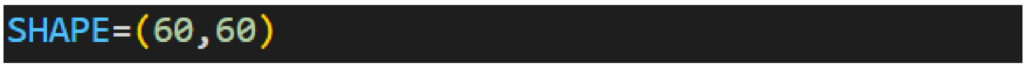
Code: Kernel Eigenface

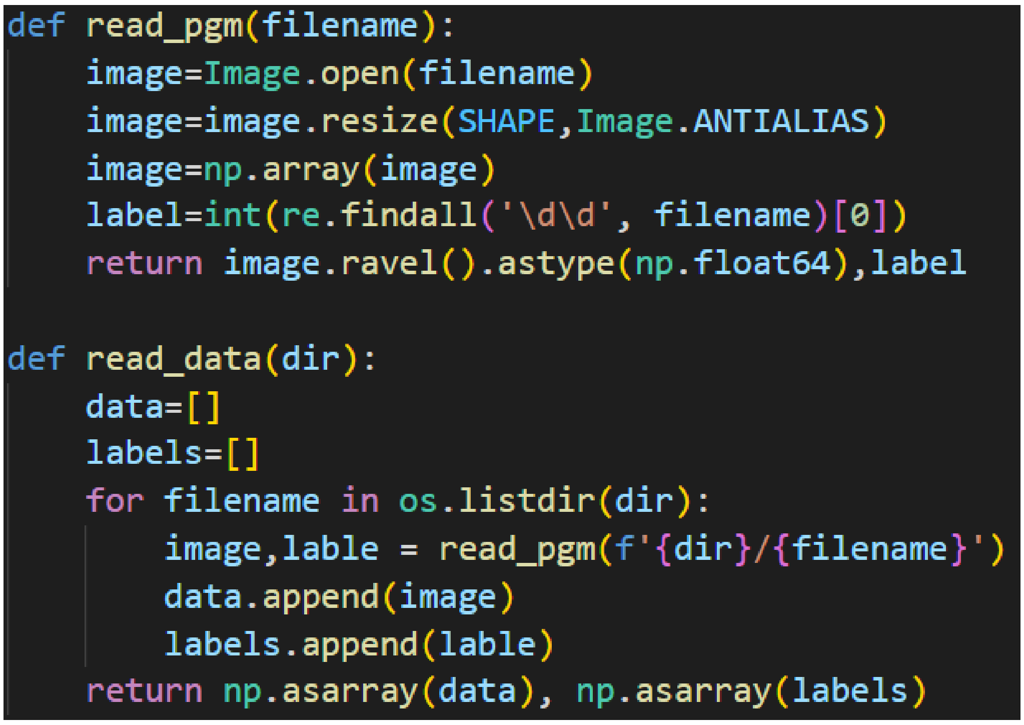
Part1:

1. **load data**

   Using *PIL.Image* to open the .pgm file, and use *re.findall()* to find each image's label from filename. Due to the cost of computing the matrix's inverse, I resize the size of image to 60*60.



```python
SHAPE=(60,60)
```

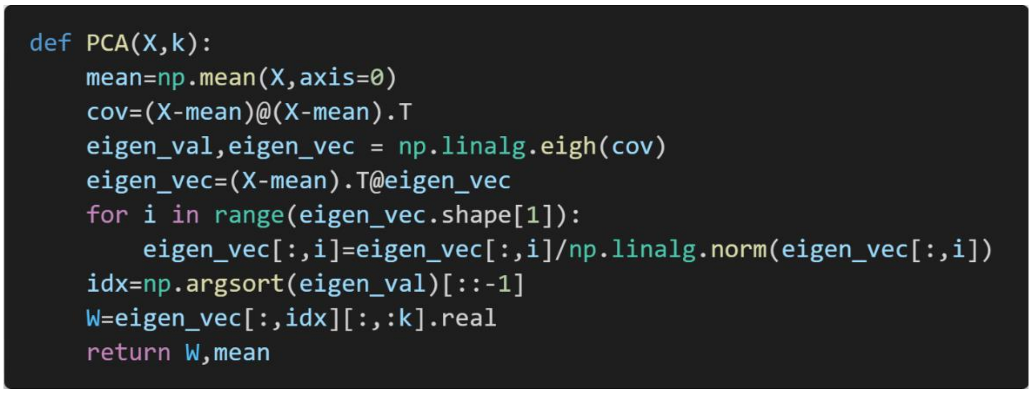


```python
def read_pgm(filename):
    image=Image.open(filename)
    image=image.resize(SHAPE,Image.ANTIALIAS)
    image=np.array(image)
    label=int(re.findall('\d\d', filename)[0])
    return image.ravel().astype(np.float64),label

def read_data(dir):
    data=[]
    labels=[]
    for filename in os.listdir(dir):
        image,lable = read_pgm(f'{dir}/{filename}')
        data.append(image)
        labels.append(lable)
    return np.asarray(data), np.asarray(labels)
```

2. **PCA**

   Compute the covariance matrix of X, and using *np.linalg.eigh()* to solve the eigenvalue problem, then sort the eigenvalue in descending order, and get the orthogonal projection matrix W which is composed of k first largest eigenvectors of covariance matrix of X.
   Also return the mean of X due to the reconstruction.



```python
def PCA(X,k):
    mean=np.mean(X,axis=0)
    cov=(X-mean)@(X-mean).T
    eigen_val,eigen_vec = np.linalg.eigh(cov)
    eigen_vec=(X-mean).T@eigen_vec
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:,i]=eigen_vec[:,i]/np.linalg.norm(eigen_vec[:,i])
    idx=np.argsort(eigen_val)[::-1]
    W=eigen_vec[:,idx][:,:k].real
    return W,mean
```

3. PDA

By reference, solve $S_W{}^{-1}S_B v_i = \lambda_i v_i$ and take k first largest

eigenvectors then we can obtain the orthogonal projection matrix W, which is given by:

$$W^{\Phi}_{OPT} = \arg\max_{W^{\Phi}} \frac{|(W^{\Phi})^T S^{\Phi}_B W^{\Phi}|}{|(W^{\Phi})^T S^{\Phi}_W W^{\Phi}|}$$

And the between-class and within-class scatter matric S_B and S_W are defined by:

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

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

The first for loop is computing S_B and S_W, and the remaining code is like PCA part, solve the eigenvalue problem and take k first largest eigenvector.

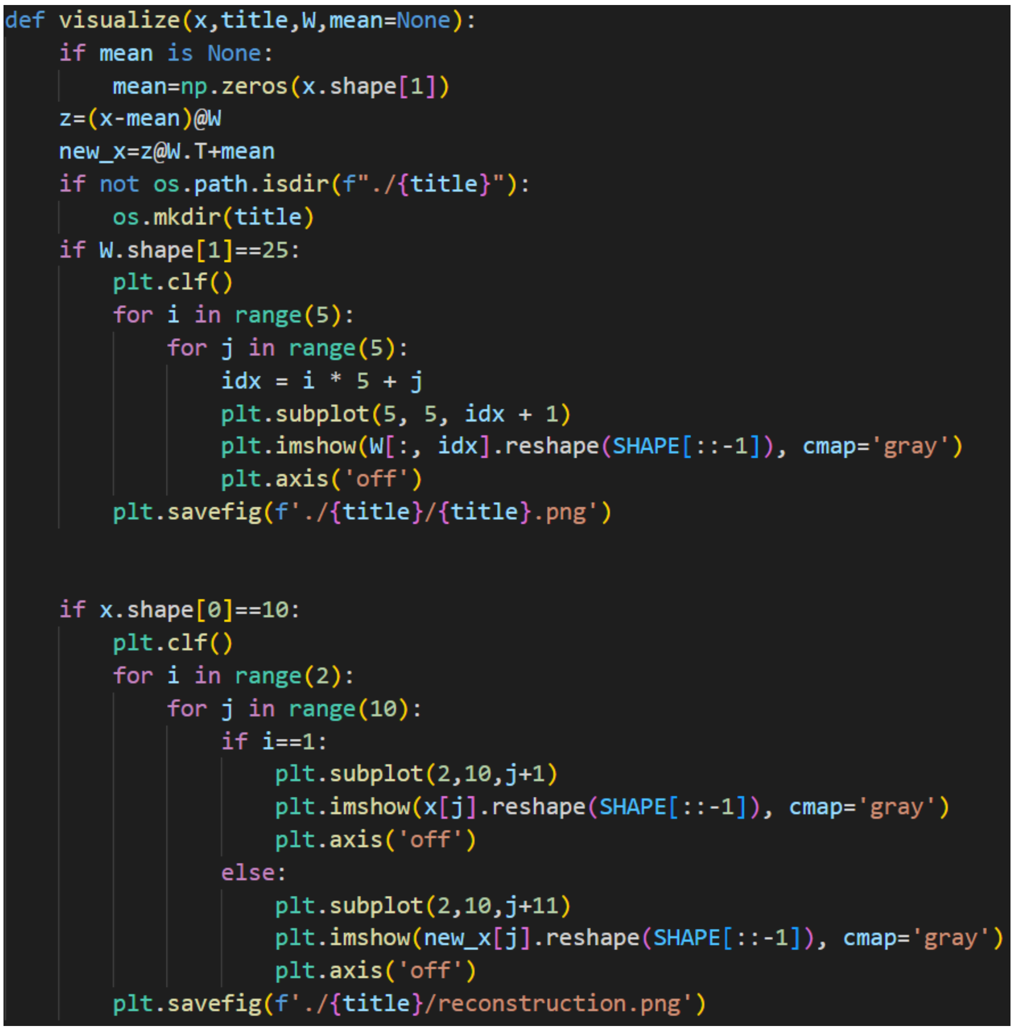
```
def LDA(X,labels,k):
    d=X.shape[1]
    labels=np.asarray(labels)
    c=np.unique(labels)
    mean=np.mean(X, axis=0)
    S_w=np.zeros((d, d))
    S_b=np.zeros((d, d))
    for i in c:
        X_i=X[np.where(labels==i)[0],:]
        mean_i=np.mean(X_i, axis=0)
        S_w+=(X_i-mean_i).T@(X_i-mean_i)
        S_b+=X_i.shape[0]*((mean_i - mean).T @ (mean_i - mean))
    eigen_val,eigen_vec=np.linalg.eig(np.linalg.pinv(S_w)@S_b)
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i]/np.linalg.norm(eigen_vec
[:, i])
    idx=np.argsort(eigen_val)[::-1]
    W=eigen_vec[:,idx][:,:k].real
    return W
```

4. **Visualize**

For visualize eigenfaces (or fisherfaces), resize the eigenvectors to 60*60 array and use *matplotlob* to plot the result. For reconstruct images, project z back to the original space by:

$$x_{reconstruct} = Wz + \mu \quad \text{where} \quad z = W^T(x - \mu)$$

And then plot 10 origin images in the first row, 10 reconstruct images in the second row.

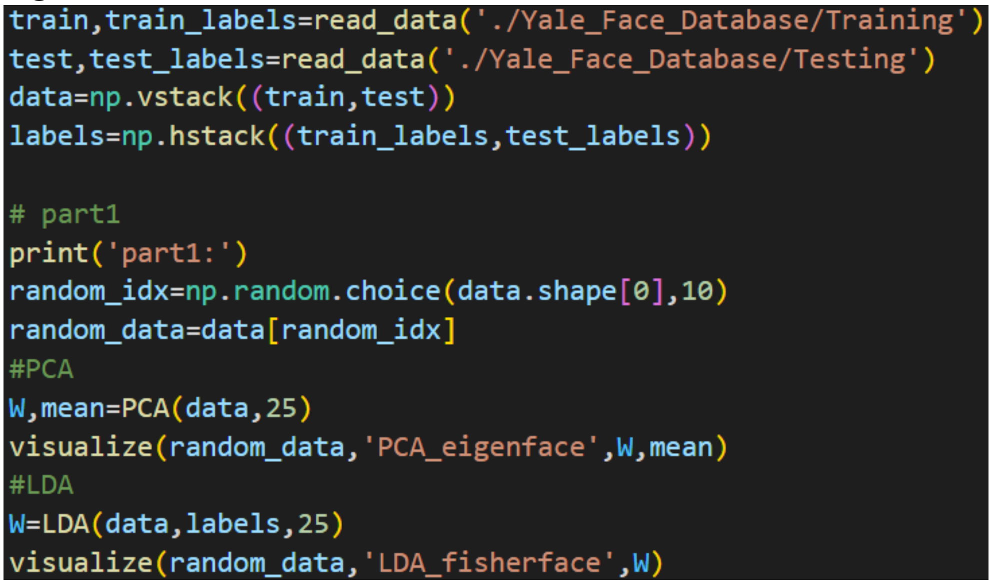


```python
def visualize(x,title,W,mean=None):
    if mean is None:
        mean=np.zeros(x.shape[1])
    z=(x-mean)@W
    new_x=z@W.T+mean
    if not os.path.isdir(f"./{title}"):
        os.mkdir(title)
    if W.shape[1]==25:
        plt.clf()
        for i in range(5):
            for j in range(5):
                idx = i * 5 + j
                plt.subplot(5, 5, idx + 1)
                plt.imshow(W[:, idx].reshape(SHAPE[::-1]), cmap='gray')
                plt.axis('off')
        plt.savefig(f'./{title}/{title}.png')


    if x.shape[0]==10:
        plt.clf()
        for i in range(2):
            for j in range(10):
                if i==1:
                    plt.subplot(2,10,j+1)
                    plt.imshow(x[j].reshape(SHAPE[::-1]), cmap='gray')
                    plt.axis('off')
                else:
                    plt.subplot(2,10,j+11)
                    plt.imshow(new_x[j].reshape(SHAPE[::-1]), cmap='gray')
                    plt.axis('off')
        plt.savefig(f'./{title}/reconstruction.png')
```

5. **Main**

   Random select 10 images first, and do PCA and LDA with the first 25 eigenfaces and fisherfaces.

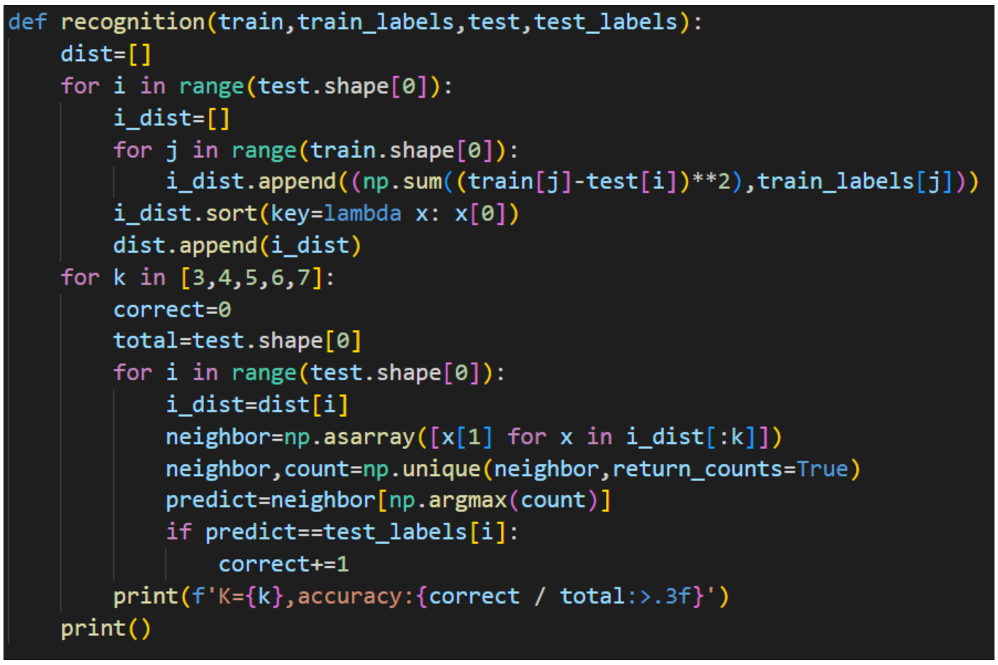


```python
train,train_labels=read_data('./Yale_Face_Database/Training')
test,test_labels=read_data('./Yale_Face_Database/Testing')
data=np.vstack((train,test))
labels=np.hstack((train_labels,test_labels))

# part1
print('part1:')
random_idx=np.random.choice(data.shape[0],10)
random_data=data[random_idx]
#PCA
W,mean=PCA(data,25)
visualize(random_data,'PCA_eigenface',W,mean)
#LDA
W=LDA(data,labels,25)
visualize(random_data,'LDA_fisherface',W)
```

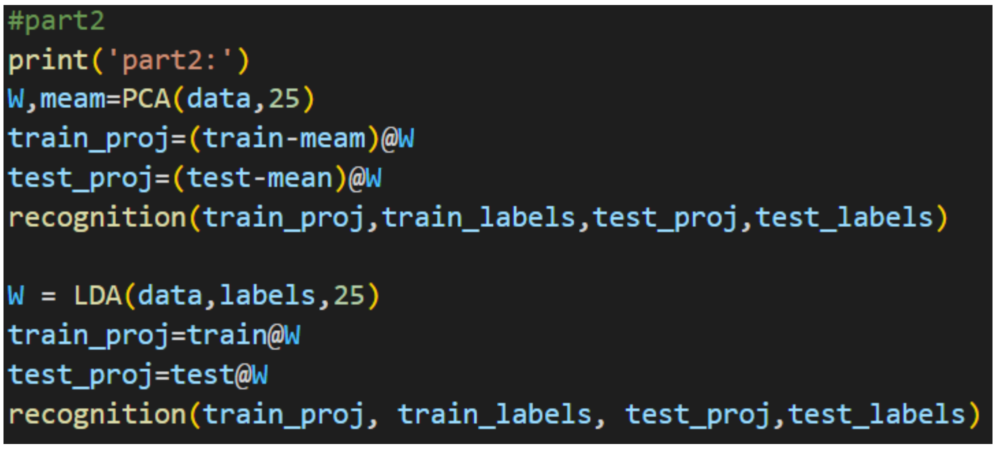
Part2:

1. **Face recognition**
   Compute the distance matric between training and testing data. And use kNN algo, take k smallest distance for each testing data, and count the largest number of repeat class to find out the predict class, in this assignment I try k=3,4,5,6,7.



```python
def recognition(train,train_labels,test,test_labels):
    dist=[]
    for i in range(test.shape[0]):
        i_dist=[]
        for j in range(train.shape[0]):
            i_dist.append((np.sum((train[j]-test[i])**2),train_labels[j]))
        i_dist.sort(key=lambda x: x[0])
        dist.append(i_dist)
    for k in [3,4,5,6,7]:
        correct=0
        total=test.shape[0]
        for i in range(test.shape[0]):
            i_dist=dist[i]
            neighbor=np.asarray([x[1] for x in i_dist[:k]])
            neighbor,count=np.unique(neighbor,return_counts=True)
            predict=neighbor[np.argmax(count)]
            if predict==test_labels[i]:
                correct+=1
        print(f'K={k},accuracy:{correct / total:>.3f}')
    print()
```

2. **Main**
   The face recognition is using the data after projecting to lower dim. Space. So, the parameter of train and test data should be projected first.



```python
#part2
print('part2:')
W,meam=PCA(data,25)
train_proj=(train-meam)@W
test_proj=(test-mean)@W
recognition(train_proj,train_labels,test_proj,test_labels)

W = LDA(data,labels,25)
train_proj=train@W
test_proj=test@W
recognition(train_proj, train_labels, test_proj,test_labels)
```

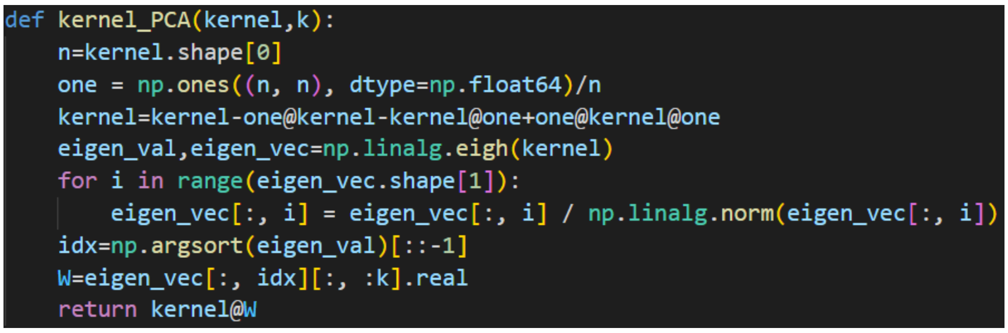
Part3:

1. **kernel PCA**
   Instead of centralizing the data, kernel PCA need to centralize the kernel, which is given by:

$$K^C = K - \mathbf{1}_N K - K \mathbf{1}_N + \mathbf{1}_N K \mathbf{1}_N$$

$\mathbf{1}_N$ is NxN matrix with every element $1/N$

And then solve the eigenvalue problem…like PCA part.



```
def kernel_PCA(kernel,k):
    n=kernel.shape[0]
    one = np.ones((n, n), dtype=np.float64)/n
    kernel=kernel-one@kernel-kernel@one+one@kernel@one
    eigen_val,eigen_vec=np.linalg.eigh(kernel)
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
    idx=np.argsort(eigen_val)[::-1]
    W=eigen_vec[:, idx][:, :k].real
    return kernel@W
```

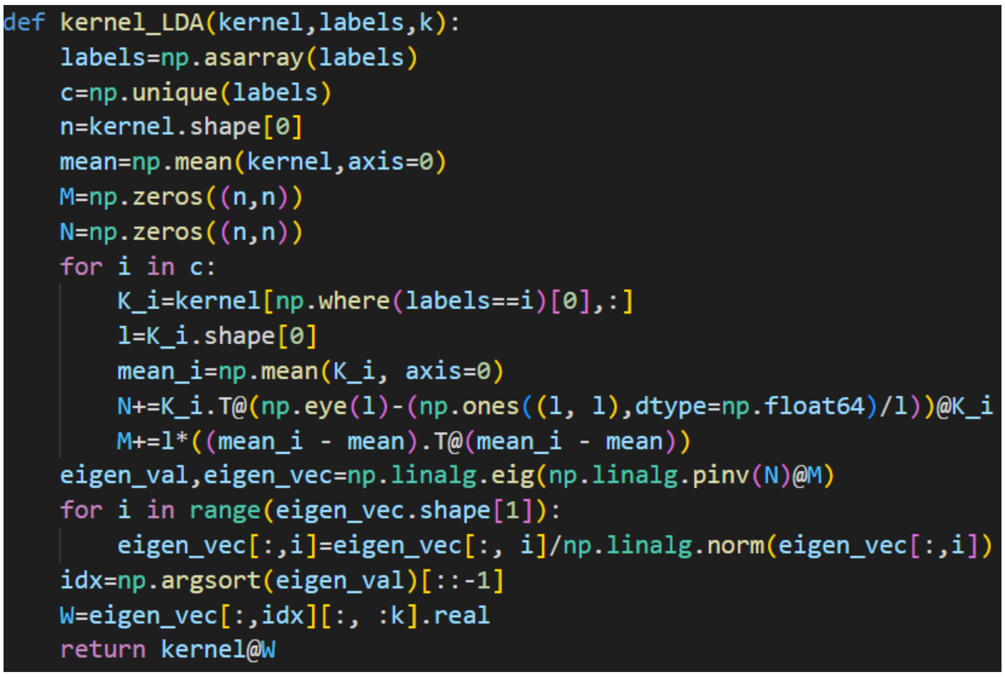
2. kernel LDA

Compute within-classes scatter N and between-classes scatter M by:

$$N = \sum_{j=1}^{c} \mathbf{K}_j (\mathbf{I} - \mathbf{1}_{l_j}) \mathbf{K}_j^{\mathrm{T}}.$$

$$M = \sum_{j=1}^{c} l_j (\mathbf{M}_j - \mathbf{M}_*)(\mathbf{M}_j - \mathbf{M}_*)^{\mathrm{T}}$$

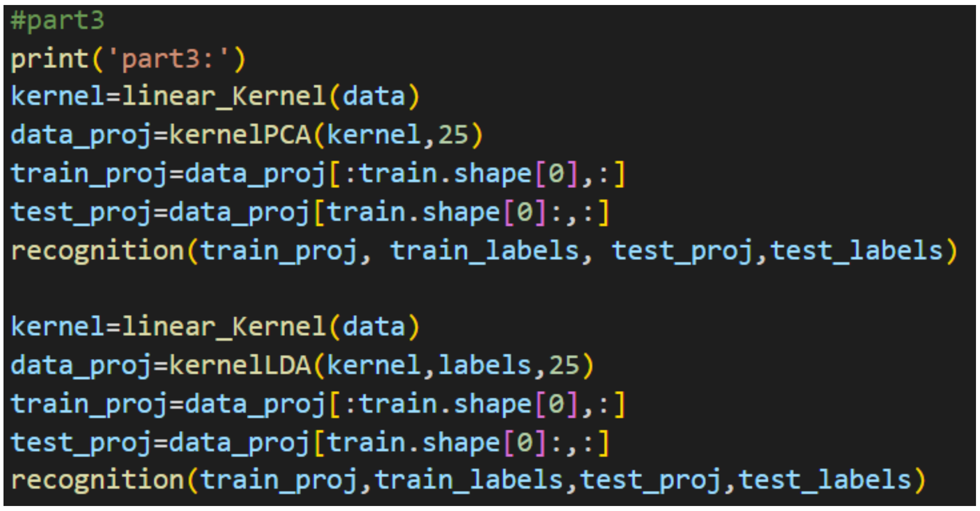
And solve the eigenvalue problem of $N^{-1}M$, and take k first largest eigenvector to get projection matrix W.



```
def kernel_LDA(kernel,labels,k):
    labels=np.asarray(labels)
    c=np.unique(labels)
    n=kernel.shape[0]
    mean=np.mean(kernel,axis=0)
    M=np.zeros((n,n))
    N=np.zeros((n,n))
    for i in c:
        K_i=kernel[np.where(labels==i)[0],:]
        l=K_i.shape[0]
        mean_i=np.mean(K_i, axis=0)
        N+=K_i.T@(np.eye(l)-(np.ones((l, l),dtype=np.float64)/l))@K_i
        M+=l*((mean_i - mean).T@(mean_i - mean))
    eigen_val,eigen_vec=np.linalg.eig(np.linalg.pinv(N)@M)
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:,i]=eigen_vec[:, i]/np.linalg.norm(eigen_vec[:,i])
    idx=np.argsort(eigen_val)[::-1]
    W=eigen_vec[:,idx][:, :k].real
    return kernel@W
```

3. Main

Similar to part2, project the data first, and then do face recognition.



```
#part3
print('part3:')
kernel=linear_Kernel(data)
data_proj=kernelPCA(kernel,25)
train_proj=data_proj[:train.shape[0],:]
test_proj=data_proj[train.shape[0]:,:]
recognition(train_proj, train_labels, test_proj,test_labels)

kernel=linear_Kernel(data)
data_proj=kernelLDA(kernel,labels,25)
train_proj=data_proj[:train.shape[0],:]
test_proj=data_proj[train.shape[0]:,:]
recognition(train_proj,train_labels,test_proj,test_labels)
```

Code: SNE

Part1:

Since using Gaussian distribution may cause crowding problem in low-D, but student t-distribution can avoid this problem in low-D. So, in low-D we prefer to use t-SNE, but in high-D we still use symmetric-sne. And the difference between these two sne is how to compute the joint probability. Besides, the gradient will also change depend on how we compute the joint probability.

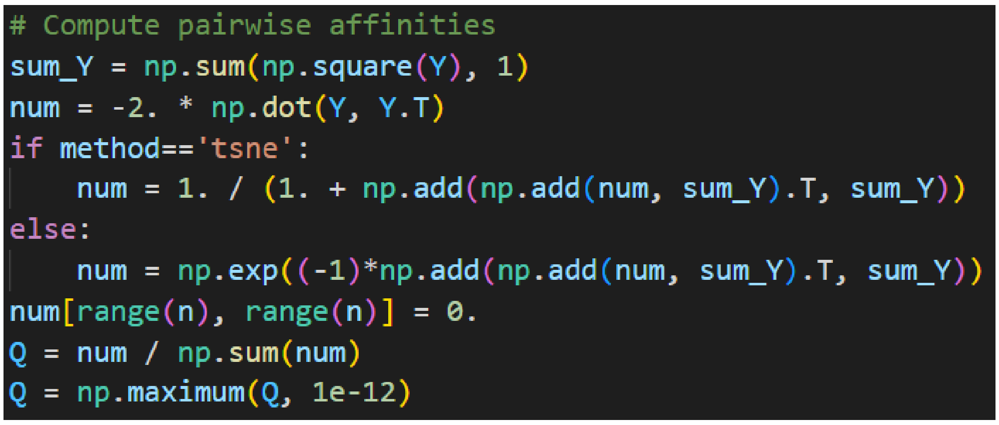
1. **modify the joint probability:**

   In t-sne, q_{ij} is given by:

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

   But in symmetric sne, q_{ij} should be:

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



```
# Compute pairwise affinities
sum_Y = np.sum(np.square(Y), 1)
num = -2. * np.dot(Y, Y.T)
if method=='tsne':
    num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
else:
    num = np.exp((-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)
```

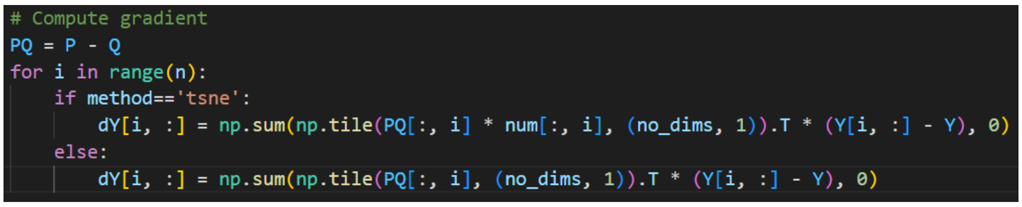
2. **modify gradient:**

   In t-sne:

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

   In symmetric sne:

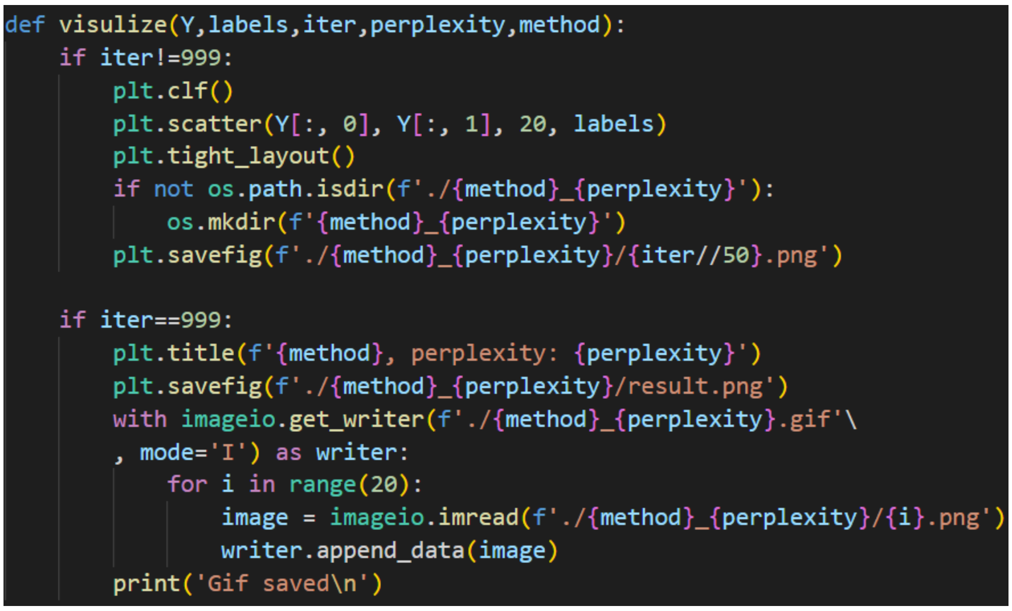
$$\frac{\partial C}{\partial y_i} = 2\sum_j (p_{ij} - q_{ij})(y_i - y_j)$$



```python
# Compute gradient
PQ = P - Q
for i in range(n):
    if method=='tsne':
        dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
    else:
        dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

Part2:

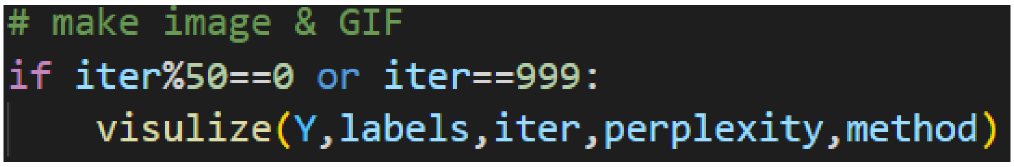
1. **visualize:**
   Save the image per 50 iterations during the optimize procedure.
   Make the GIF and save the result image in the last iteration (=999).



```python
def visulize(Y,labels,iter,perplexity,method):
    if iter!=999:
        plt.clf()
        plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
        plt.tight_layout()
        if not os.path.isdir(f'./{method}_{perplexity}'):
            os.mkdir(f'{method}_{perplexity}')
        plt.savefig(f'./{method}_{perplexity}/{iter//50}.png')

    if iter==999:
        plt.title(f'{method}, perplexity: {perplexity}')
        plt.savefig(f'./{method}_{perplexity}/result.png')
        with imageio.get_writer(f'./{method}_{perplexity}.gif'\
        , mode='I') as writer:
            for i in range(20):
                image = imageio.imread(f'./{method}_{perplexity}/{i}.png')
                writer.append_data(image)
        print('Gif saved\n')
```

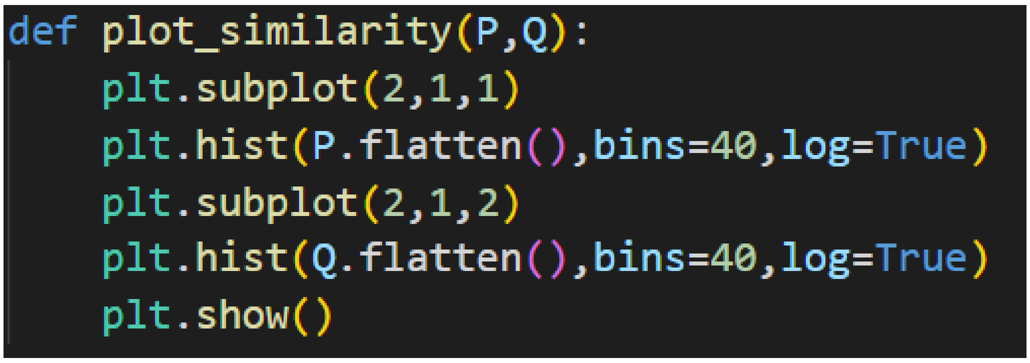
In sne(), add



```python
# make image & GIF
if iter%50==0 or iter==999:
    visulize(Y,labels,iter,perplexity,method)
```

Part3:

Plot the probability of high dimension(P) and low dimension (Q).



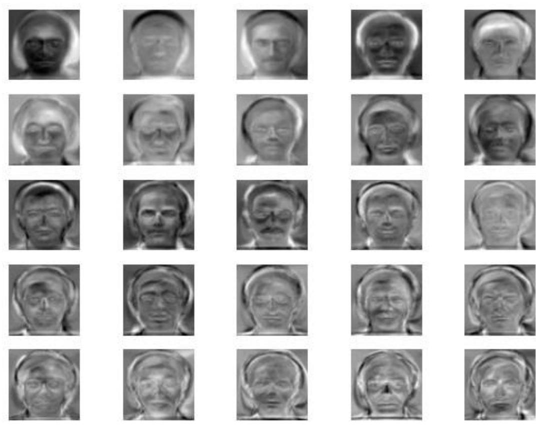
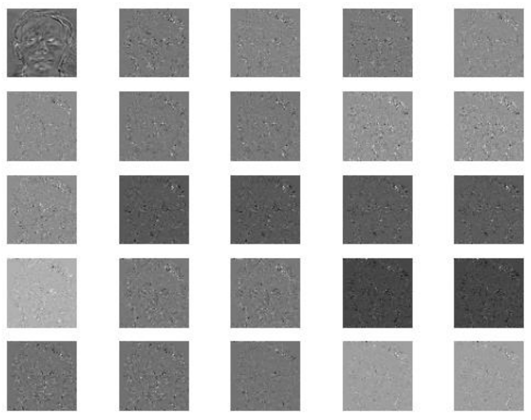
```python
def plot_similarity(P,Q):
    plt.subplot(2,1,1)
    plt.hist(P.flatten(),bins=40,log=True)
    plt.subplot(2,1,2)
    plt.hist(Q.flatten(),bins=40,log=True)
    plt.show()
```

Part4:

This task is implemented in part1 and part2, just change the perplexity.

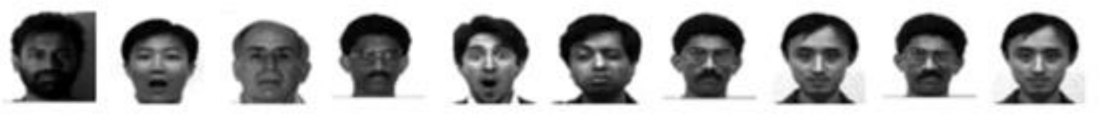
Experiment: Kernel Eigenface

Eigenfaces:

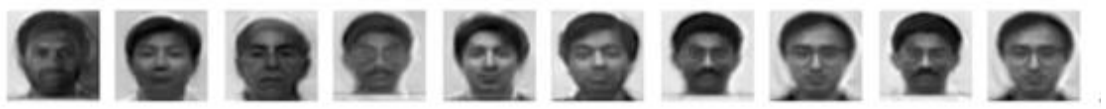
PCA	LDA
	

Reconstruction:

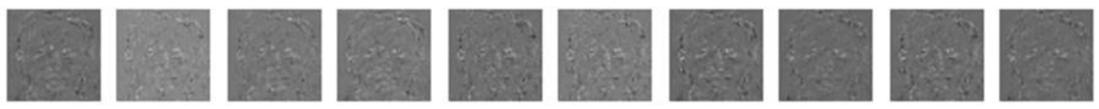
Origin



PCA Reconstruction

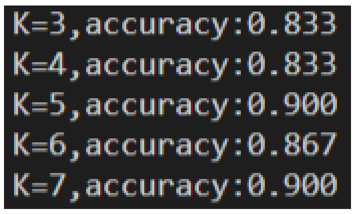
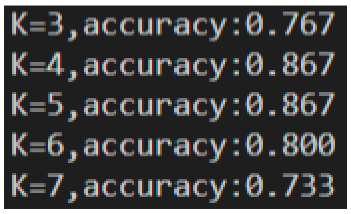


LDA Reconstruction



Face recognition:

PCA	LDA

 K=3,accuracy:0.833
K=4,accuracy:0.833
K=5,accuracy:0.900
K=6,accuracy:0.867
K=7,accuracy:0.900
	 K=3,accuracy:0.767
	K=4,accuracy:0.867
	K=5,accuracy:0.867
	K=6,accuracy:0.800
	K=7,accuracy:0.733

Kernel PCA:

Using k=5 in kNN algo., because it has better performance in previous part. And the parameter of each kernel is using grid search method to find the best performance.

Kernel type	linear	Polynomial (gamma=10^-6, constant=0, degree=3)	RBF (gamma=10^-9)
Accuracy	0.833	0.833	0.833

Kernel LDA:

Kernel type	linear	Polynomial (gamma=10, constant=0, degree=3)	RBF (gamma=10^-9)
Accuracy	0.700	0.733	0.767

Discussion:

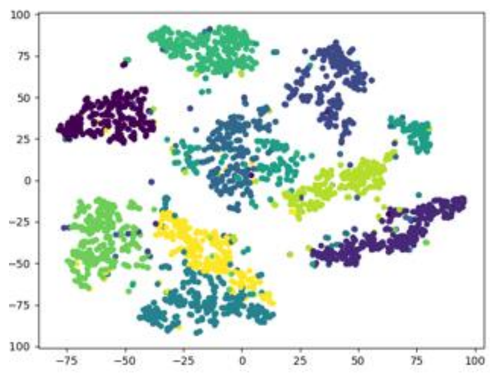
When I use grid search method, I find that different parameter of kernel has more effect on LDA than PCA. In LDA, sometimes it has bad performance but sometime the performance is good, it depends on the parameter of kernel. But in PCA, it seems always have good performance whatever the parameter is.
In addition, the polynomial and RBF kernel have better performance than linear kernel in average, it is the same result with HW5.
And the result of using kernel or not, in this assignment, the kernel PCA/LDA doesn't have better performance than not using kernel.

Experiment: SNE

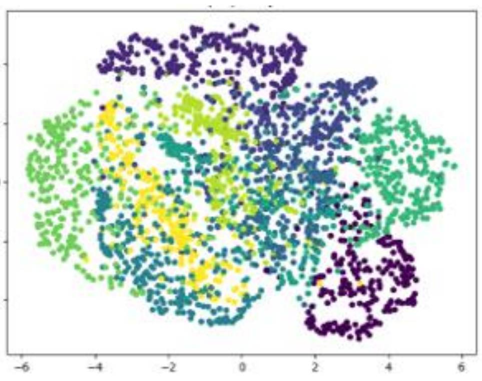
Visualize embedding:

Perplexity:20.0, no_dims=2, initial_dims=50

t-sne



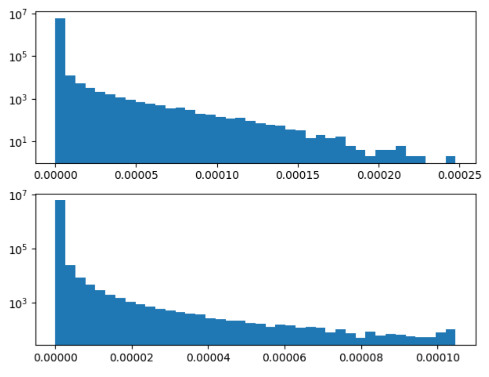
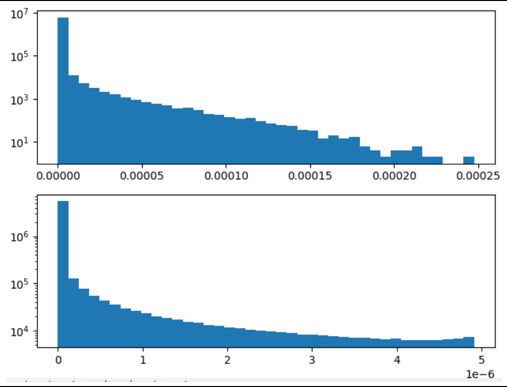
symmetric sne



From above images, we can observe that symmetric SNE has crowding problem, all data points lie in (-6,6) , and the boundary of each group is not clear. But in t-SNE, there are about 5 groups clearly separate from other groups, and the data lies in (-100,100), which use more space to separate from each other.

Visualize distribution of pairwise similarities:

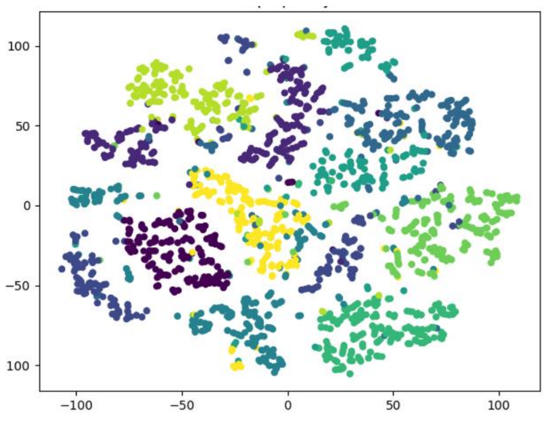
Perplexity:20.0, no_dims=2, initial_dims=50

t-sne	**Symmetric sne**
	

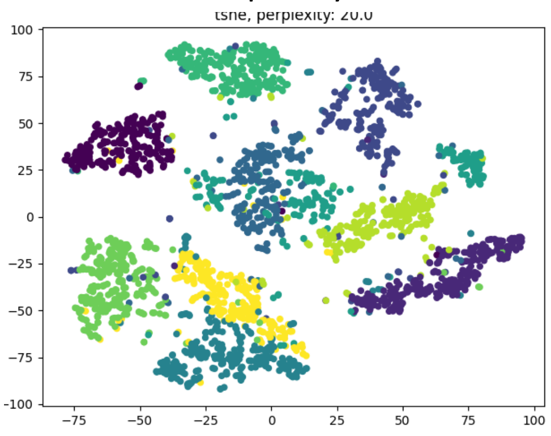
Different perplexity:

t-sne:

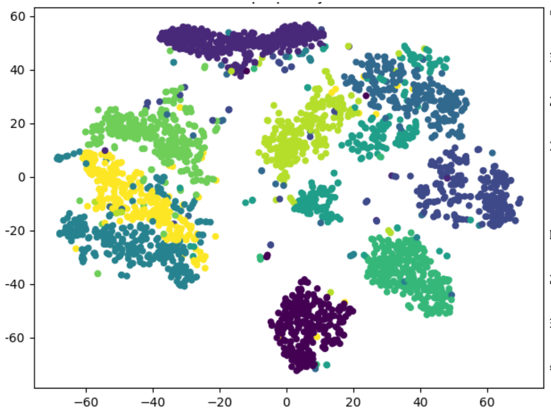
Perplexity=5



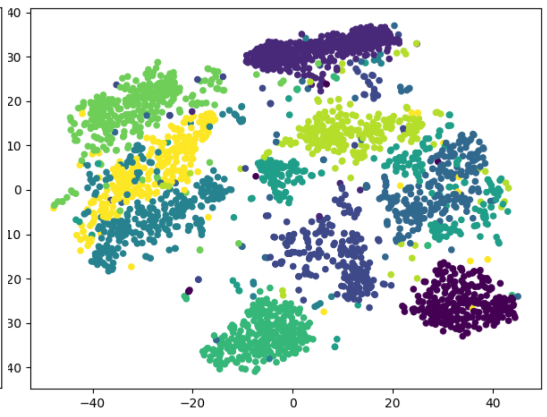
Perplexity=20



Perplexity=50

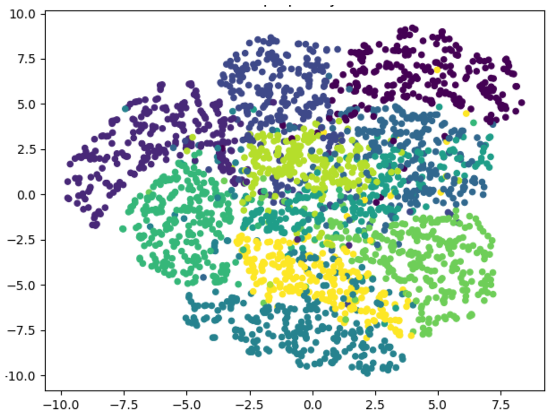


Perplexity=100

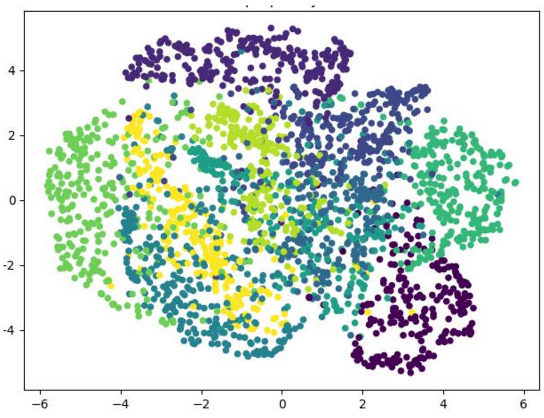


Symmetric sne:

Perplexity=5

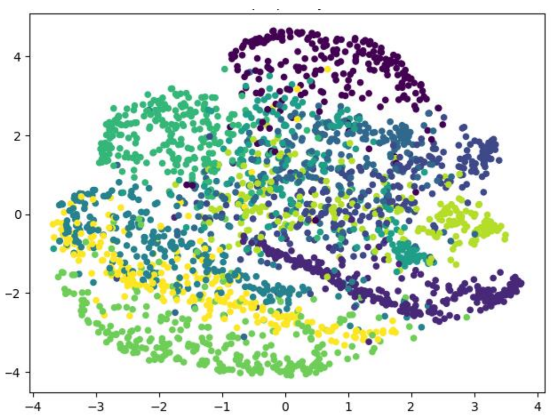


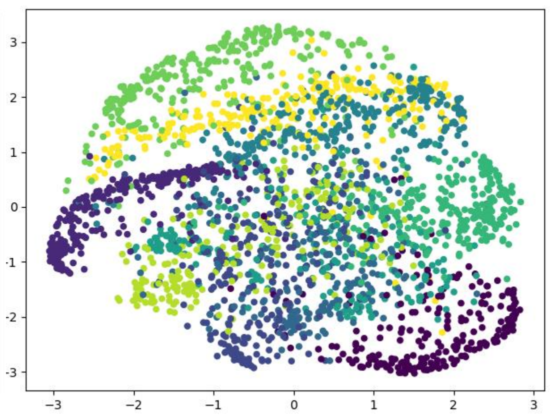
Perplexity=20



Perplexity=50

Perplexity=100



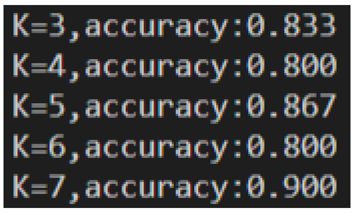
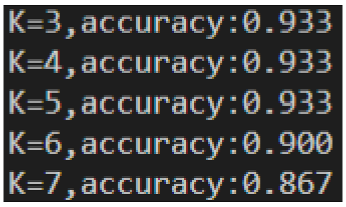


In symmetric sne, all the images have crowding problem, so the different perplexity seems has no effect.
In t-sne, the larger the perplexity is, the denser the distribution of points is.

Observation and discussion:

In eigenfaces, we can see the facial feature in most of these images, and it seems have shown many people's faces in one image. But in fisherfaces, we can only see the face outline from each image. It is because of PCA is to get the max. variance, so we can see the face features in each eigenfaces, but LDA is to get the max. ratio of between-classes and within-classes, in other words, it focuses on separating the image from others, so we can only see the outline of the face in fisherfaces.
And for recognition, if I increase the dimensions of projection space to 200, the LDA performance will be better than PCA.

PCA	LDA

 K=3,accuracy:0.833
K=4,accuracy:0.800
K=5,accuracy:0.867
K=6,accuracy:0.800
K=7,accuracy:0.900
	 K=3,accuracy:0.933
	K=4,accuracy:0.933
	K=5,accuracy:0.933
	K=6,accuracy:0.900
	K=7,accuracy:0.867

So, if the memory resource is limited, we should use PCA, but if not, LDA would be a better choice which has a better performance.