Machine Learning Homework 7

K-means & spectral clustering

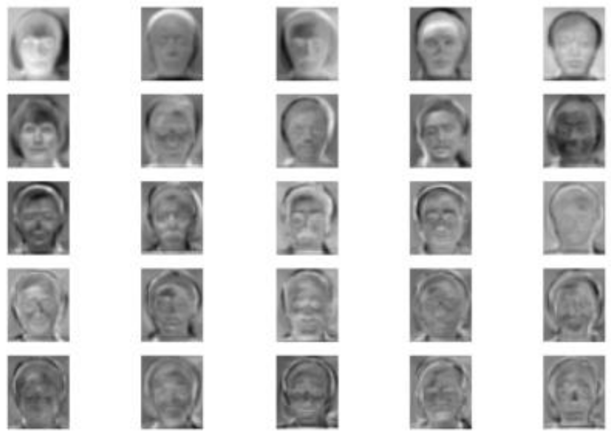
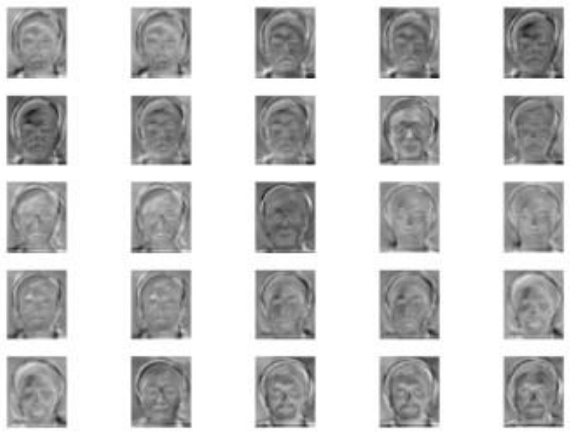
數據所碩一葉詠富 310554031

Github: https://github.com/frankye1000/NYCU-MachineLearning/tree/master/HW7

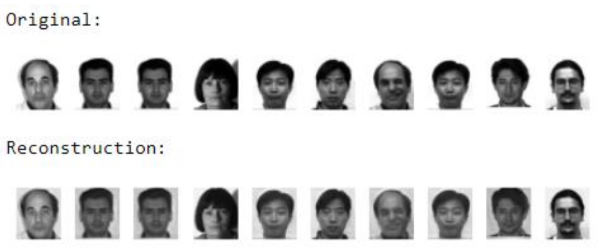
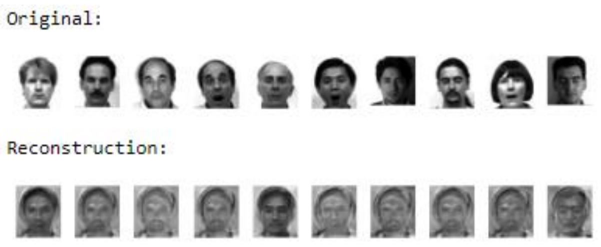
I. Kernel Eigenfaces

➢ Part1: Use PCA and LDA to show the first 25 eigenfaces and fisherfaces, and randomly pick 10 images to show their reconstruction. (please refer to the lecture slides).

- First 25 eigenfaces & fisherfaces

eigenfaces	fisherfaces
	

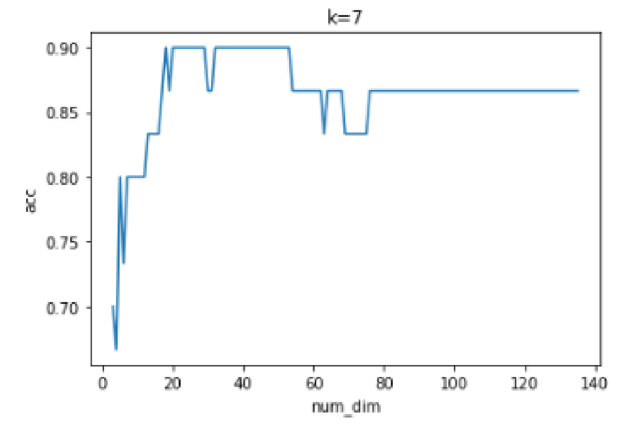
- Random pick 10 images recostruction

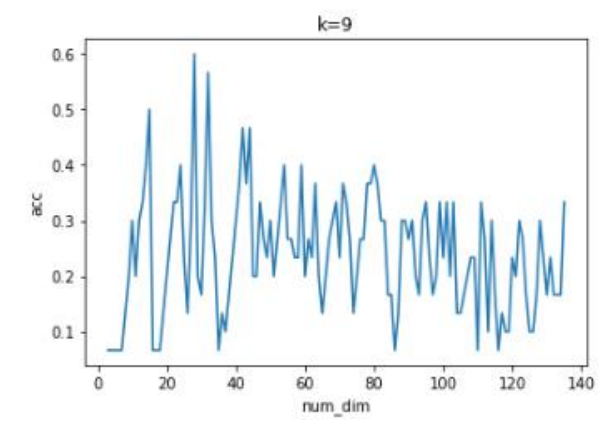
eigenfaces	fisherfaces
 Original: Reconstruction:	 Original: Reconstruction:

➢ Part2: Use PCA and LDA to do face recognition, and compute the performance. You should use k nearest neighbor to classify which subject the testing image belongs to.

- I try num_dim:3~135, k: 3~9 to get the best performance.

PCA	LDA
num_dim:18, k:7, acc:90%	**num_dim:28, k:9, acc:60%**





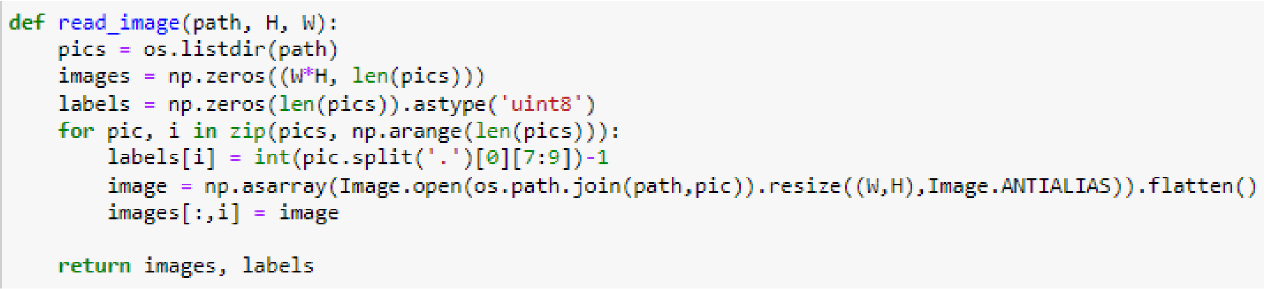
➢ Part3: Use kernel PCA and kernel LDA to do face recognition, and compute the performance. (You can choose whatever kernel you want, but you should try different kernels in your implementation.) Then compare the difference between simple LDA/PCA and kernel LDA/PCA, and the difference between different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

- In order to compare the difference between the simple LDA and the kernel LDA, I did not change the parameters num_dim & k.
- We can find the kernel PCA have a little worse performance than simple PCA. The RBF kernel is better than polynomial kernel.
- We can find the kernel LDA have the worse performance than simple LDA. The RBF kernel is better than polynomial kernel.

kernel	kernel PCA	kernel LDA
RBF	num_dim:18, k:7, acc:86.67%	num_dim:28, k:9, acc:56.67%
polynomial	num_dim:18, k:7, acc:83.33%	num_dim:28, k:9, acc:46.67%

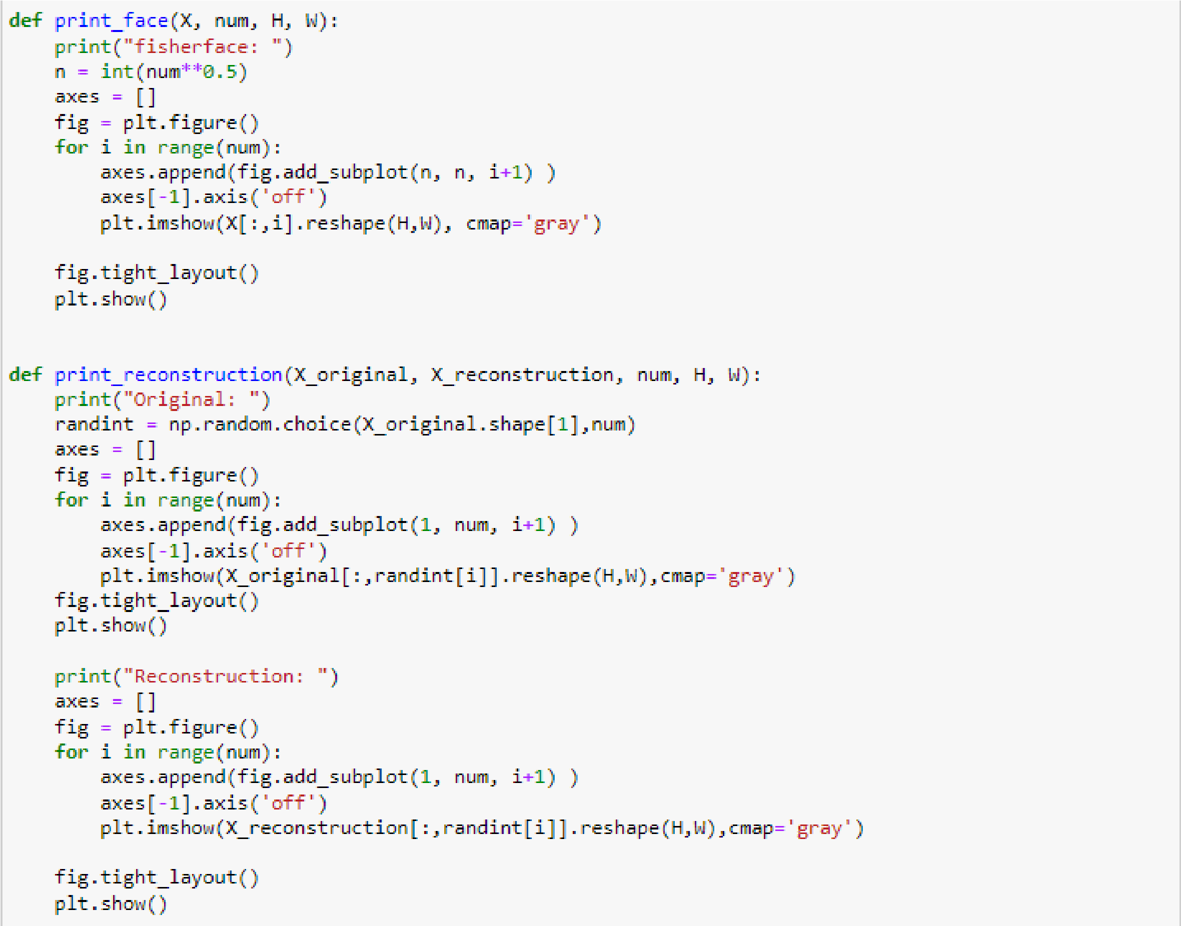
Code:

1. Read image.



```
def read_image(path, H, W):
    pics = os.listdir(path)
    images = np.zeros((W*H, len(pics)))
    labels = np.zeros(len(pics)).astype('uint8')
    for pic, i in zip(pics, np.arange(len(pics))):
        labels[i] = int(pic.split('.')[0][7:9])-1
        image = np.asarray(Image.open(os.path.join(path,pic)).resize((W,H),Image.ANTIALIAS)).flatten()
        images[:,i] = image

    return images, labels
```

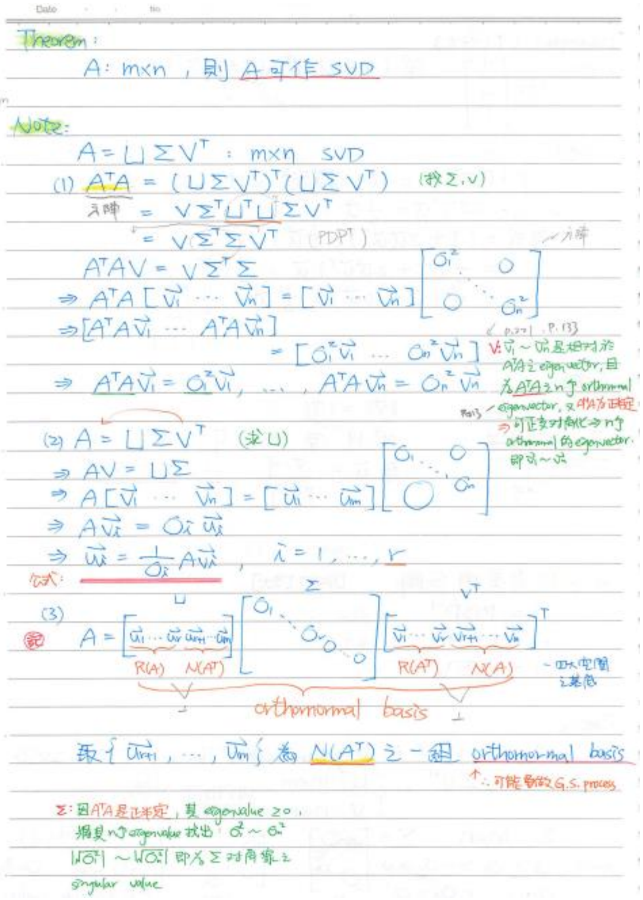
2. Print face & reconstruction result.



```python
def print_face(X, num, H, W):
    print("fisherface: ")
    n = int(num**0.5)
    axes = []
    fig = plt.figure()
    for i in range(num):
        axes.append(fig.add_subplot(n, n, i+1) )
        axes[-1].axis('off')
        plt.imshow(X[:,i].reshape(H,W), cmap='gray')

    fig.tight_layout()
    plt.show()


def print_reconstruction(X_original, X_reconstruction, num, H, W):
    print("Original: ")
    randint = np.random.choice(X_original.shape[1],num)
    axes = []
    fig = plt.figure()
    for i in range(num):
        axes.append(fig.add_subplot(1, num, i+1) )
        axes[-1].axis('off')
        plt.imshow(X_original[:,randint[i]].reshape(H,W),cmap='gray')
    fig.tight_layout()
    plt.show()

    print("Reconstruction: ")
    axes = []
    fig = plt.figure()
    for i in range(num):
        axes.append(fig.add_subplot(1, num, i+1) )
        axes[-1].axis('off')
        plt.imshow(X_reconstruction[:,randint[i]].reshape(H,W),cmap='gray')

    fig.tight_layout()
    plt.show()
```

3. PCA

I use the concept of SVD(U@Sigma@V.T) to calculate PCA.



Date No

Theorem:

A: m×n，則 A 可作 SVD

Note:

$A = U\Sigma V^T$: m×n SVD

(1) $A^TA = (U\Sigma V^T)^T(U\Sigma V^T)$ (找Σ, V)

方陣 $= V\Sigma^T U^T U\Sigma V^T$

$= V\Sigma^T\Sigma V^T$ (PDP^T) 方陣

$A^TAV = V\Sigma^T\Sigma$

$\Rightarrow A^TA[\vec{v_1} \cdots \vec{v_n}] = [\vec{v_1} \cdots \vec{v_n}]\begin{bmatrix}\sigma_1^2 & & 0\\ & \ddots & \\ 0 & & \sigma_n^2\end{bmatrix}$

$\Rightarrow [A^TA\vec{v_1} \cdots A^TA\vec{v_n}]$

$= [\sigma_1^2\vec{v_1} \cdots \sigma_n^2\vec{v_n}]$

$\Rightarrow A^TA\vec{v_1} = \sigma_1^2\vec{v_1}, \ldots, A^TA\vec{v_n} = \sigma_n^2\vec{v_n}$

p.271, P.133
V: $\vec{v_1}\sim\vec{v_n}$ 是相对於 A^TA 之 eigenvector，且為 A^TA 之 n 个 orthonormal eigenvector，又 A^TA 為正定 ⇒ 可正交对角化 ⇒ n个 orthonormal 的 eigenvector. 即 $\vec{v_1}\sim\vec{v_n}$

(2) $A = U\Sigma V^T$ (求U)

$\Rightarrow AV = U\Sigma$

$\Rightarrow A[\vec{v_1} \cdots \vec{v_n}] = [\vec{u_1} \cdots \vec{u_m}]\begin{bmatrix}\sigma_1 & & 0\\ & \ddots & \sigma_n \\ 0 & & \end{bmatrix}$

$\Rightarrow A\vec{v_i} = \sigma_i\vec{u_i}$

公式: $\Rightarrow \vec{u_i} = \frac{1}{\sigma_i}A\vec{v_i}$, $i = 1, \ldots, r$

(3)

記 $$A = \overset{U}{\left[\underbrace{\vec{u_1}\cdots\vec{u_r}}_{R(A)}\ \underbrace{\vec{u_{r+1}}\cdots\vec{u_m}}_{N(A^T)}\right]}\overset{\Sigma}{\begin{bmatrix}\sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_r & & \\ & & & 0 & \\ & & & & \ddots\ 0\end{bmatrix}}\overset{V^T}{\left[\underbrace{\vec{v_1}\cdots\vec{v_r}}_{R(A^T)}\ \underbrace{\vec{v_{r+1}}\cdots\vec{v_n}}_{N(A)}\right]^T}$$

— 四大空間之基底

⊥ orthonormal basis ⊥

取 $\{\vec{u_{r+1}}, \ldots, \vec{u_m}\}$ 為 $N(A^T)$ 之一組 orthonormal basis

↑∴可能要做 G.S. process

Σ: 因 A^TA 是正半定，其 eigenvalue ≥ 0，
將其 n个 eigenvalue 找出：$\sigma_1^2 \sim \sigma_n^2$
$|\sqrt{\sigma_1^2}| \sim |\sqrt{\sigma_n^2}|$ 即為 Σ 对角線之
singular value

I use five steps to get PCA.

step1: around center.

step2: X.T@X to caculate sigma&V.

step3: sort eigenvalue big -> small.

step4: delete eigenvalue<0

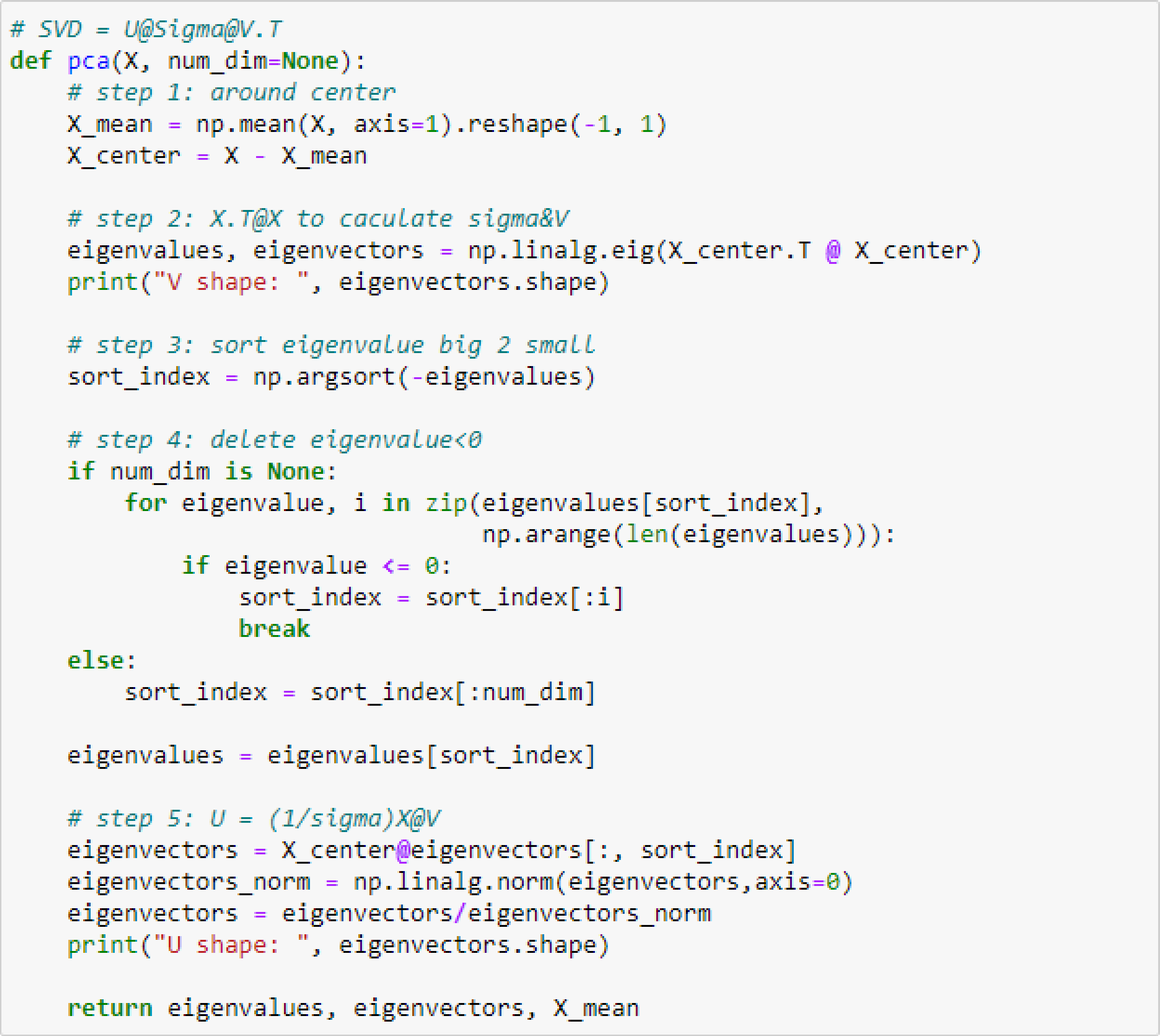
step5: U = (1/sigma)X@V

We can get:

X shape:(45045, 135)

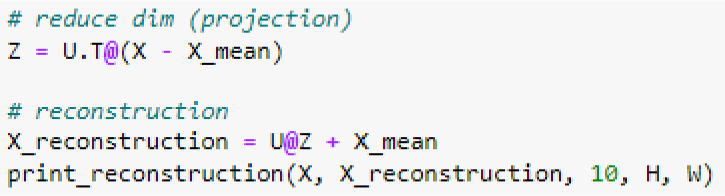
V shape:(135, 135)

U shape:(45045, num_dim)   # can set num_dim



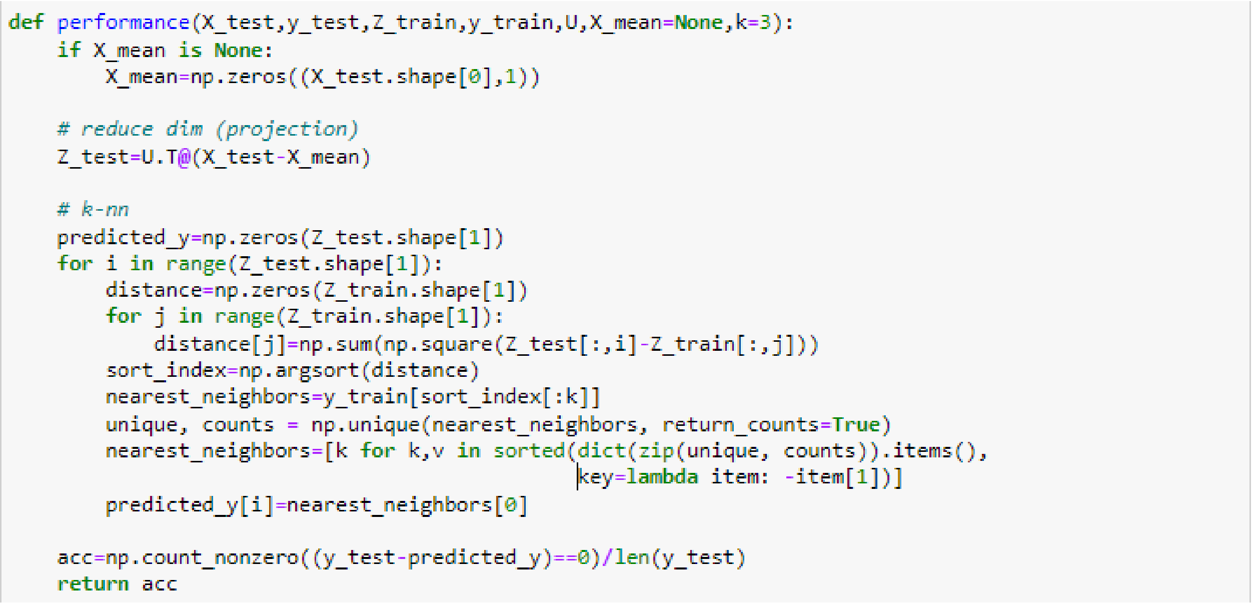
```python
# SVD = U@Sigma@V.T
def pca(X, num_dim=None):
    # step 1: around center
    X_mean = np.mean(X, axis=1).reshape(-1, 1)
    X_center = X - X_mean

    # step 2: X.T@X to caculate sigma&V
    eigenvalues, eigenvectors = np.linalg.eig(X_center.T @ X_center)
    print("V shape: ", eigenvectors.shape)

    # step 3: sort eigenvalue big 2 small
    sort_index = np.argsort(-eigenvalues)

    # step 4: delete eigenvalue<0
    if num_dim is None:
        for eigenvalue, i in zip(eigenvalues[sort_index],
                                 np.arange(len(eigenvalues))):
            if eigenvalue <= 0:
                sort_index = sort_index[:i]
                break
    else:
        sort_index = sort_index[:num_dim]

    eigenvalues = eigenvalues[sort_index]

    # step 5: U = (1/sigma)X@V
    eigenvectors = X_center@eigenvectors[:, sort_index]
    eigenvectors_norm = np.linalg.norm(eigenvectors,axis=0)
    eigenvectors = eigenvectors/eigenvectors_norm
    print("U shape: ", eigenvectors.shape)

    return eigenvalues, eigenvectors, X_mean
```

Reconstruction and print result.



```python
# reduce dim (projection)
Z = U.T@(X - X_mean)

# reconstruction
X_reconstruction = U@Z + X_mean
print_reconstruction(X, X_reconstruction, 10, H, W)
```

Use KNN to compare the performance.

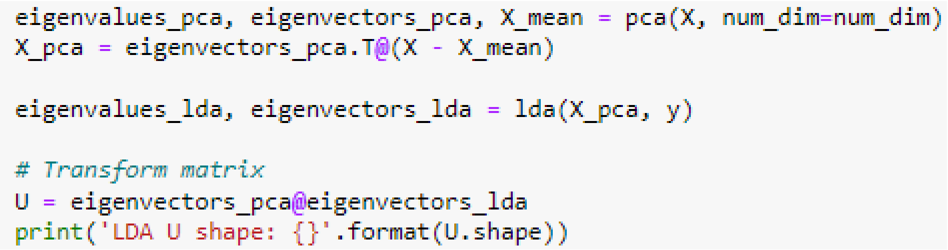


```
def performance(X_test,y_test,Z_train,y_train,U,X_mean=None,k=3):
    if X_mean is None:
        X_mean=np.zeros((X_test.shape[0],1))

    # reduce dim (projection)
    Z_test=U.T@(X_test-X_mean)

    # k-nn
    predicted_y=np.zeros(Z_test.shape[1])
    for i in range(Z_test.shape[1]):
        distance=np.zeros(Z_train.shape[1])
        for j in range(Z_train.shape[1]):
            distance[j]=np.sum(np.square(Z_test[:,i]-Z_train[:,j]))
        sort_index=np.argsort(distance)
        nearest_neighbors=y_train[sort_index[:k]]
        unique, counts = np.unique(nearest_neighbors, return_counts=True)
        nearest_neighbors=[k for k,v in sorted(dict(zip(unique, counts)).items(),
                                               key=lambda item: -item[1])]
        predicted_y[i]=nearest_neighbors[0]

    acc=np.count_nonzero((y_test-predicted_y)==0)/len(y_test)
    return acc
```

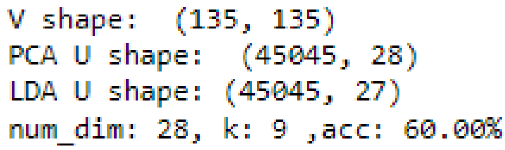
4. LDA

Because of the singularity of the within-class scatter matrix. First I use pca to reduce dimension to 28, then use lda to reduce dimension to 27. It can get the best performance acc:60%.

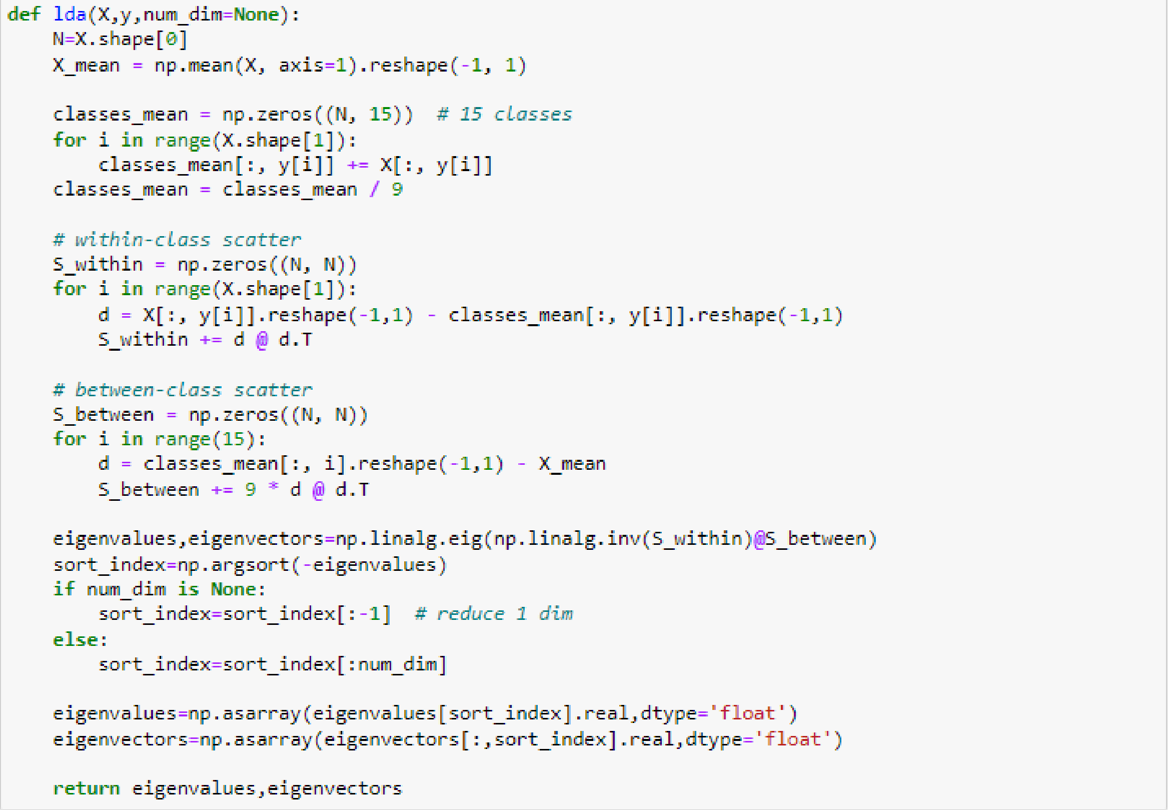
The transformation matrix U is eigenvector of PCA @ eigenvector of LDA.



```
eigenvalues_pca, eigenvectors_pca, X_mean = pca(X, num_dim=num_dim)
X_pca = eigenvectors_pca.T@(X - X_mean)

eigenvalues_lda, eigenvectors_lda = lda(X_pca, y)

# Transform matrix
U = eigenvectors_pca@eigenvectors_lda
print('LDA U shape: {}'.format(U.shape))
```



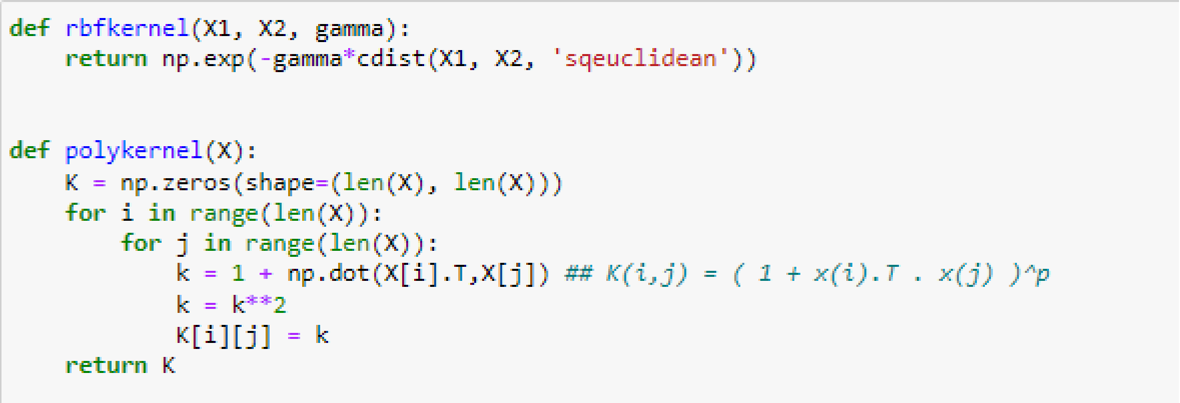
```
V shape:  (135, 135)
PCA U shape:  (45045, 28)
LDA U shape: (45045, 27)
num_dim: 28, k: 9 ,acc: 60.00%
```



```
def lda(X,y,num_dim=None):
    N=X.shape[0]
    X_mean = np.mean(X, axis=1).reshape(-1, 1)

    classes_mean = np.zeros((N, 15))  # 15 classes
    for i in range(X.shape[1]):
        classes_mean[:, y[i]] += X[:, y[i]]
    classes_mean = classes_mean / 9

    # within-class scatter
    S_within = np.zeros((N, N))
    for i in range(X.shape[1]):
        d = X[:, y[i]].reshape(-1,1) - classes_mean[:, y[i]].reshape(-1,1)
        S_within += d @ d.T

    # between-class scatter
    S_between = np.zeros((N, N))
    for i in range(15):
        d = classes_mean[:, i].reshape(-1,1) - X_mean
        S_between += 9 * d @ d.T

    eigenvalues,eigenvectors=np.linalg.eig(np.linalg.inv(S_within)@S_between)
    sort_index=np.argsort(-eigenvalues)
    if num_dim is None:
        sort_index=sort_index[:-1]  # reduce 1 dim
    else:
        sort_index=sort_index[:num_dim]

    eigenvalues=np.asarray(eigenvalues[sort_index].real,dtype='float')
    eigenvectors=np.asarray(eigenvectors[:,sort_index].real,dtype='float')

    return eigenvalues,eigenvectors
```

5. Kernel

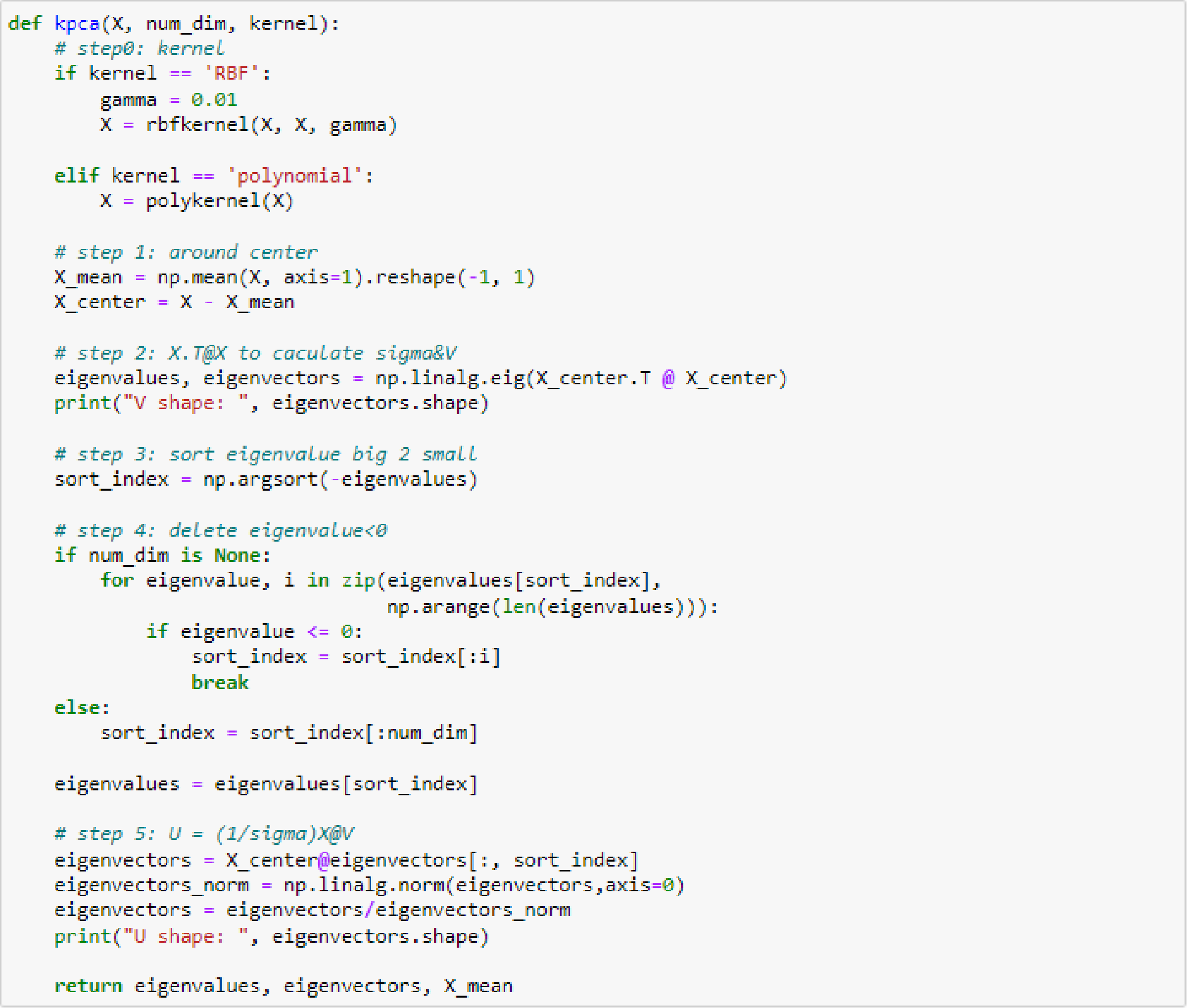
   Define rbf kernel & polynomial kernel.



```
def rbfkernel(X1, X2, gamma):
    return np.exp(-gamma*cdist(X1, X2, 'sqeuclidean'))


def polykernel(X):
    K = np.zeros(shape=(len(X), len(X)))
    for i in range(len(X)):
        for j in range(len(X)):
            k = 1 + np.dot(X[i].T,X[j]) ## K(i,j) = ( 1 + x(i).T . x(j) )^p
            k = k**2
            K[i][j] = k
    return K
```

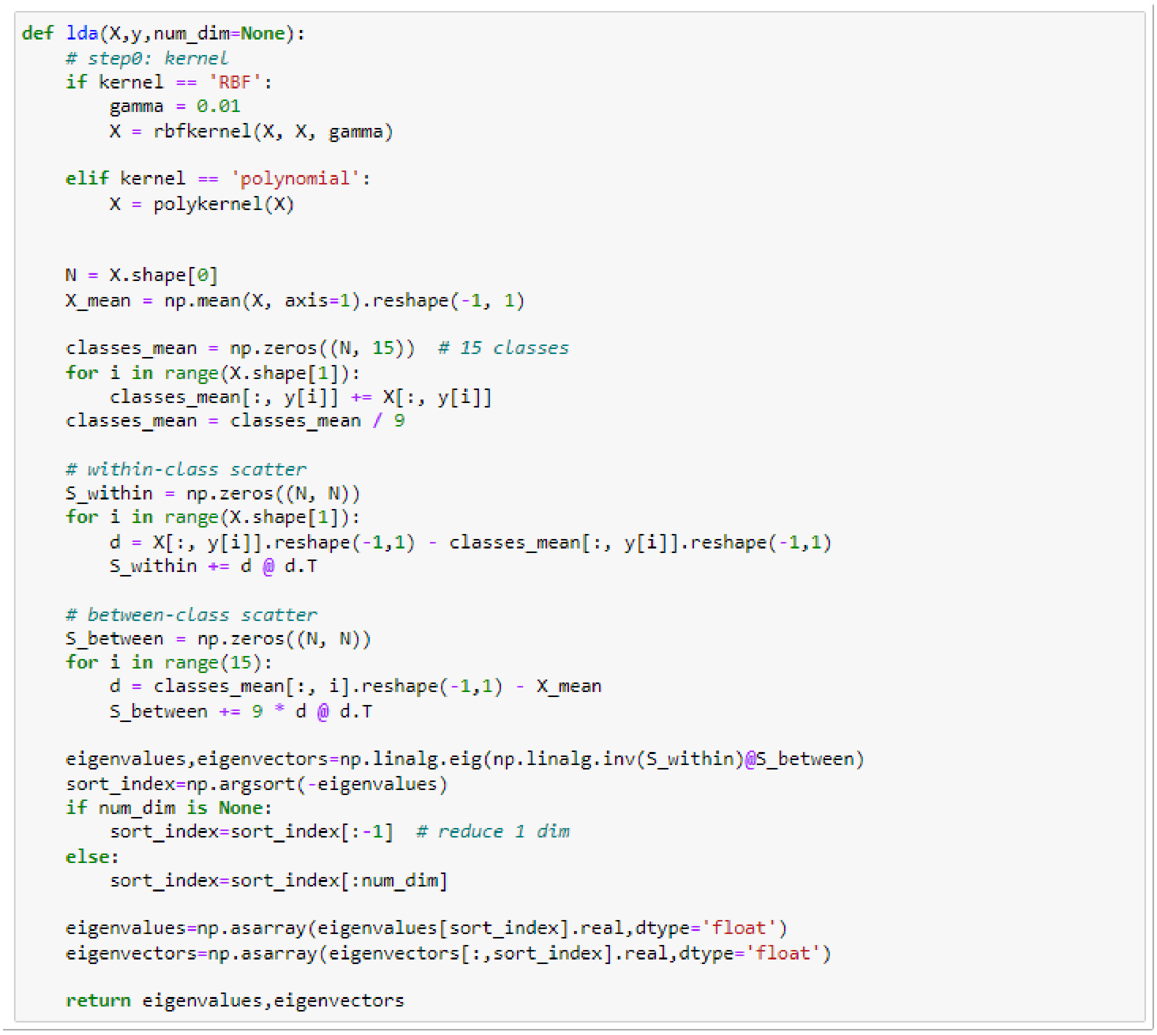
6. Kernel PCA

   Through the kernel PCA, we perform a non-linear correspondence function to transform the data into a higher-dimensional space, and in this high-dimensional space, use standard PCA, and then project the data back to a lower-dimensional subspace.



```
def kpca(X, num_dim, kernel):
    # step0: kernel
    if kernel == 'RBF':
        gamma = 0.01
        X = rbfkernel(X, X, gamma)

    elif kernel == 'polynomial':
        X = polykernel(X)

    # step 1: around center
    X_mean = np.mean(X, axis=1).reshape(-1, 1)
    X_center = X - X_mean

    # step 2: X.T@X to caculate sigma&V
    eigenvalues, eigenvectors = np.linalg.eig(X_center.T @ X_center)
    print("V shape: ", eigenvectors.shape)

    # step 3: sort eigenvalue big 2 small
    sort_index = np.argsort(-eigenvalues)

    # step 4: delete eigenvalue<0
    if num_dim is None:
        for eigenvalue, i in zip(eigenvalues[sort_index],
                                 np.arange(len(eigenvalues))):
            if eigenvalue <= 0:
                sort_index = sort_index[:i]
                break
    else:
        sort_index = sort_index[:num_dim]

    eigenvalues = eigenvalues[sort_index]

    # step 5: U = (1/sigma)X@V
    eigenvectors = X_center@eigenvectors[:, sort_index]
    eigenvectors_norm = np.linalg.norm(eigenvectors,axis=0)
    eigenvectors = eigenvectors/eigenvectors_norm
    print("U shape: ", eigenvectors.shape)

    return eigenvalues, eigenvectors, X_mean
```

7. Kernel LDA
   Also through the kernel LDA, we perform a non-linear correspondence function to transform the data into a higher-dimensional space, and in this high-dimensional space, use standard LDA, and then project the data back to a lower-dimensional subspace.



```
def lda(X,y,num_dim=None):
    # step0: kernel
    if kernel == 'RBF':
        gamma = 0.01
        X = rbfkernel(X, X, gamma)

    elif kernel == 'polynomial':
        X = polykernel(X)


    N = X.shape[0]
    X_mean = np.mean(X, axis=1).reshape(-1, 1)

    classes_mean = np.zeros((N, 15))  # 15 classes
    for i in range(X.shape[1]):
        classes_mean[:, y[i]] += X[:, y[i]]
    classes_mean = classes_mean / 9

    # within-class scatter
    S_within = np.zeros((N, N))
    for i in range(X.shape[1]):
        d = X[:, y[i]].reshape(-1,1) - classes_mean[:, y[i]].reshape(-1,1)
        S_within += d @ d.T

    # between-class scatter
    S_between = np.zeros((N, N))
    for i in range(15):
        d = classes_mean[:, i].reshape(-1,1) - X_mean
        S_between += 9 * d @ d.T

    eigenvalues,eigenvectors=np.linalg.eig(np.linalg.inv(S_within)@S_between)
    sort_index=np.argsort(-eigenvalues)
    if num_dim is None:
        sort_index=sort_index[:-1]  # reduce 1 dim
    else:
        sort_index=sort_index[:num_dim]

    eigenvalues=np.asarray(eigenvalues[sort_index].real,dtype='float')
    eigenvectors=np.asarray(eigenvectors[:,sort_index].real,dtype='float')

    return eigenvalues,eigenvectors
```

II. t-SNE

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

- t-SNE & symmetric SNE is different in **q_{ij} & gradient**. t-SNE uses t- distribution. symmetric SNE uses Gaussian distribution.
- We can find $\|y_i - y_j\|^2 = np.add(np.add(num, sum\_Y).T, sum\_Y)$, so we modify num.

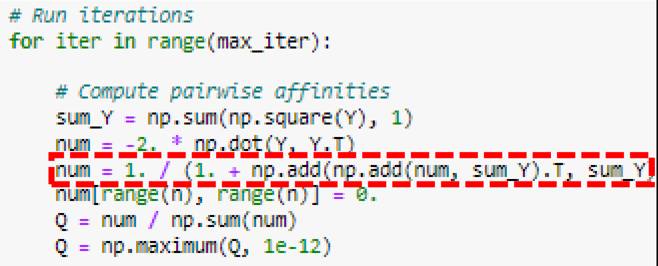
<table>
<tr><td>t-SNE</td><td>symmetric SNE</td></tr>
<tr><td>

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

</td><td>

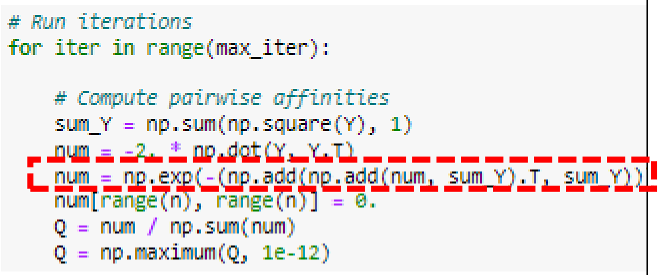
$$q_{ij} = \frac{\exp(- \mid\mid y_i - y_j \mid\mid^2)}{\sum_{k \neq l} \exp(- \mid\mid y_l - y_k \mid\mid^2)}$$

</td></tr>
<tr><td>



```
# Run iterations
for iter in range(max_iter):

    # Compute pairwise affinities
    sum_Y = np.sum(np.square(Y), 1)
    num = -2. * np.dot(Y, Y.T)
    num = 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)
```

</td><td>



```
# Run iterations
for iter in range(max_iter):

    # Compute pairwise affinities
    sum_Y = np.sum(np.square(Y), 1)
    num = -2. * np.dot(Y, Y.T)
    num = np.exp(-(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)
```

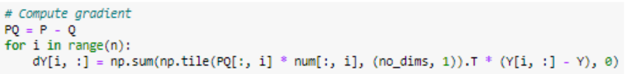
</td></tr>
<tr><td>

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

</td><td>

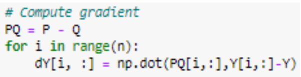
$$\frac{\partial C}{\partial y_i} = 2\sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

</td></tr>
<tr><td>



```
# 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, :] - Y), 0)
```

</td><td>



```
# Compute gradient
PQ = P - Q
for i in range(n):
    dY[i, :] = np.dot(PQ[i,:],Y[i,:]-Y)
```

</td></tr>
</table>

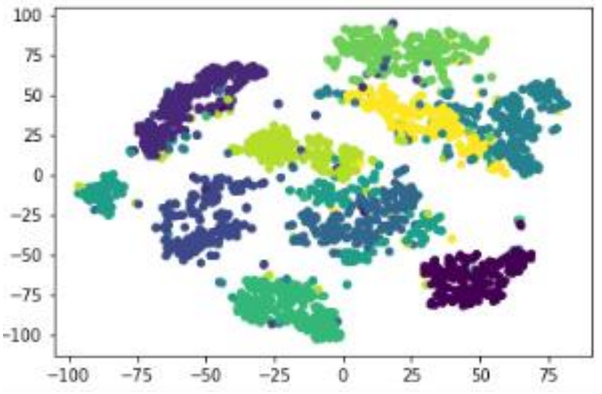
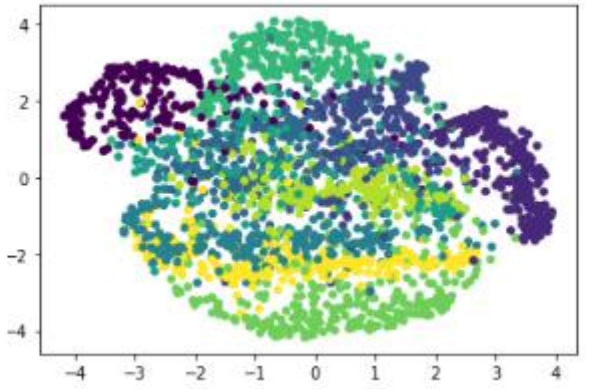
- The **crowded problem** means that the clusters are gathered together and cannot be distinguished(fig.1).
- The distance relationship in high-dimensional data cannot be completely retained in low-dimensional space.

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

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

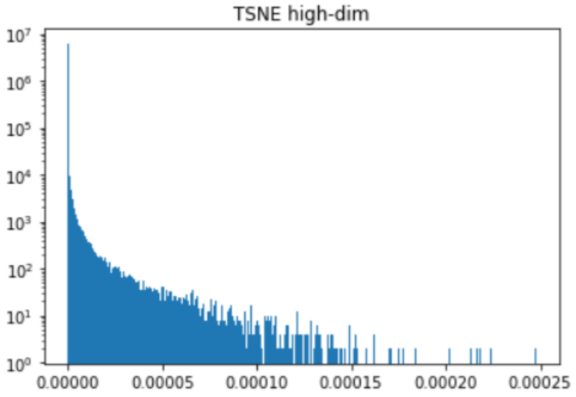
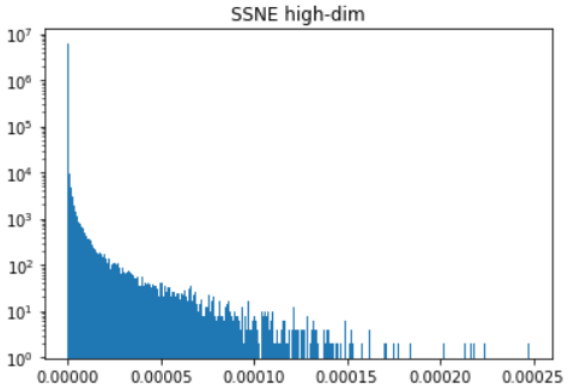
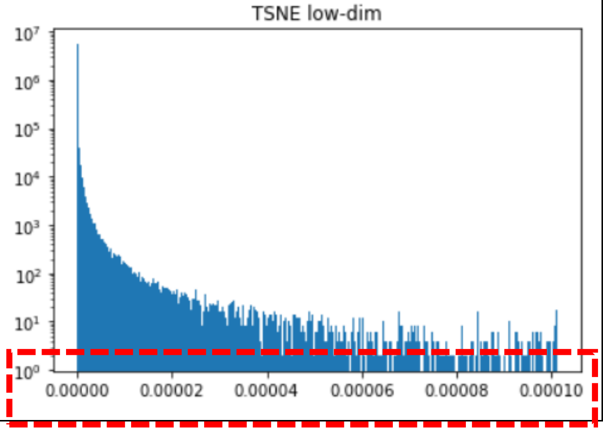
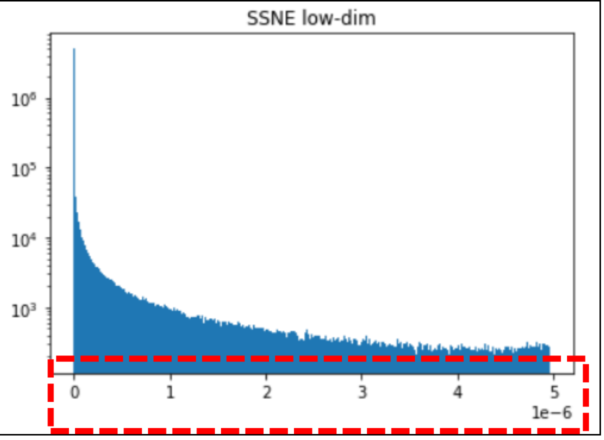
perplexity=20

t-SNE	symmetric SNE

 (fig.1)	 (fig.2)
HW7\tsne_python\GIF\tsne.gif	HW7\tsne_python\GIF\ssne.gif

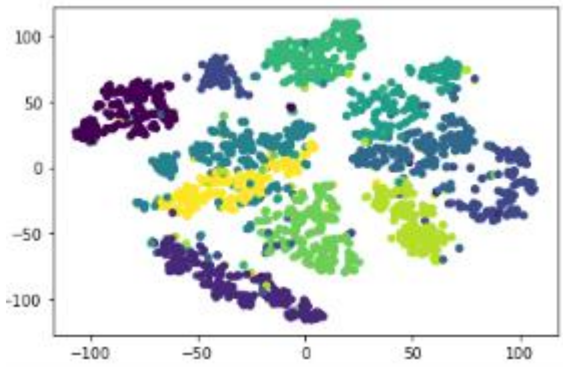
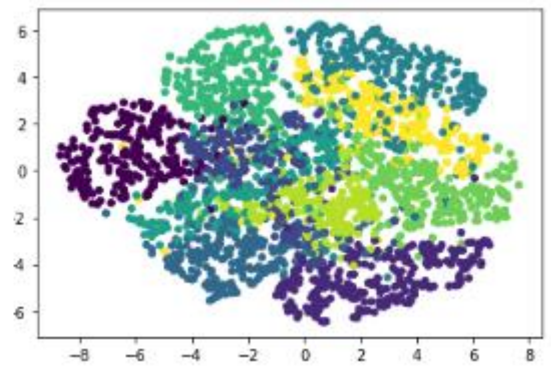
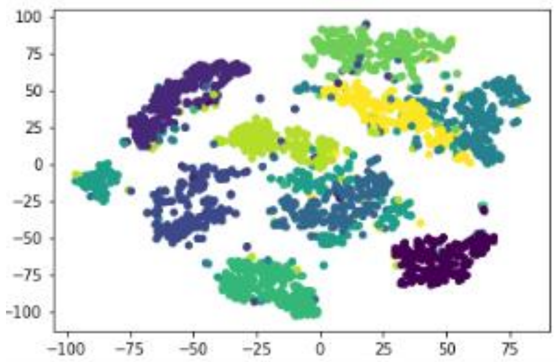
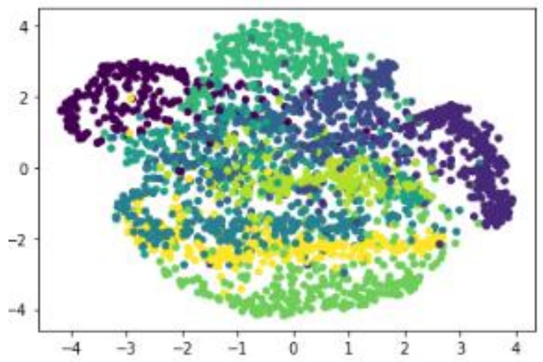
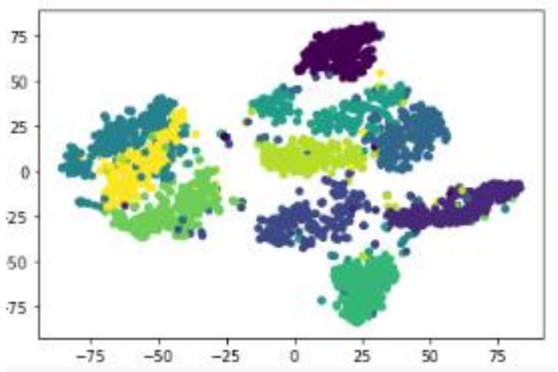
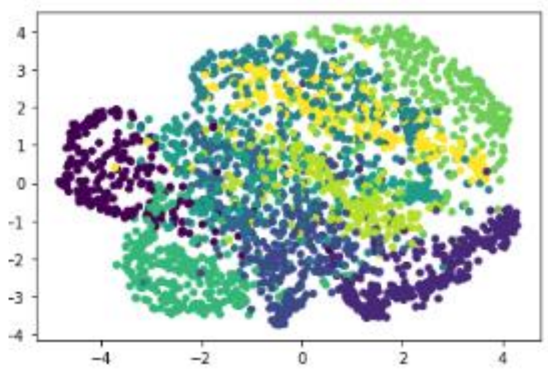
- ➢ Part3: Visualize the distribution of pairwise similarities in both high-dimensional space and low-dimensional space, based on both t-SNE and symmetric SNE.

  - ● The range of ssne in low-dim is much smaller than that of tsne, so ssne has a **crowded problem**.

	t-SNE	symmetric SNE
High-dim		
Low-dim		

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

  - ● We observe a tendency towards clearer shapes as the perplexity value increases.

perplexity	t-SNE	symmetric SNE
10		
20		
40		

III. reference

[1] https://www.youtube.com/watch?v=g-Hb26agBFg&ab_channel=Serrano.Academy

[2] https://kknews.cc/code/a6olrln.html

[3] https://www.uj5u.com/houduan/294222.html

[4] https://kknews.cc/code/o2gbm45.html

[5] https://arbu00.blogspot.com/2017/02/7-kernel-pca.html