Return solution
    '''Part 3: as well as distributions P and Q'''
    return Y, P, Q
```

In ***t-SNE***, the distribution Q of the ***low dimensional space*** is built on ***student t-distribution***.
This allows that far away points stay relatively far in t-SNE, compared with those in symmetric SNE, which use ordinary Gaussian distribution in the low dimensional space.

$$q_{ij} = \frac{(1 + \|\vec{y}_i - \vec{y}_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|\vec{y}_i - \vec{y}_j\|^2)^{-1}}$$

```
if method == 'tsne':
    num = 1 / (1 + cdist(Y, Y, metric = 'sqeuclidean'))
```

In ***symmetric-SNE***, the distribution Q of the ***low dimensional space*** is assumed to be ***Gaussian distribution***.

$$q_{ij} = \frac{e^{-\|\vec{y}_i - \vec{y}_j\|^2}}{\sum_{k \neq l} e^{-\|\vec{y}_i - \vec{y}_j\|^2}}$$

```
else: # ssne
    num = np.exp(-cdist(Y, Y, metric = 'sqeuclidean'))
```

As for the distribution P of the ***high dimensional space***, both symmetric SNE and t-SNE use the ***Gaussian distribution***.

$$p_{ij} = \frac{e^{\frac{-\|\vec{x}_i - \vec{x}_j\|^2}{2\sigma^2}}}{\sum_{k \neq l} e^{\frac{-\|\vec{x}_i - \vec{x}_j\|^2}{2\sigma^2}}}$$

The difference between ***symmetric-SNE*** and ***t-SNE*** yields a difference in the results of dimensionality reduction.

t-SNE solves the ***crowded problem*** in symmetric SNE.

I will show the results later in ***Experiment Results & Discussion*** section.

Part 2

In this part, we would like to visualize the 2D embeddings of both symmetric SNE and t-SNE, and then make an animated GIF file to show the optimize procedure.
The following code is added in the function **tsne()**. You can find the position in **Part 1** above.

```
'''part 2: draw out a result image every drawout=50 iterations'''
if (iter + 1) % drawout == 0:
    visualization(Y, labels, iter + 1, method, perplexity)
```

There are totally 1000 iterations in optimizing procedure by default. We set that every `drawout` = ***50*** iterations, we will visualize the current 2D embedding result, by the help of function **visualization()**.
And then finally, outside all the iterations loop, we will call the function **GIF()** to gather these $\frac{1000}{50} = 20$ result embedding images and make a GIF animation.
Both implementation details of **visualization()** and **GIF()** are described as belows.

function: visualization()

```
def visualization(Y, labels, epoch, method, perplexity):
    plt.clf()
    scatter_plot = plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
    plt.legend(*scatter_plot.legend_elements(), loc = 'lower left',
    title = 'Num Digits')
    plt.title("method: {}, perplexity: {}\nepoch: {}".format(method, perplexity,
    epoch))
    plt.tight_layout()
    plt.savefig("output/{}/epoch{:4d}_perp{}.png".format(method, epoch, perplexity
```

In this function, we visualize the current 2D embeddings, which are stored in the matrix Y. Y has 2 columns, and every row denotes a 2D embedding coordinate for some data point.
This function will first use `matplotlib.pyplot.clf()` to clean the cached image before, and then call function `matplotlib.pyplot.scatter()` to draw out the 2D embeddings of every data points based on the

coordinated provided in Y.
Also, we will color each data points based on their ground truth labels, which are their **number digits** in this case.
We set the titles, legends and file names properly, then output the images.

function: GIF()

```python
def GIF(output_dir, perplexity):
    # output_dir must contain all the images for making gif
    images_arr = []
    for file in os.listdir(output_dir):
        if ("epoch" in file) and (str(perplexity) in file):
            images_arr.append(imageio.imread(os.path.join(output_dir, file)))
    imageio.mimsave(os.path.join(output_dir, "animation_perp{}.gif".
    format(perplexity)), images_arr, fps = 1.5)
```

This function is used to gather all the embedding images previously saved to make a GIF animation. Since, we name our embedding images properly, we can easily find them.
We will use `imread()` function from package **imageio** to open these images, and gather them in a list called `images_arr`.
And then, we use `mimsave()` function from the same package to make a GIF animation.

Part 3

In this part, we would like to visualize the ***pairwise similarities*** in both high-dimensional space and low-dimensional space, based on the symmetric SNE and t-SNE.
In the function **tsne()**, we not only return the embeddings matrix Y, but also return the distribution P in **high-dimensional space** and the distribution Q in **low-dimensional space**.
You can find the position in **Part 1** above.

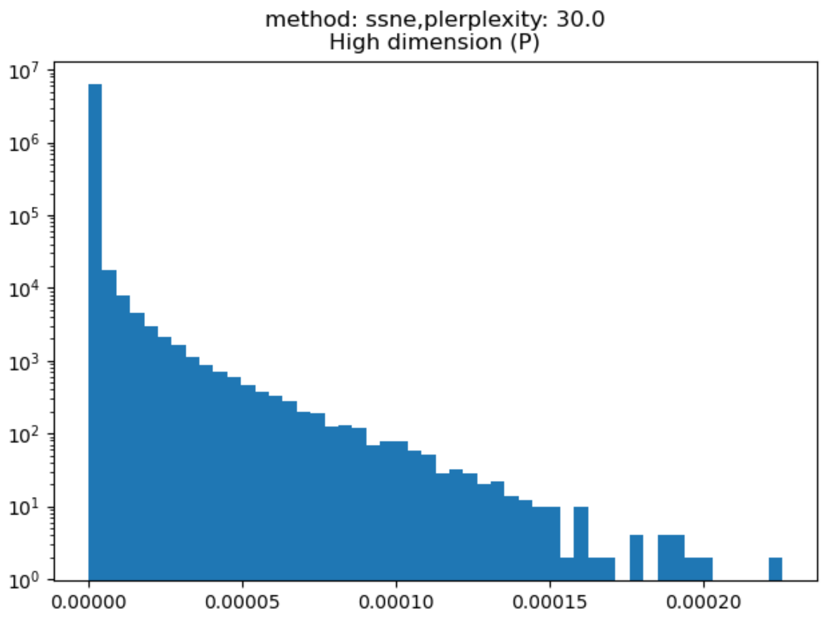
```python
# Return solution
'''Part 3: as well as distributions P and Q'''
return Y, P, Q
```

Later on, in function **plotPQ()**, we will plot out the histogram of both distributions P and Q. Please refer to the section below.

function: plotPQ()

```
def plotPQ(Y, P, Q, method, perplexity):
    plt.clf()
    plt.title("method: {},plerplexity: {}\nHigh dimension (P)".
    format(method, perplexity))
    plt.hist(P.flatten(), bins = 50, log = True)
    plt.savefig("output/{}/P_perp{}.png".format(method, perplexity))
    plt.clf()
    plt.title("method: {},plerplexity: {}\nLow dimension (Q)".
    format(method, perplexity))
    plt.hist(Q.flatten(), bins = 50, log = True)
    plt.savefig("output/{}/Q_perp{}.png".format(method, perplexity))
```

In this function, we use the distribution of the pairwise similarities in ***high-dimensional*** space, P, as well as the distribution of the pairwise similarities in ***low-dimensional*** space, Q, to draw out 2 histograms.
About these 2 histograms, the x-axis is set to be the joint probability values, namely **pairwise similarities**. As for the y-axis, it's just the frequency counts of each bar, like so:



Note that since these frequency counts range too wide, we apply ***logarithm scale*** in the y-axis.

```
'''log = True'''
plt.hist(P.flatten(), bins = 50, log = True)
plt.hist(Q.flatten(), bins = 50, log = True)
```

Part 4

Lastly, in this part, we try to play with different **perplexity** values, and then visualize the results.

```python
'''part 4'''
candidate_perplexity = [3.0, 300.0] # 10 times smaller and bigger
'''method: tsne'''
for perplexity in candidate_perplexity:
    method = 'tsne'
    Y, P, Q = tsne(X, 2, 50, perplexity = perplexity, method = method)
    plotPQ(Y, P, Q, method, perplexity)
    output_dir = 'output/tsne'
    GIF(output_dir, perplexity)
    method = 'ssne'
    Y, P, Q = tsne(X, 2, 50, perplexity = perplexity, method = method)
    plotPQ(Y, P, Q, method, perplexity)
    output_dir = 'output/ssne'
    GIF(output_dir, perplexity)
```

Aside from the default perplexity **30**, other candidate perplexity values we've tried are **3** and **300**, which are just 10 times larger and 10 times smaller, respectively. We store 3 and 300 in the list `candidate_perplexity` , and use a for loop to run the whole processes.
We don't contain 30 here simply because we've done them in part 1~3.
The whole processes are basically the same, just change the perplexity values:

1. Specify the method to be used, namely both ***symmetric SNE*** and ***t-SNE***
2. > <Task in ***Part 1***>
   > By function **tsne()**, compute the 2D embeddings Y, as well as the pairwise similarities in **high-dimensional** space (P) and in **low-dimensional** space (Q).
3. > <Task in ***Part 3***>
   > Plot the distributions of P and Q, by function **plotPQ()**.
4. > <Task in ***Part 2***>
   > Input all the embedding images in every 50 iterations, and make them into a GIF animation by the function **GIF()**

Experiments Results & Discussion

Kernel Eigenfaces

Part 1

Experimental Settings:

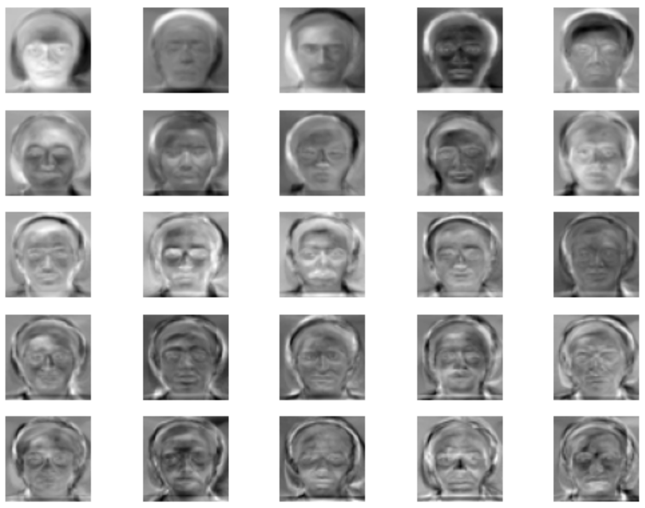
<PCA>

1. the number of principal components used: $PCnum = 25$
2. images sizes: 50×50 resolution
3. random seed: 100

<LDA>

1. the dimension of target space: $q = 25$
2. images sizes: 50×50 resolution
3. random seed: 100

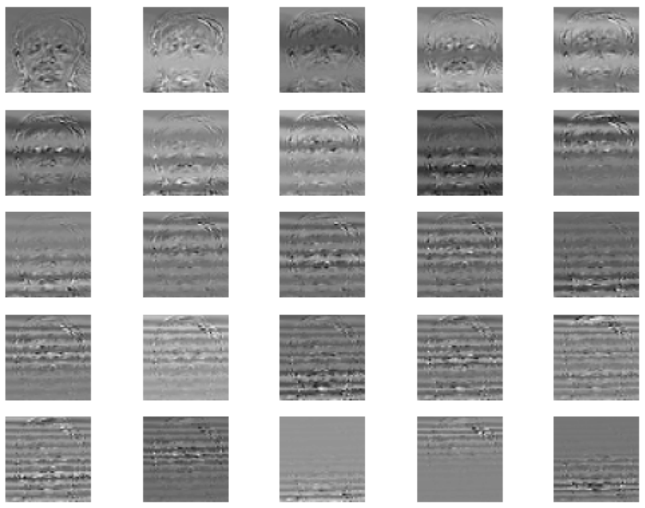
These are the 25 ***eigenfaces***, generated by ***PCA*** method.



file name: `output/eigenfaces/all.png`

(If you would like to check each image individually, please go to the output folder I handed in. These 25 images are `output/eigenfaces/1.png` to `output/eigenfaces/25.png`)

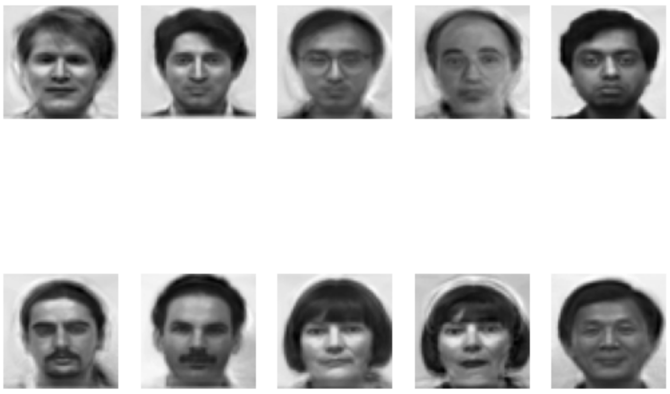
These are the 25 ***fisherfaces***, generated by ***LDA*** method.



file name: `output/fisherfaces/all.png`

(Again, if you would like to check each image individually, please go to the output folder I handed in. These 25 images are `output/fisherfaces/1.png` to `output/fisherfaces/25.png`)

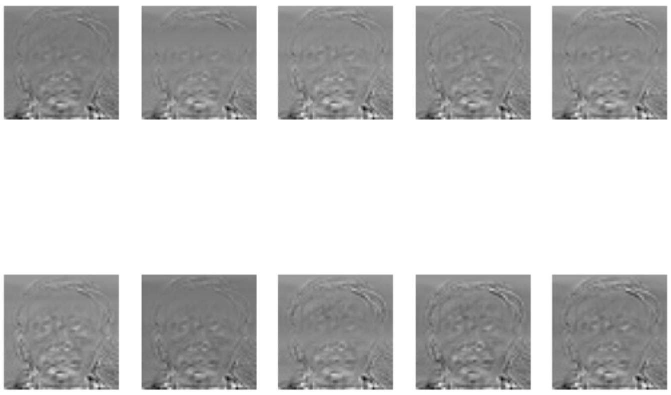
The following images are 10 randomly picked images, projected by **PCA** then reconstructed.



file name: `output/eigenfaces/reconstructions.png`

(If you would like to check each image individually, please go to the output folder I handed in.
These 10 reconstructed images are within `output/eigenfaces`)

And the following images are 10 randomly picked images, projected by **LDA** then reconstructed.

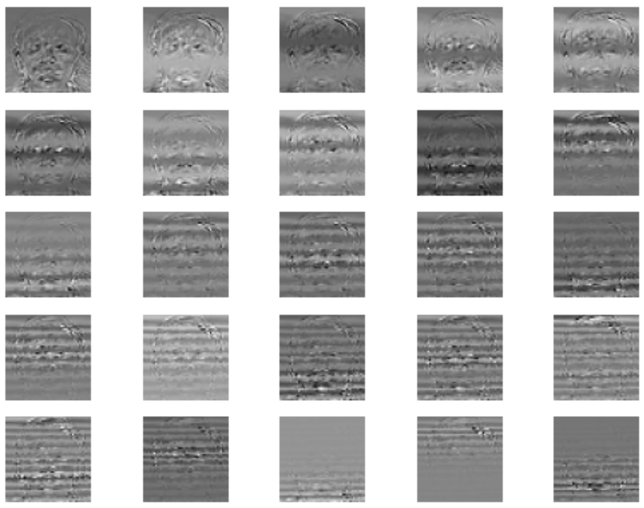


file name: `output/fisherfaces/reconstructions.png`

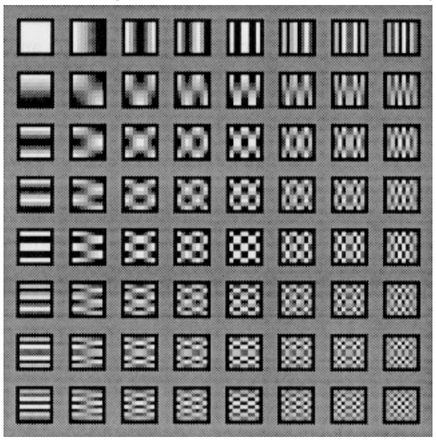
(If you would like to check each image individually, please go to the output folder I handed in.
These 10 reconstructed images are within `output/fisherfaces`)

One can see that ***eigenfaces*** are so much clearer than fisherfaces. The main reason behind this might be that PCA focuses on ***maximizing variances***, which helps the method capturing most characteristics behind all images. Those eigenfaces constructed by the eigenvectors

corresponding to the first few largest eigenvalues shows the main structure behind all these input images, and has bigger explainable variances.
On the other hand, ***fisherfaces*** performs quite bad. One can see that almost every fisherfaces and reconstructed images looks exactly the same. This might because that LDA mainly considers maximizing ***between-class scatter*** and minimizing ***within-class scatter***, which makes it more proficient in **classfication task** rather than reconstruction task.
Finally, there's an interesting phenomenon in part 1. One can see that the 25 **fisherfaces** all have some kinds of different ***interference fringes(干涉條紋)***.



In fisherface **1**, there's **no** any fringe; in fisherface **2**, there are **1** fringe; in fisherface **3**, there are **2** fringes (one white on black), and so on and so forth. This kinda looks like those patterns appeared in **discrete cosine transform**, although I don't think they have any relations.



This is really intriguing, though I don't know why exactly **LDA** would yield such results.

Part 2

Experimental Settings:

<PCA>

1. the number of principal components used: $PCnum = 25$
2. images sizes: 50×50 resolution
3. random seed: 100
4. candidate neighbor numbers for KNN: $\{1, 3, 5, 7, 9, 11, 13, 15\}$

<LDA>

1. the dimension of target space: $q = 25$
2. images sizes: 50×50 resolution
3. random seed: 100
4. candidate neighbor numbers for KNN: $\{1, 3, 5, 7, 9, 11, 13, 15\}$

The following are the face recognition performances with different number of nearest neighbors (K), in **PCA** method. You can find the text file in `output/PCA.txt` .

```
method used: PCA, clustering by: KNN
K = 1, accuracy = 0.833 (25/30)
K = 3, accuracy = 0.833 (25/30)
K = 5, accuracy = 0.900 (27/30)
K = 7, accuracy = 0.900 (27/30)
K = 9, accuracy = 0.833 (25/30)
K = 11, accuracy = 0.800 (24/30)
K = 13, accuracy = 0.833 (25/30)
K = 15, accuracy = 0.800 (24/30)
```

The best accuracy **90%** happened when $K = 5, 7$.

The following are the face recognition performances with different number of nearest neighbors (K), in **LDA** method. You can find the text file in `output/LDA.txt` .

```
method used: LDA, clustering by: KNN
K = 1, accuracy = 0.867 (26/30)
K = 3, accuracy = 0.867 (26/30)
K = 5, accuracy = 0.833 (25/30)
K = 7, accuracy = 0.800 (24/30)
K = 9, accuracy = 0.767 (23/30)
K = 11, accuracy = 0.767 (23/30)
K = 13, accuracy = 0.767 (23/30)
K = 15, accuracy = 0.767 (23/30)
```

The best accuracy **86.7%** happened when $K = 1, 3$. The performance decreases as we see more neighbors around each data point.
The **PCA** method performs better in higher K values while **LDA** method is better in lower K values. However, in general, **PCA** outperforms the **LDA** method, which is a little weird since LDA is a ***supervised*** method and is good at this kind of **classification task**. So theorectically, **LDA** should have better performances.
After some research, I think that this might be due to lacks of data ($n = 135$ only for training). If we train on **LDA** using too few data, sometimes it will lead us some *overfitting* results.

Part 3

Experimental Settings:

<PCA>

1. the number of principal components used: $PCnum = 25$
2. images sizes: 50×50 resolution
3. random seed: 100
4. candidate neighbor numbers for KNN: $\{1, 3, 5, 7, 9, 11, 13, 15\}$
5. kernels used: ***linear kernel, polynomial kernel, RBF kernel***
   5.1 **linear kernel**: no hyper-parameter
   5.2 **polynomial kernel**: $\gamma = 5, c_0 = 10, d = 2$
   5.3 **RBF kernel**: $\gamma = 10^{-7}$

<LDA>

1. the dimension of target space: $q = 25$
2. images sizes: 50×50 resolution
3. random seed: 100
4. candidate neighbor numbers for KNN: $\{1, 3, 5, 7, 9, 11, 13, 15\}$
5. kernels used: ***linear kernel, polynomial kernel, RBF kernel***
   5.1 **linear kernel**: no hyper-parameter
   5.2 **polynomial kernel**: $\gamma = 5, c_0 = 10, d = 2$
   5.3 **RBF kernel**: $\gamma = 10^{-7}$

The following are the face recognition performances with different number of nearest neighbors (K), in **kerenl PCA** method. You can find the text file in `output/kernel_PCA.txt` . We have tried 3 different kernels as well: ***linear kernel, polynomial kernel, RBF kernel***.

```
method used: kernel_PCA, clustering by: KNN,
kernel type: linear
K = 1, accuracy = 0.800 (24/30)
K = 3, accuracy = 0.833 (25/30)
K = 5, accuracy = 0.833 (25/30)
K = 7, accuracy = 0.800 (24/30)
K = 9, accuracy = 0.833 (25/30)
K = 11, accuracy = 0.867 (26/30)
K = 13, accuracy = 0.800 (24/30)
K = 15, accuracy = 0.800 (24/30)
method used: kernel_PCA, clustering by: KNN,
kernel type: polynomial
K = 1, accuracy = 0.800 (24/30)
K = 3, accuracy = 0.833 (25/30)
K = 5, accuracy = 0.867 (26/30)
K = 7, accuracy = 0.833 (25/30)
K = 9, accuracy = 0.833 (25/30)
K = 11, accuracy = 0.867 (26/30)
K = 13, accuracy = 0.767 (23/30)
K = 15, accuracy = 0.767 (23/30)
method used: kernel_PCA, clustering by: KNN,
kernel type: RBF
K = 1, accuracy = 0.833 (25/30)
K = 3, accuracy = 0.833 (25/30)
K = 5, accuracy = 0.800 (24/30)
K = 7, accuracy = 0.767 (23/30)
K = 9, accuracy = 0.833 (25/30)
K = 11, accuracy = 0.800 (24/30)
K = 13, accuracy = 0.800 (24/30)
K = 15, accuracy = 0.767 (23/30)
```

The best accuracy **86.7%** happened in **polynomial kernel** with $K = 5, 11$ and **linear kernel** with $K = 11$.

The following are the face recognition performances with different number of nearest neighbors (K), in **kerenl LDA** method. You can find the text file in `output/kernel_LDA.txt` . We have tried 3 different kernels as well: ***linear kernel, polynomial kernel, RBF kernel***.

```
method used: kernel_LDA, clustering by: KNN,
kernel type: linear
K = 1, accuracy = 0.767 (23/30)
K = 3, accuracy = 0.700 (21/30)
K = 5, accuracy = 0.600 (18/30)
K = 7, accuracy = 0.600 (18/30)
K = 9, accuracy = 0.567 (17/30)
K = 11, accuracy = 0.500 (15/30)
K = 13, accuracy = 0.533 (16/30)
K = 15, accuracy = 0.500 (15/30)
method used: kernel_LDA, clustering by: KNN,
kernel type: polynomial
K = 1, accuracy = 0.767 (23/30)
K = 3, accuracy = 0.700 (21/30)
K = 5, accuracy = 0.600 (18/30)
K = 7, accuracy = 0.567 (17/30)
K = 9, accuracy = 0.567 (17/30)
K = 11, accuracy = 0.467 (14/30)
K = 13, accuracy = 0.433 (13/30)
K = 15, accuracy = 0.500 (15/30)
method used: kernel_LDA, clustering by: KNN,
kernel type: RBF
K = 1, accuracy = 0.833 (25/30)
K = 3, accuracy = 0.800 (24/30)
K = 5, accuracy = 0.800 (24/30)
K = 7, accuracy = 0.833 (25/30)
K = 9, accuracy = 0.800 (24/30)
K = 11, accuracy = 0.733 (22/30)
K = 13, accuracy = 0.733 (22/30)
K = 15, accuracy = 0.767 (23/30)
```

The best accuracy **83.3%** happened in **RBF kernel**, with $K = 1, 7$.

One can see that no matter PCA or LDA, kernel versions of these 2 methods **do not** outperform the original version. This might be due to the fact that the performances of these kinds of ***kernel methods*** are heavily effected by ***hyper-parameters***. So, fine-tuning these hyper-

parameters might help a lot. Therefore, we would like to try tunning these parameters in the later ***Observations and Discussion*** section.
Also, the best accuracy in **kernel PCA** is generated by **linear kernel** and **polynomial kernel**, while the best accuracy in **kernel LDA** is generated by **RBF kernel**. These might be owing to the intrinsic properties of these 2 methods, making them suitable for different kinds of kernels.

t-SNE

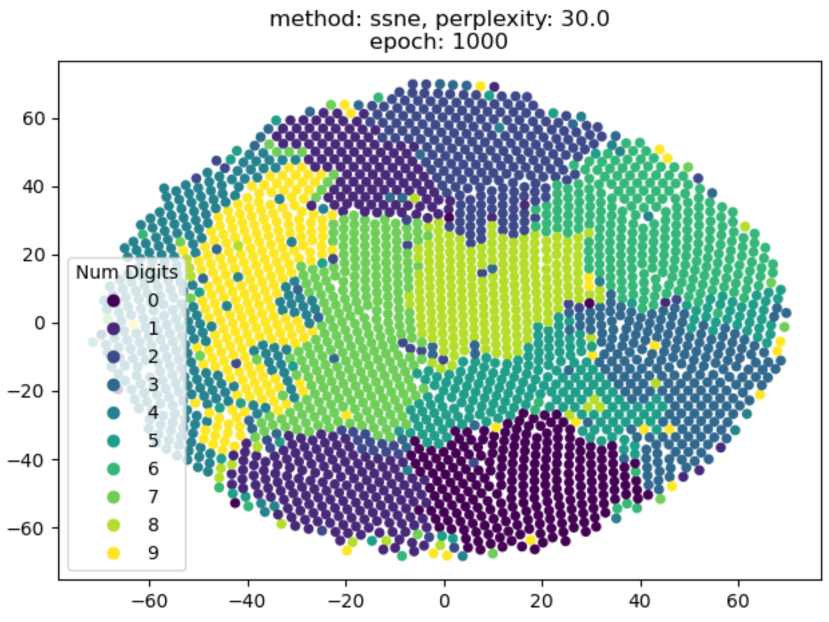
Part 1

Experimental Settings
<symmetric SNE>, **<t-SNE>**

1. dimensionality of original data space: **50**
2. dimensionality of reduced target space: **2**
3. perplexity: **30.0**
4. iterations: **1000** epochs

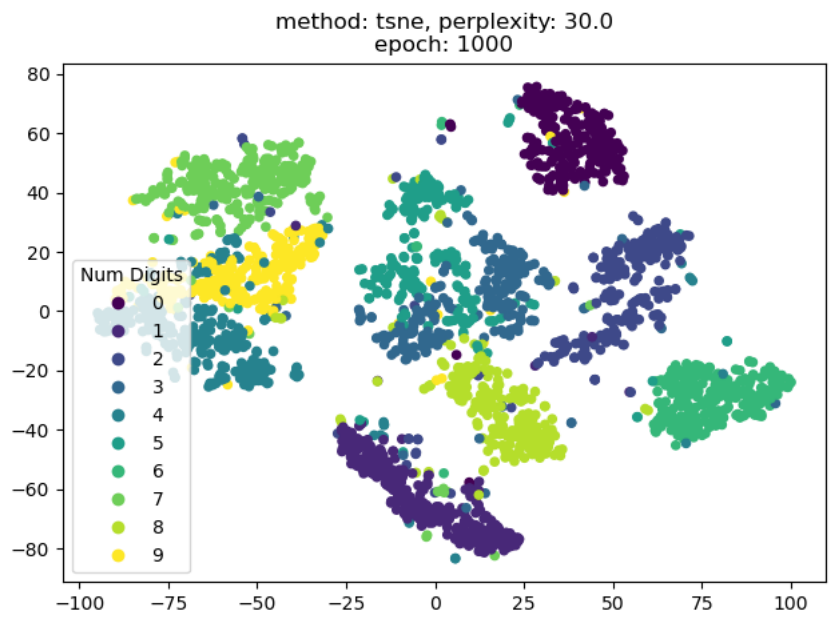
Other parameters are left as default values as in the original **tsne.py** (http://tsne.py).

This is the final embeddings result of **symmetric SNE**



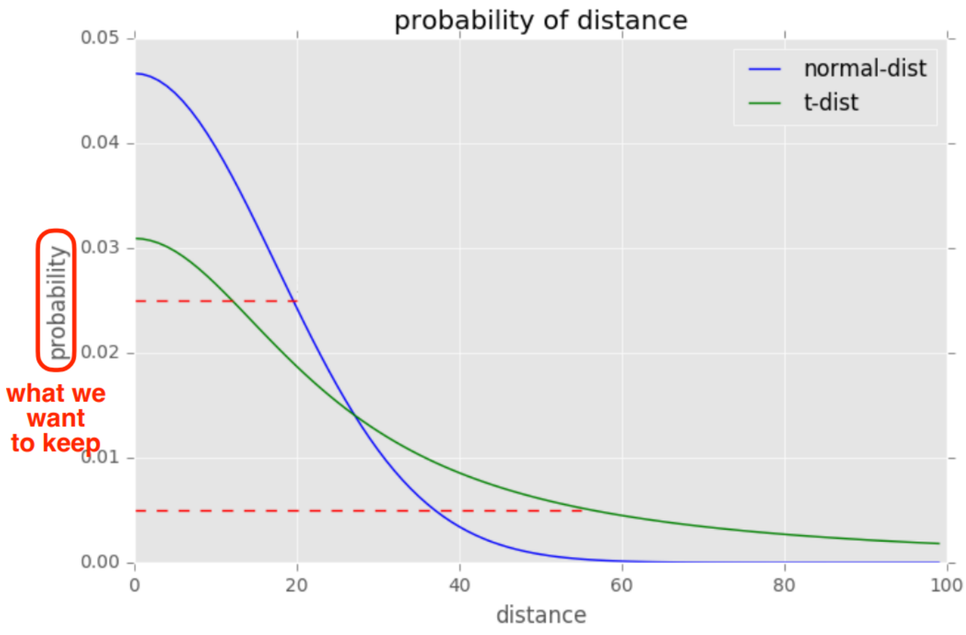
file name: `output/ssne/epoch1000_perp30.0.png`

And this is the final embeddings result of **t-SNE**



file name: `output/tsne/epoch1000_perp30.0.png`

It's quite obvious that the embeddings using **symmetric SNE** method are much more ***crowded***. The crowded problem might be owing to the fact that *Gaussian distribution* has a *thinner* tail probability, which makes those supposedly faraway data points cannot be pulled away in low-dimensional space, if the normal probability is used in low-dimensional space.



Also, from the cost functions of **symmetric SNE**:

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

One can see that for points where p_{ij} is large (considered close in high dimensional space) and q_{ij} is small (considered faraway in low dimensional space), we lose ***a lot***, since SNE prefers to preserve local structure. On the other hand, for points where q_{ij} is large (considered close in low dimensional space) and p_{ij} is large (considered faraway in high dimensional space), we lose ***a little***. This means that the penalty of pulling faraway data points close to each others in low-dimensional space is quite mild. This might lead to crowded problem.
Other reasons of crowded problem are:

1. In a high-D space, points can have many close-by neighbors in different directions. However, in a 2D space, you essentially have to arrange close-by neighbors in a circle around the central point, which constrains relationships among neighbors.
2. 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.
3. Volume of a sphere scales as r^d in d dimensions → on a 2D display, there is much less area available at radius r (r^2) than the corresponding volume in the original space.

To overcome this, ***t-SNE*** method use **student t-distribution** instead of **normal distribution** in low-dimensional space. Data should be further away in low-dimension in order to achieve low probability. So, one can see that ***t-SNE* has much clearer and nicer embeddings than those of the *symmetric SNE***.

Part 2

⚠️ **The gif animations and some markdown notations just won't work on pdf file! So please *refer to my hackmd or gif files in the zip file handed in to check the animations!***
my hackmd:
https://hackmd.io/@witty27818a/r1Ufpx-u9

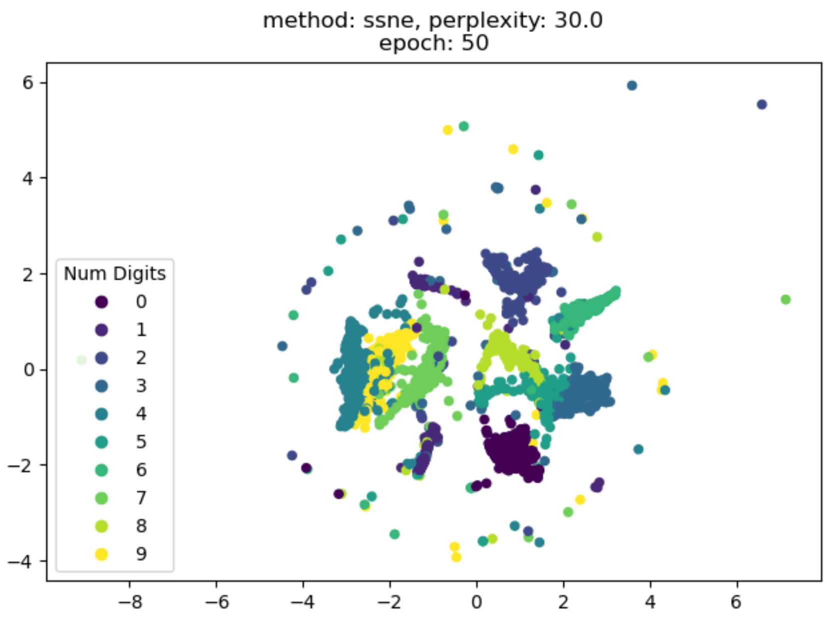
(https://hackmd.io/@witty27818a/r1Ufpx-u9)

Experimental Settings
<symmetric SNE>, **<t-SNE>**

1. dimensionality of original data space: **50**
2. dimensionality of reduced target space: **2**
3. perplexity: **30.0**
4. iterations: **1000** epochs

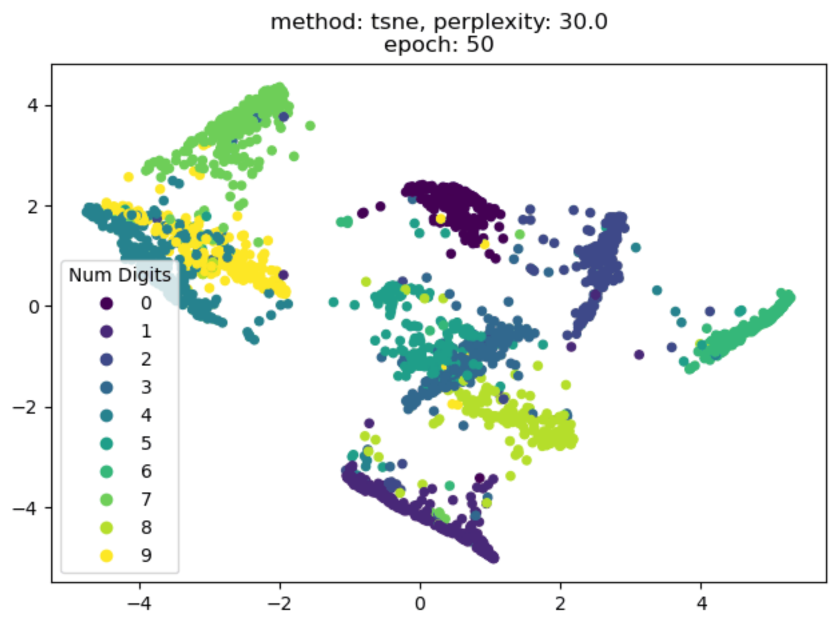
Other parameters are left as default values as in the original **tsne.py** (http://tsne.py).

The animation of optimizing procedures in **symmetric SNE**:



file name: `output/ssne/animation_perp30.0.gif`

The animation of optimizing procedures in **t-SNE**:



file name: `output/tsne/animation_perp30.0.gif`

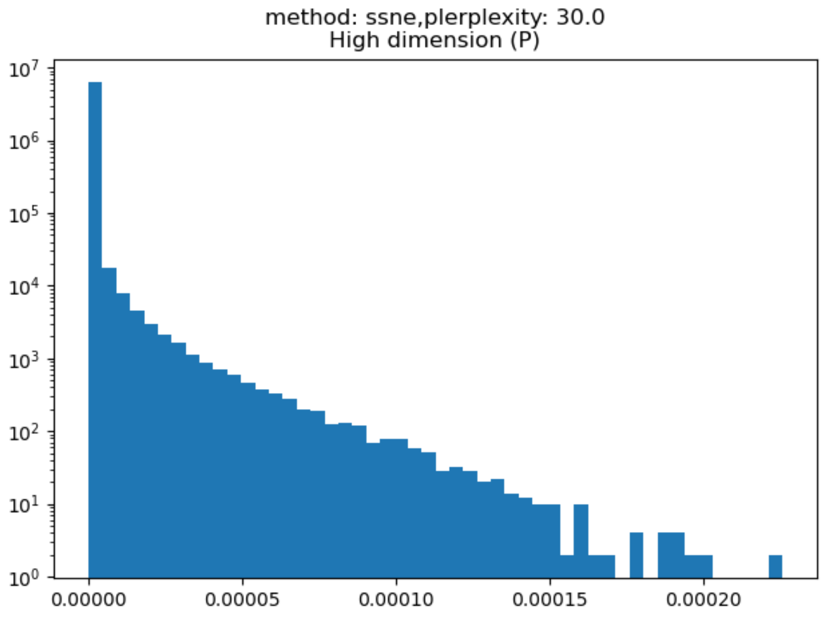
Part 3

Experimental Settings
<symmetric SNE>, **<t-SNE>**

1. dimensionality of original data space: **50**
2. dimensionality of reduced target space: **2**
3. perplexity: **30.0**
4. iterations: **1000** epochs

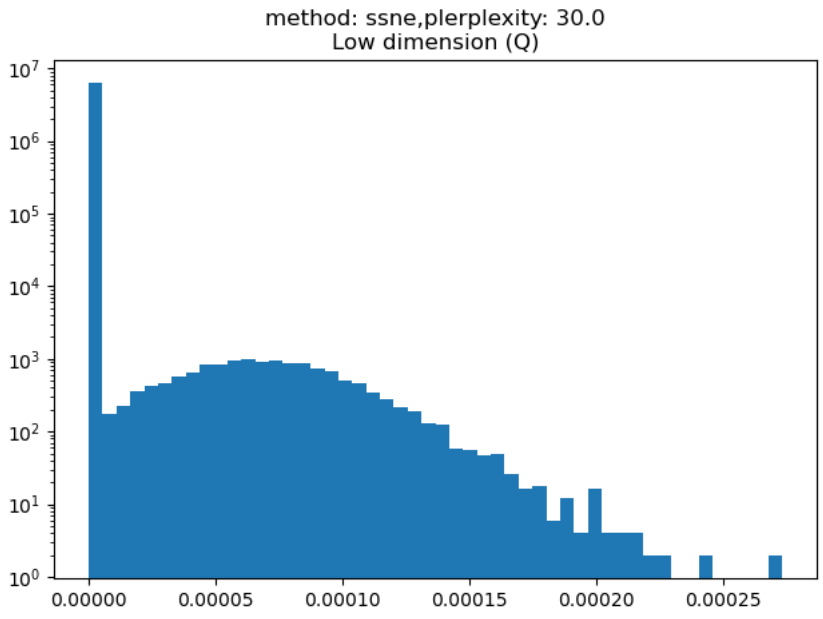
Other parameters are left as default values as in the original **tsne.py** (http://tsne.py).
x-axis: the probability(pairwise similarity)
y-axis: the frequency count(***in logarithm scale***)

<**symmetric SNE**>
The pairwise similarity distribution in high-dimensional space, P.



file name: `output/ssne/P_perp30.0.png`

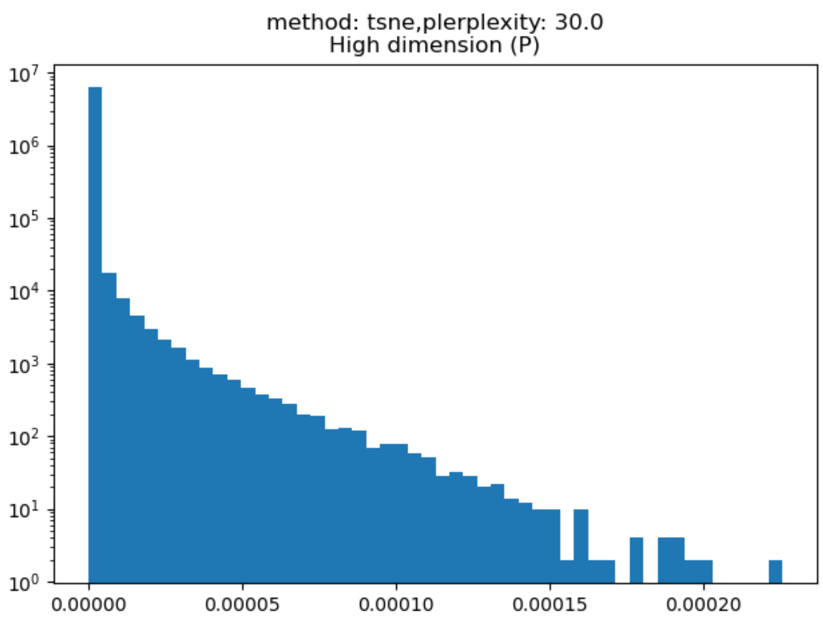
The pairwise similarity distribution in low-dimensional space, Q.



file name: `output/ssne/Q_perp30.0.png`

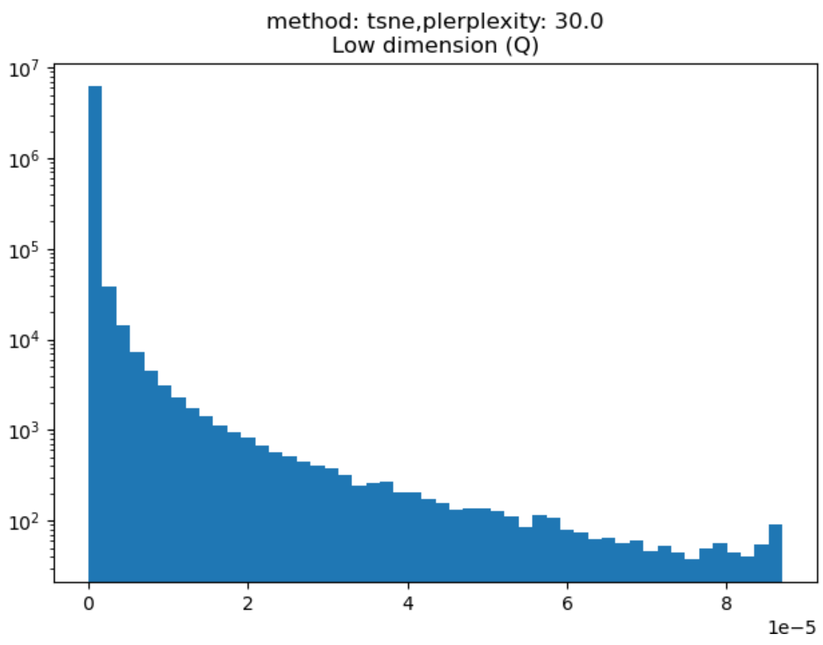
<**t-SNE**>

The pairwise similarity distribution in high-dimensional space, P.



file name: `output/tsne/P_perp30.0.png`

The pairwise similarity distribution in low-dimensional space, Q.



file name: `output/tsne/Q_perp30.0.png`

Part 4

⚠️ **The gif animations and some markdown notations just won't work on pdf file! So please *refer to my hackmd or gif files in the zip file handed in to check the animations!***

my hackmd:

https://hackmd.io/@witty27818a/r1Ufpx-u9

(https://hackmd.io/@witty27818a/r1Ufpx-u9)

Experimental Settings

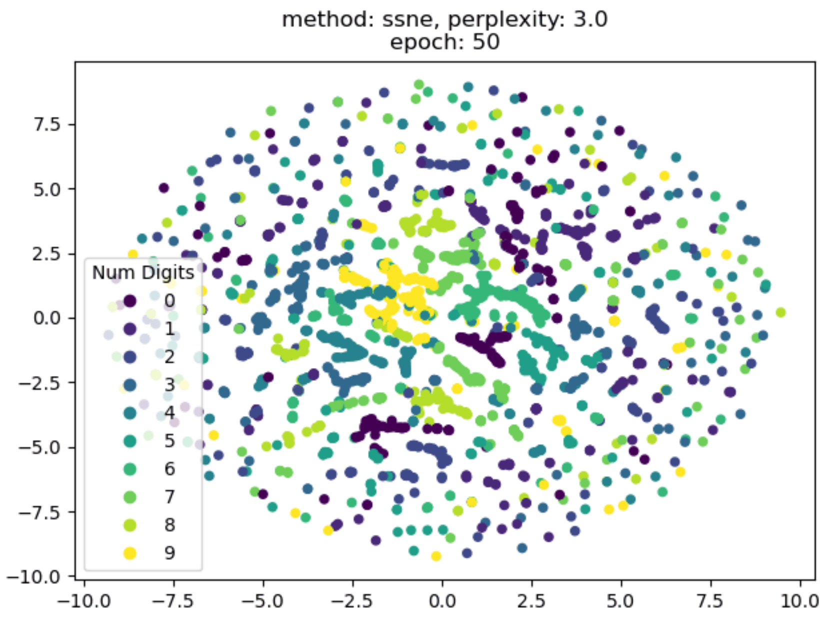
<symmetric SNE>, **<t-SNE>**

1. dimensionality of original data space: **50**
2. dimensionality of reduced target space: **2**
3. perplexity: **[3.0, 30.0, 300.0]**
4. iterations: **1000** epochs

Other parameters are left as default values as in the original **tsne.py** (http://tsne.py).

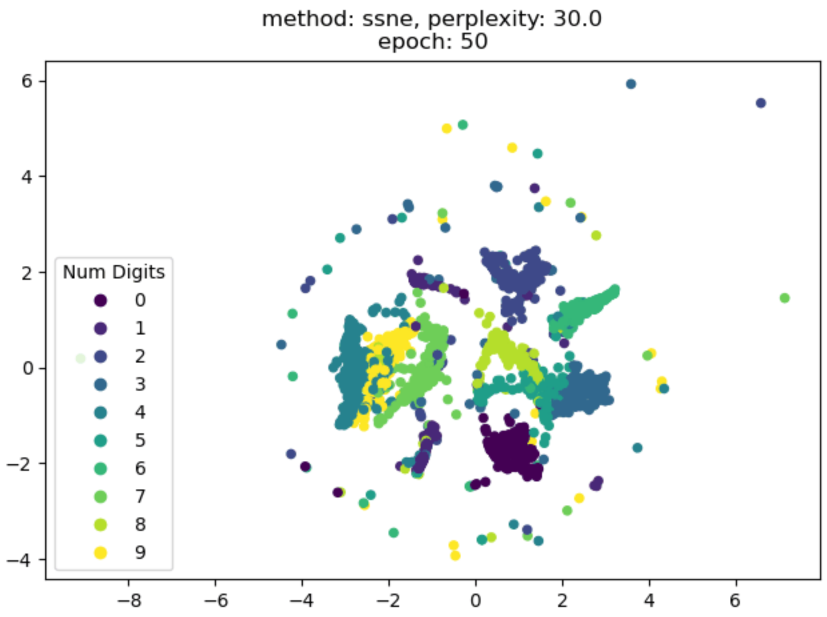
<symmetric SNE>

1. $perplexity = 3$



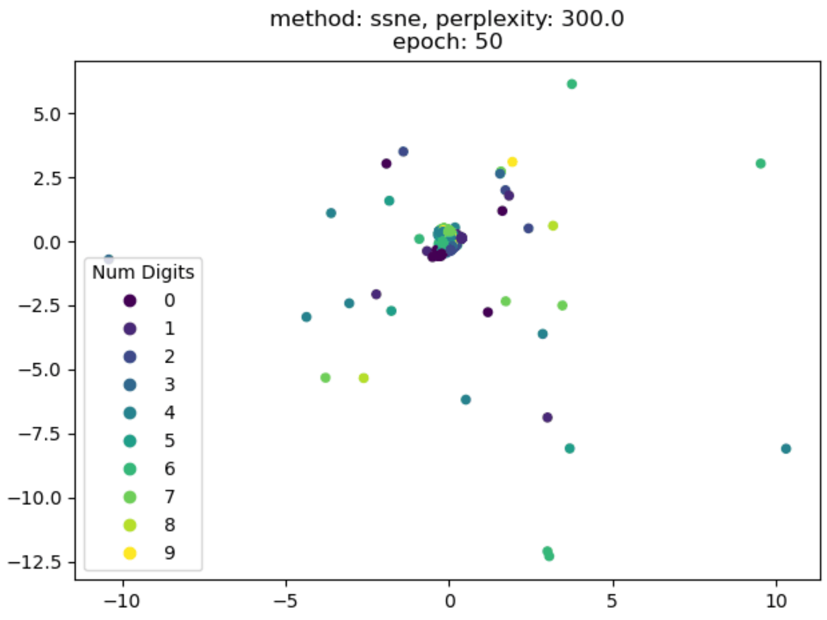
file name: `output/ssne/animation_perp3.0.gif`

2. $perplexity = 30$



file name: `output/ssne/animation_perp30.0.gif`

3. $perplexity = 300$

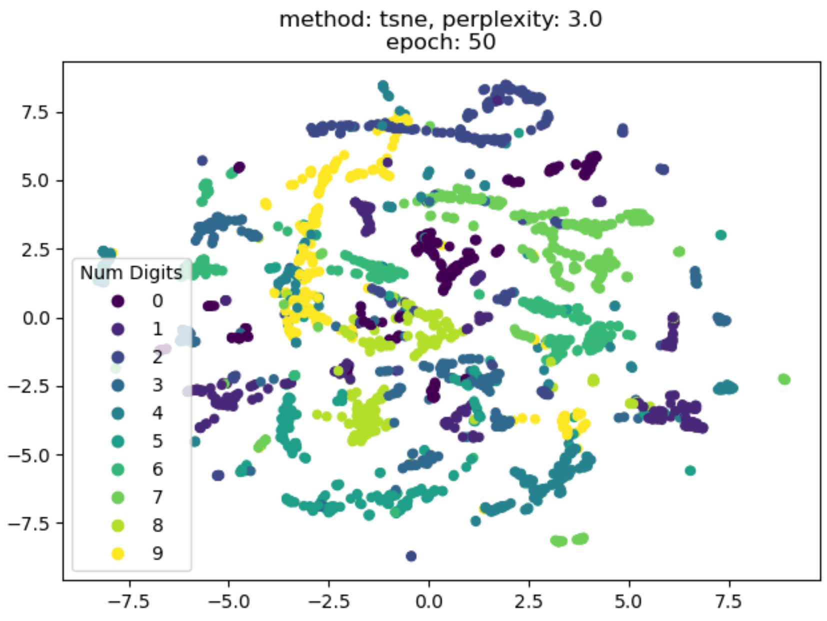


file name: `output/ssne/animation_perp300.0.gif`

No matter what perplexity we chose, it is inevitable that crowded problem will happen when ***symmetric SNE*** method is used. Different perplexity values just affect at **the speed of occurence** of this problem.

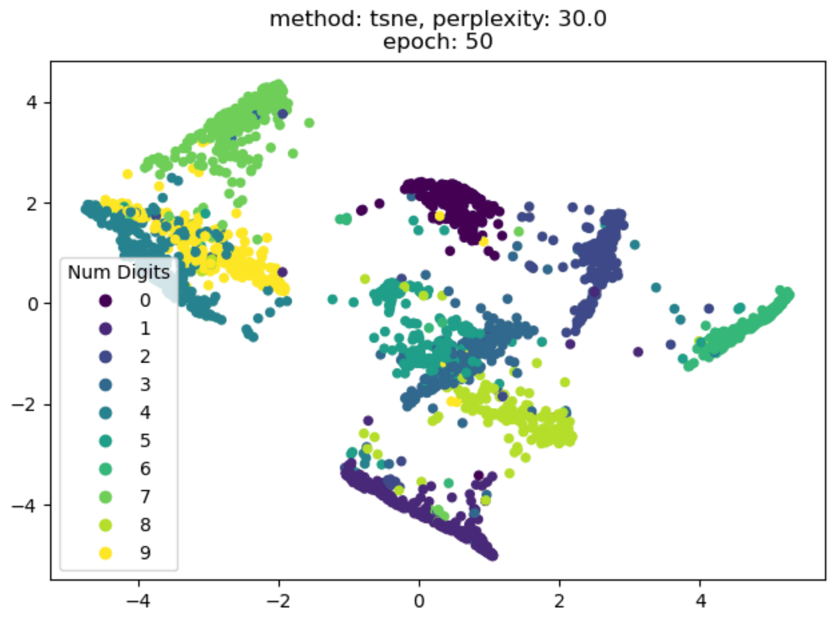
<**t-SNE**>

1. $perplexity = 3$



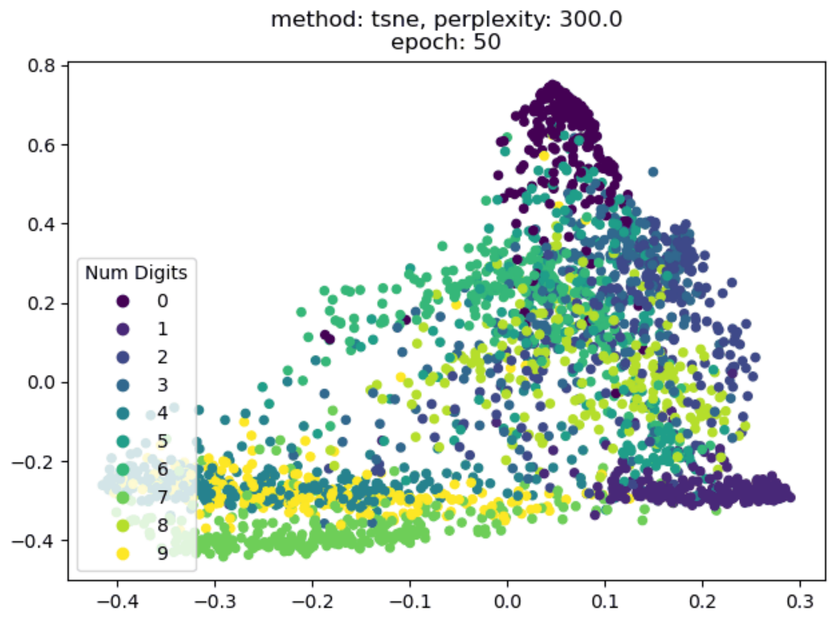
file name: `output/tsne/animation_perp3.0.gif`

2. $perplexity = 30$



file name: `output/tsne/animation_perp30.0.gif`

3. $perplexity = 300$

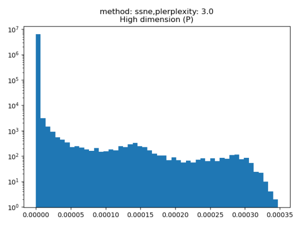
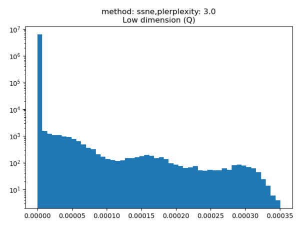
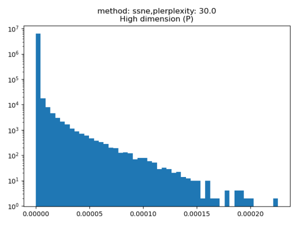
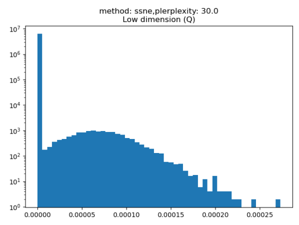
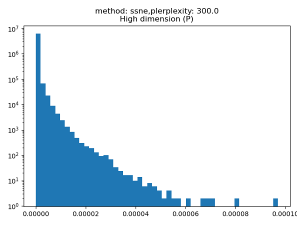
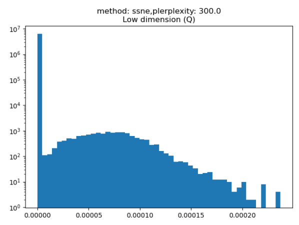


file name: `output/tsne/animation_perp300.0.gif`

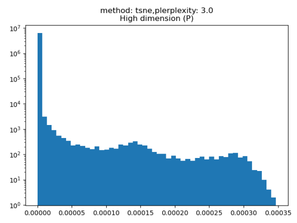
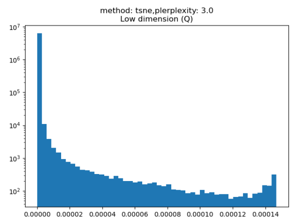
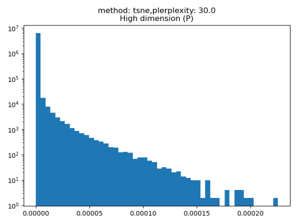
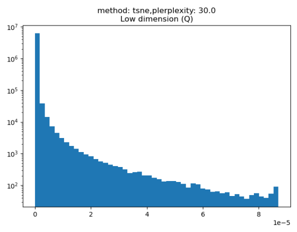
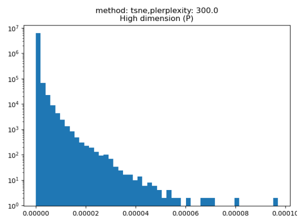
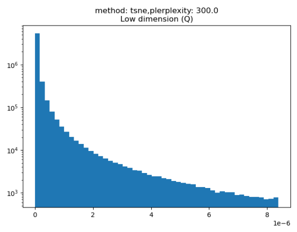
One can see that by adjusting the **perplexity values**, we can control the attention balance put on local struture or on global structure. We can see that with ***small*** perplexity value in **t-SNE**, data points within each cluster are not that "tight". If we tune up the perplexity, those data points in the same cluster are more likely to be closely arranged in the 2D embeddings figure.

However, with the perplexity set too high, it might become **difficult** for **t-SNE** to **classify different clusters**. One can see that the embeddings result of perplexity = 300 are worse than that of perplexity = 30.

Additionally, let's look at the distributions of pairwise similarities in both high-dimensional space and low-dimensional space, namely P and Q.
We will check both **symmetric SNE** and **t-SNE** method.
<Symmetric SNE>

	high-dim P	**low-dim Q**
perplexity=3		
perplexity=30		
perplexity=300		

In **symmetric SNE**, we can see that with **low perplexity**, pairwise similarities are in average **larger**. However, interestingly in **t-SNE** below, even with **low perplexity**, the pairwise similarities in **low-dimensional space** looks quite the same shape as those of bigger perplexity.
This exhibits that using ***t-distribution*** in low-dimensional space does help give out more robust similarities.
Also, one can see that in **symmetric SNE** with **higher perplexity**, the histograms kinda look like little hills, indicating that the similarities are in average bigger than those in **t-SNE** in low-dimensional space. This is also a sign of ***crowded problem***.
<t-SNE>

	high-dim P	**low-dim Q**
perplexity=3		
perplexity=30		
perplexity=300		

Observations & Discussion

Kernel Eigenfaces

In this section, we would like to tune the hyper-parameters of those kernels used in **kernel PCA** and **kernel LDA**. We will directly try on the best combinations from the results listed in ***part 3***, ***kernel eigenfaces*** section above.

1. In **kernel PCA**, the best combination was **polynomial kernel** with the number of nearest neighbors being $K = 5, 11$. We will try out $K = 5$ since 11 seems a little too many.
   (We don't consider linear kernel here, since it does not have tunable hyper-parameter.)
2. In **kernel LDA**, the best combination was **RBF kernel** with the number of nearest neighbors being $K = 1, 7$. We will try out $K = 7$ since 1 seems a little too few.

The default settings are as follows:

1. **polynomial kernel**: $\gamma = 5, c_0 = 10, d = 2$
2. **RBF kernel**: $\gamma = 10^{-7}$

The candidate values of these parameters to be experimented:

1. **polynomial kernel**
   1.1 $\gamma = \frac{1}{5}, 1, 5$
   1.2 $c_0 = 0, 10$
   1.3 $d = 2, 3, 4$
2. **RBF kernel**
   2.1 $\gamma = 10^{-10}, 10^{-7}, 10^{-4}$

The results are:

<kernel PCA>

$c_0 = 0$	$\gamma = \frac{1}{5}$	$\gamma = 1$	$\gamma = 5$
$d = 2$	0.867 (26/30)	0.867 (26/30)	0.867 (26/30)
$d = 3$	0.833 (25/30)	0.833 (25/30)	0.833 (25/30)
$d = 4$	0.800 (24/30)	0.800 (24/30)	0.800 (24/30)

$c_0 = 10$	$\gamma = \frac{1}{5}$	$\gamma = 1$	$\gamma = 5$
$d = 2$	0.867 (26/30)	0.867 (26/30)	0.867 (26/30)
$d = 3$	0.833 (25/30)	0.833 (25/30)	0.833 (25/30)
$d = 4$	0.800 (24/30)	0.800 (24/30)	0.800 (24/30)

file name:

`output/observation_and_discussion/kernel_PCA`

One can see from the results that in **kernel PCA**, the main hyper-parameter affecting the accuracy is the degree d. As for coefficient c_0 and gamma γ, they don't affect that much. Recall the formula:

$$k(\vec{x}_i, \vec{x}_j) = (\gamma \vec{x}_i^T \vec{x}_j + c_0)^d$$

This might be due to the fact that we don't need too complicated polynomial to fit the kernels.

<kernel LDA>

γ	10^{-10}	10^{-7}	10^{-3}
	0.733 (22/30)	0.833 (25/30)	0.033 (1/30)

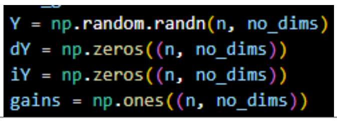
file name:

`output/observation_and_discussion/kernel_LDA`

We can observe γ does affect the accuracy **a lot**. The γ needs to be very small, since we can see that at $\gamma = 10^{-3}$, the accuracy is only $\frac{1}{30} = 0.033$, much worse than randomly guessing.
The default value $\gamma = 10^{-7}$ still yield the best accuracy $\frac{25}{30} = 0.833$. Smaller γ values may drop the performance, but not that much though.

Some properties of ***t-SNE***

1. **Randomness**: t-SNE has randomness in its algorithm while other dimensionalty reduction methods such as **PCA** are **deterministic**, meaning that the results of such methods are always the same (fixed). In **t-SNE**, the embeddings matrix Y is randomly generated at first.

> This screenshot is a snippet of the function **tsne()**
>
> 
>
> ```
> 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))
> ```

2. **Careful with explainability**: Since t-SNE will adjust the size of a cluster based on the crowded degree of its neighborhood, it's meaningless to compare the sizes of 2 clusters. Also, it would be inappropriate to compare the distance between 2 clusters, since t-SNE is based on probability, not distance. Finally, t-SNE cannot be used to detect *outliers*, since it will pull the data points in some cluster.
3. **Intrinsic dimensionality**: Sometimes, we will find that the visualization of 2D embeddings via t-SNE still look kinda messy. Nevertheless, t-SNE might not be to blame for. Bad visualization embeddings might be resulted from **high intrinsic dimensionality** of these data themselves.

Barnes-Hut t-SNE
This is a tree-based t-SNE, which mainly focuses on optimizing the speed of t-SNE.

> The time complexity went from $O(n^2)$ to $O(n \log n)$.

The following is some specialties of this kind of t-SNE:

1. Additional parameter which is related to **angle**, controling the optimization results.
2. Mainly focus on **data visualization**. Thus, it only works when the dimensionality of target space is either **2** or **3**.
3. Does not work well on **sparse matrix**. Need to apply some special embedding technique or projection approximation.