# Machine Learning Homework 7

**PCA, LDA & SNE**

Student ID: 0886006

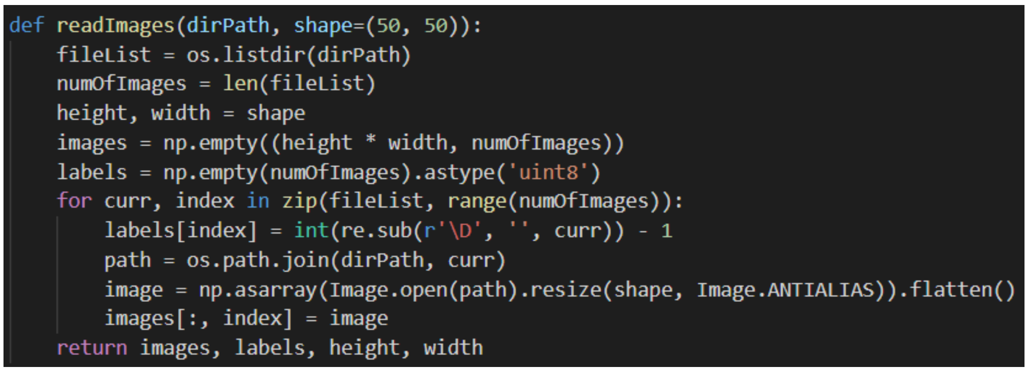
Name: 張又允

I. Kernel Eigenfaces/Fisherfaces

part1.py and util.py

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

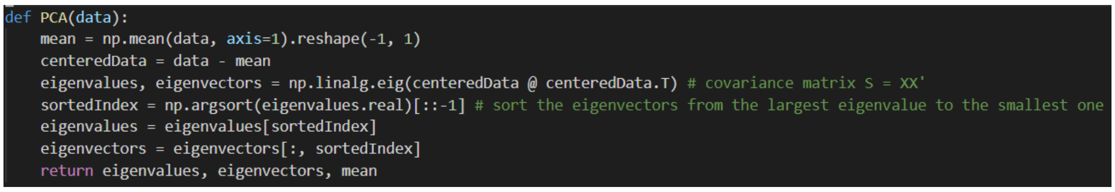
For the ease of implementation, we resize the image to 50 X 50. To get the label of each image, we read the first number encountered in each filename.



```python
def readImages(dirPath, shape=(50, 50)):
    fileList = os.listdir(dirPath)
    numOfImages = len(fileList)
    height, width = shape
    images = np.empty((height * width, numOfImages))
    labels = np.empty(numOfImages).astype('uint8')
    for curr, index in zip(fileList, range(numOfImages)):
        labels[index] = int(re.sub(r'\D', '', curr)) - 1
        path = os.path.join(dirPath, curr)
        image = np.asarray(Image.open(path).resize(shape, Image.ANTIALIAS)).flatten()
        images[:, index] = image
    return images, labels, height, width
```

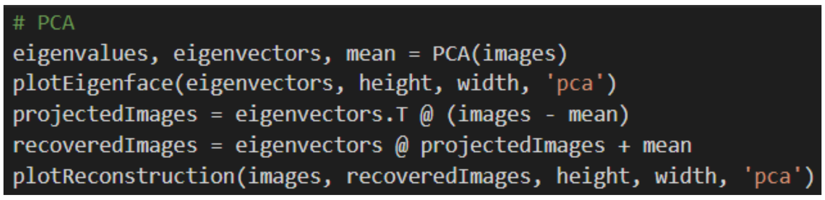
<u>PCA</u>

For PCA, we first center the data by subtracting the mean and then compute the eigenvectors of the covariance matrix of the centered data. We sort the eigenvectors based on the corresponding eigenvalues.



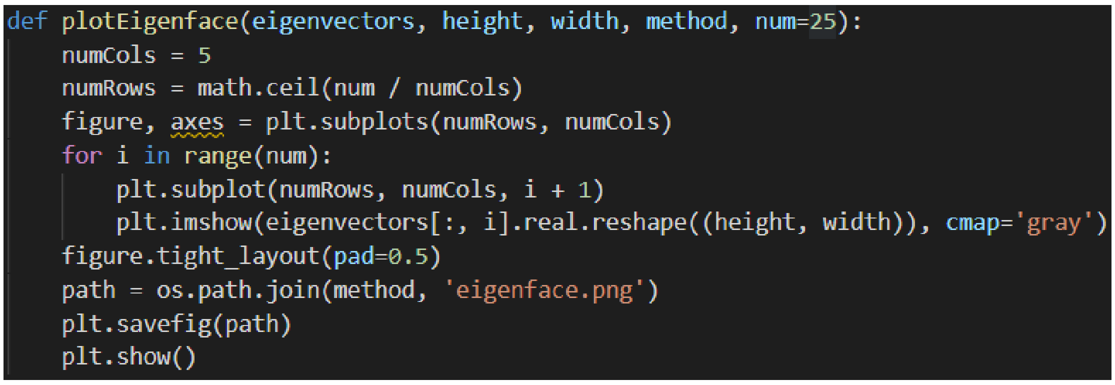
```python
def PCA(data):
    mean = np.mean(data, axis=1).reshape(-1, 1)
    centeredData = data - mean
    eigenvalues, eigenvectors = np.linalg.eig(centeredData @ centeredData.T) # covariance matrix S = XX'
    sortedIndex = np.argsort(eigenvalues.real)[::-1] # sort the eigenvectors from the largest eigenvalue to the smallest one
    eigenvalues = eigenvalues[sortedIndex]
    eigenvectors = eigenvectors[:, sortedIndex]
    return eigenvalues, eigenvectors, mean
```

For projection, we simply multiply the data by the transpose of the eigenvector matrix to get the coordinate of the data in the eigenspace. To recover, we just do the converse operation, i.e., multiply the projected data by the eigenvector matrix and then add the mean back.

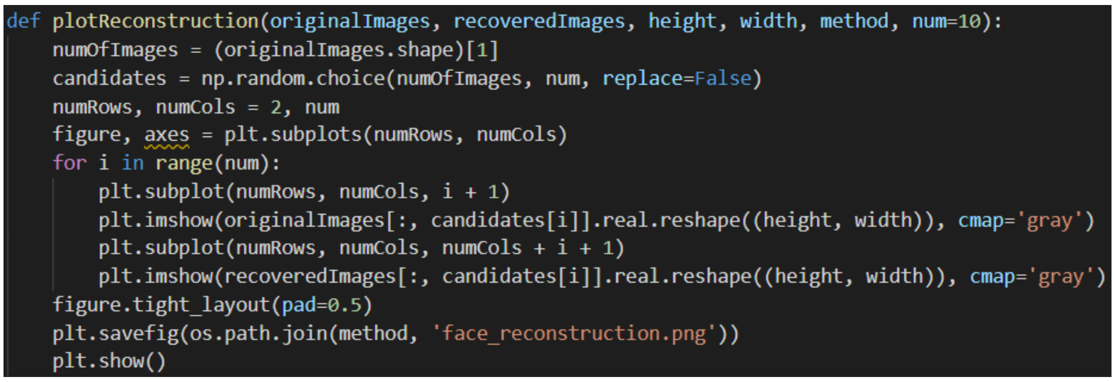


```python
# PCA
eigenvalues, eigenvectors, mean = PCA(images)
plotEigenface(eigenvectors, height, width, 'pca')
projectedImages = eigenvectors.T @ (images - mean)
recoveredImages = eigenvectors @ projectedImages + mean
plotReconstruction(images, recoveredImages, height, width, 'pca')
```

We show the eigenface by plotting the eigenvectors and recover 10 images randomly without replacement.

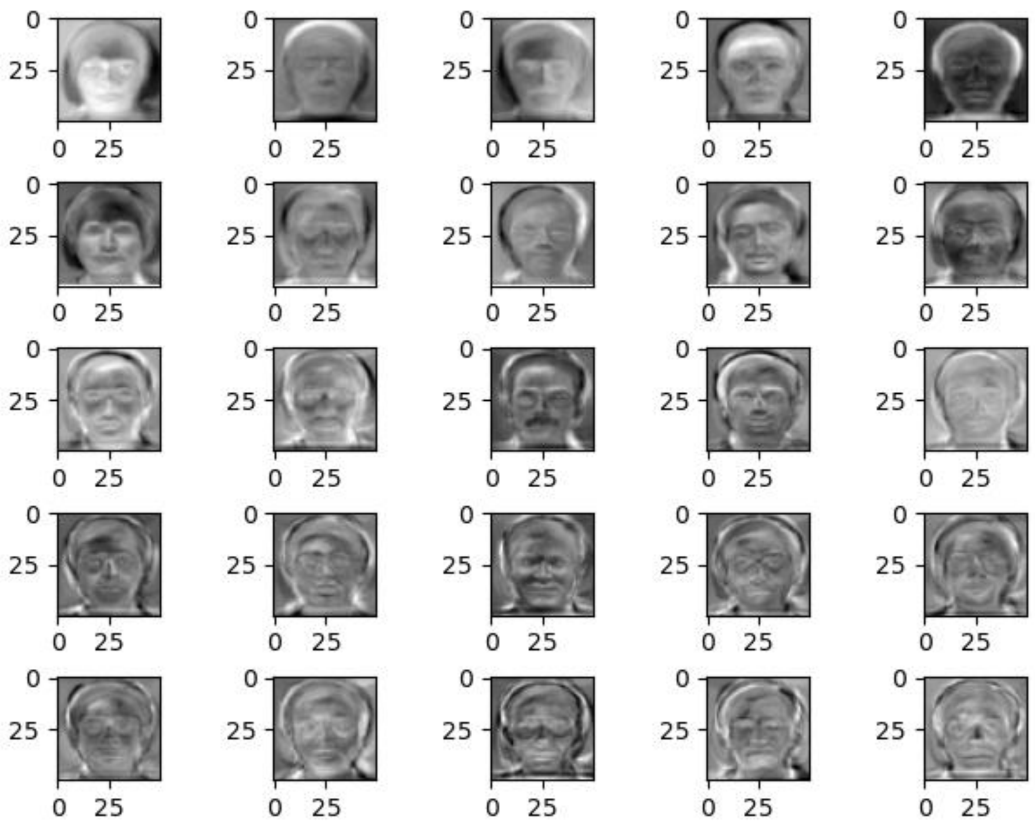


```
def plotEigenface(eigenvectors, height, width, method, num=25):
    numCols = 5
    numRows = math.ceil(num / numCols)
    figure, axes = plt.subplots(numRows, numCols)
    for i in range(num):
        plt.subplot(numRows, numCols, i + 1)
        plt.imshow(eigenvectors[:, i].real.reshape((height, width)), cmap='gray')
    figure.tight_layout(pad=0.5)
    path = os.path.join(method, 'eigenface.png')
    plt.savefig(path)
    plt.show()
```

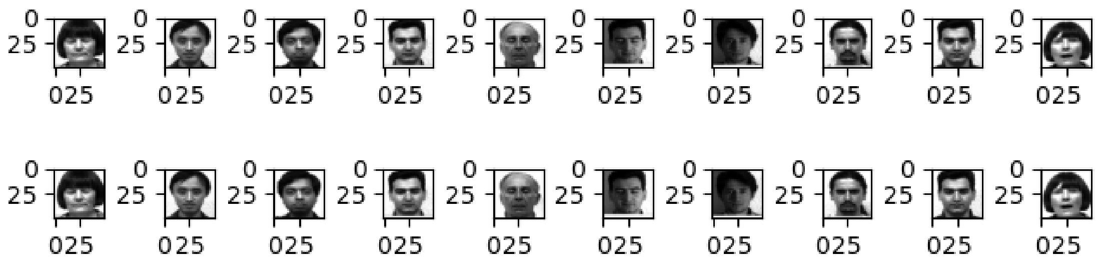


```
def plotReconstruction(originalImages, recoveredImages, height, width, method, num=10):
    numOfImages = (originalImages.shape)[1]
    candidates = np.random.choice(numOfImages, num, replace=False)
    numRows, numCols = 2, num
    figure, axes = plt.subplots(numRows, numCols)
    for i in range(num):
        plt.subplot(numRows, numCols, i + 1)
        plt.imshow(originalImages[:, candidates[i]].real.reshape((height, width)), cmap='gray')
        plt.subplot(numRows, numCols, numCols + i + 1)
        plt.imshow(recoveredImages[:, candidates[i]].real.reshape((height, width)), cmap='gray')
    figure.tight_layout(pad=0.5)
    plt.savefig(os.path.join(method, 'face_reconstruction.png'))
    plt.show()
```

- Eigenfaces



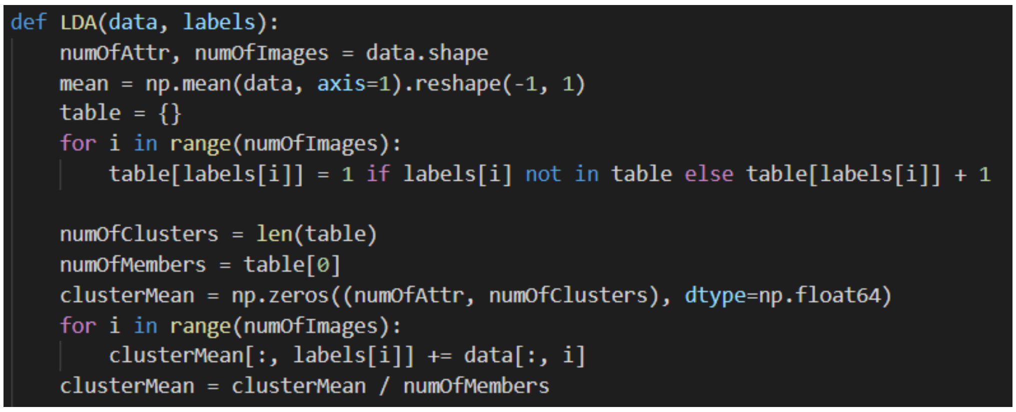
- Reconstruction results



(Upper row: original images; lower row: reconstructed images)

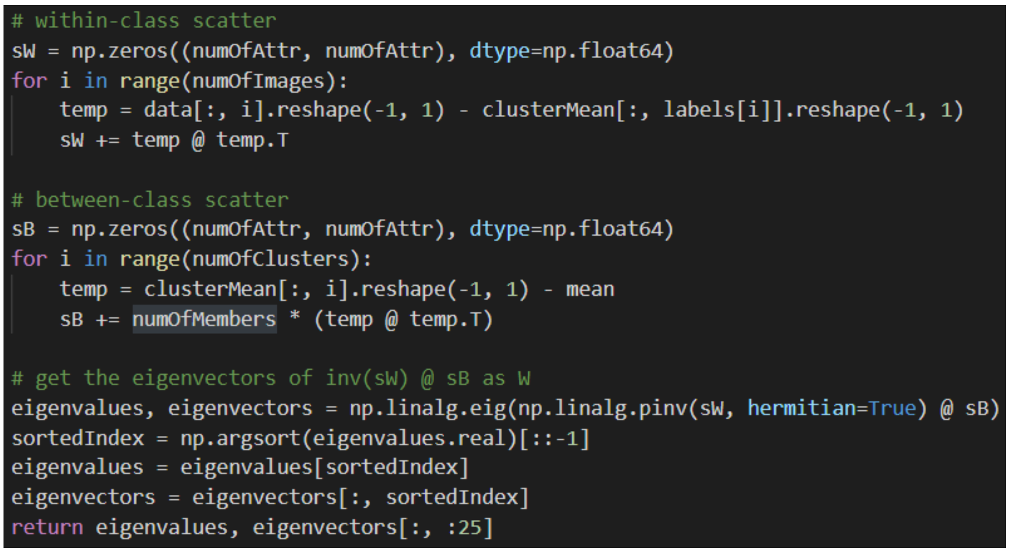
<u>LDA</u>

For LDA, we first calculate the mean of each class by the help of the label (i.e., supervised learning).



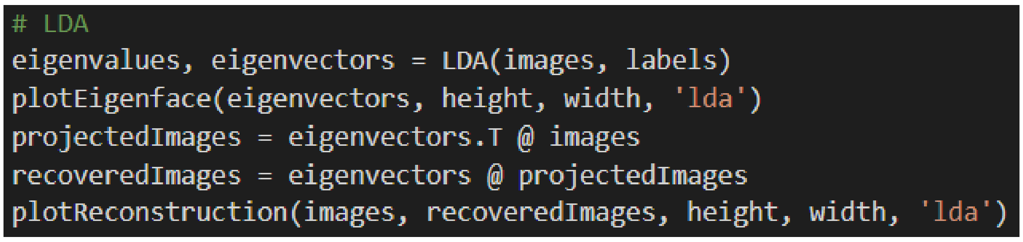
```python
def LDA(data, labels):
    numOfAttr, numOfImages = data.shape
    mean = np.mean(data, axis=1).reshape(-1, 1)
    table = {}
    for i in range(numOfImages):
        table[labels[i]] = 1 if labels[i] not in table else table[labels[i]] + 1

    numOfClusters = len(table)
    numOfMembers = table[0]
    clusterMean = np.zeros((numOfAttr, numOfClusters), dtype=np.float64)
    for i in range(numOfImages):
        clusterMean[:, labels[i]] += data[:, i]
    clusterMean = clusterMean / numOfMembers
```

Then we calculate the within-class scatter $S_W$ and the between-class scatter $S_B$. We get the projection matrix $W$ by calculating $S_W^{-1}S_B$. To prevent the singularity problem, we take the pseudo-inverse of $S_W$.



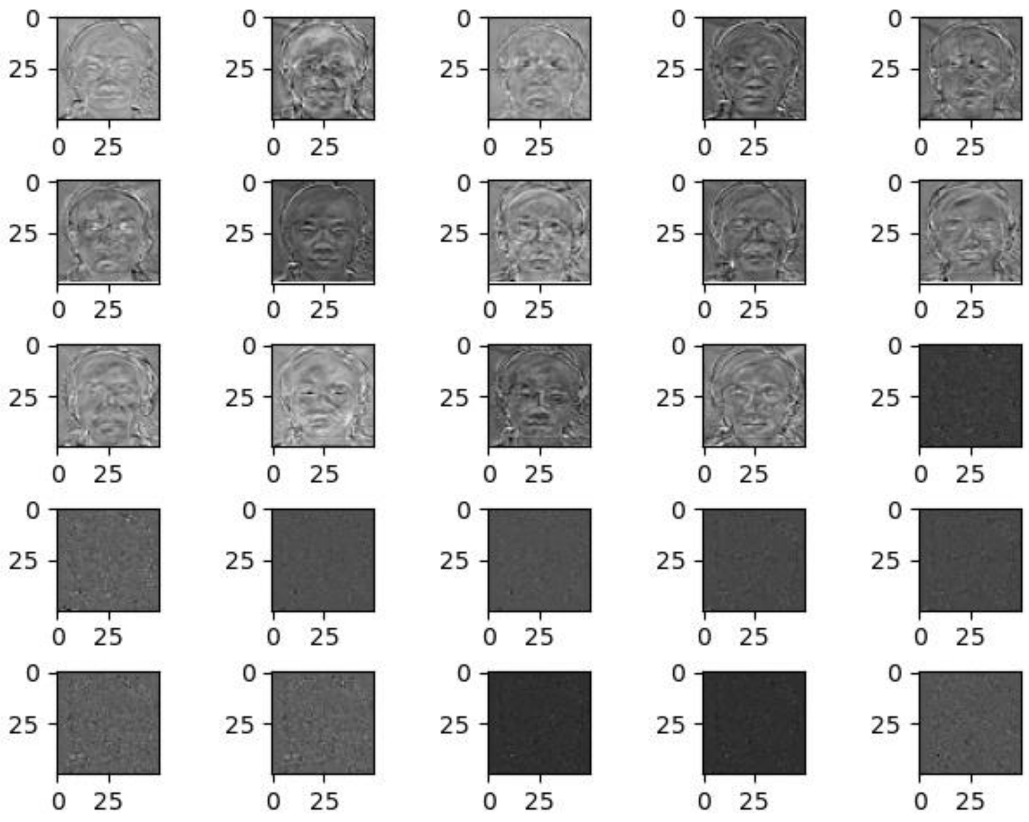
```python
# within-class scatter
sW = np.zeros((numOfAttr, numOfAttr), dtype=np.float64)
for i in range(numOfImages):
    temp = data[:, i].reshape(-1, 1) - clusterMean[:, labels[i]].reshape(-1, 1)
    sW += temp @ temp.T

# between-class scatter
sB = np.zeros((numOfAttr, numOfAttr), dtype=np.float64)
for i in range(numOfClusters):
    temp = clusterMean[:, i].reshape(-1, 1) - mean
    sB += numOfMembers * (temp @ temp.T)

# get the eigenvectors of inv(sW) @ sB as W
eigenvalues, eigenvectors = np.linalg.eig(np.linalg.pinv(sW, hermitian=True) @ sB)
sortedIndex = np.argsort(eigenvalues.real)[::-1]
eigenvalues = eigenvalues[sortedIndex]
eigenvectors = eigenvectors[:, sortedIndex]
return eigenvalues, eigenvectors[:, :25]
```

Then the projection and reconstruction procedures are the same as PCA does.

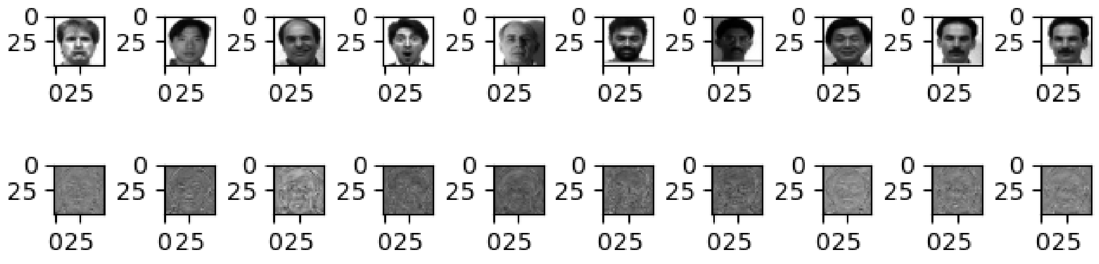


```
# LDA
eigenvalues, eigenvectors = LDA(images, labels)
plotEigenface(eigenvectors, height, width, 'lda')
projectedImages = eigenvectors.T @ images
recoveredImages = eigenvectors @ projectedImages
plotReconstruction(images, recoveredImages, height, width, 'lda')
```

- Eigenfaces



- Reconstruction results

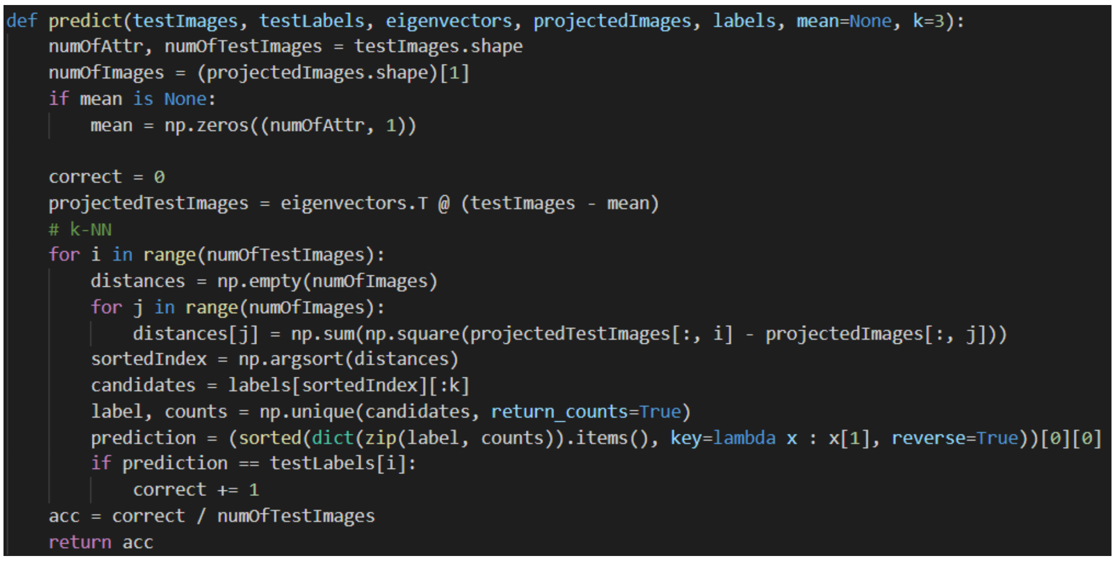


(Upper row: original images; lower row: reconstructed images)

We observe that the eigenfaces can show the facial features, and the fisherfaces can show the illumination and a little bit of the facial features. It can be explained that PCA maintains the maximum variance property but the purpose of LDA is to distinguish the different clusters from the given data. Hence, the recovered images by using PCA is closer to the original ones than using LDA.

(2) 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.

To use k-NN, we need to calculate the pairwise distance between the test image and each training image, and then take the majority label of the first k nearest images as the prediction. We set k as 3 as the default value.



```python
def predict(testImages, testLabels, eigenvectors, projectedImages, labels, mean=None, k=3):
    numOfAttr, numOfTestImages = testImages.shape
    numOfImages = (projectedImages.shape)[1]
    if mean is None:
        mean = np.zeros((numOfAttr, 1))

    correct = 0
    projectedTestImages = eigenvectors.T @ (testImages - mean)
    # k-NN
    for i in range(numOfTestImages):
        distances = np.empty(numOfImages)
        for j in range(numOfImages):
            distances[j] = np.sum(np.square(projectedTestImages[:, i] - projectedImages[:, j]))
        sortedIndex = np.argsort(distances)
        candidates = labels[sortedIndex][:k]
        label, counts = np.unique(candidates, return_counts=True)
        prediction = (sorted(dict(zip(label, counts)).items(), key=lambda x : x[1], reverse=True))[0][0]
        if prediction == testLabels[i]:
            correct += 1
    acc = correct / numOfTestImages
    return acc
```

(k = 3)

PCA

Accuracy: 84.85%

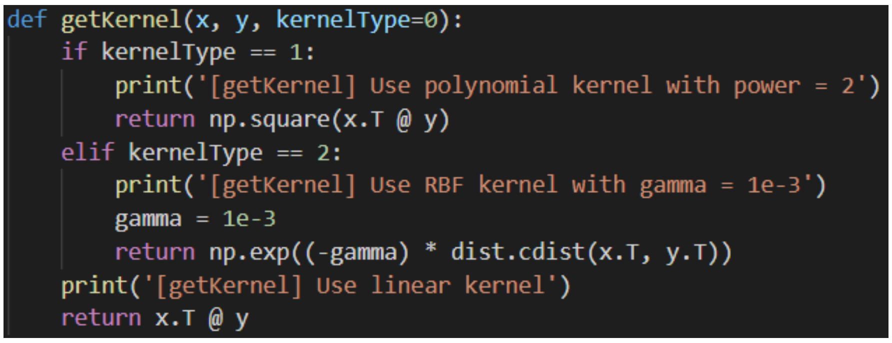
LDA

Accuracy: 93.94%

The accuracy of LDA is better than the one of PCA. It can be explained that LDA maximizes the between-class scatter and minimizes the within-class scatter, which can facilitate the face recognition.

(3) 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.

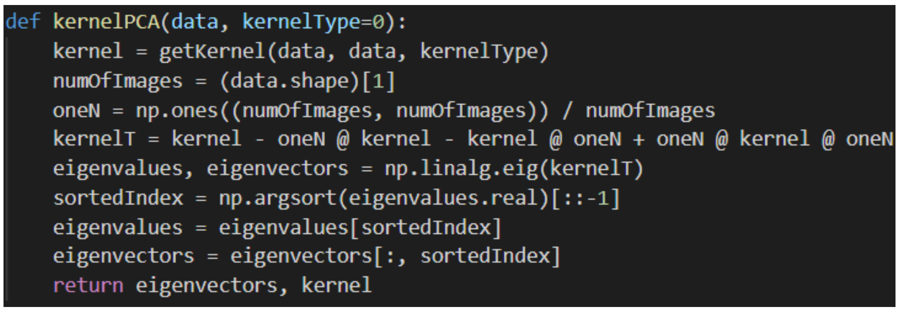
We have implemented three different kernels: linear kernel, polynomial kernel, and RBF kernel. In the following experimental results, we set the degree of the polynomial kernel as 2 and the gamma value in the RBF kernel as 1e-3.



```
def getKernel(x, y, kernelType=0):
    if kernelType == 1:
        print('[getKernel] Use polynomial kernel with power = 2')
        return np.square(x.T @ y)
    elif kernelType == 2:
        print('[getKernel] Use RBF kernel with gamma = 1e-3')
        gamma = 1e-3
        return np.exp((-gamma) * dist.cdist(x.T, y.T))
    print('[getKernel] Use linear kernel')
    return x.T @ y
```

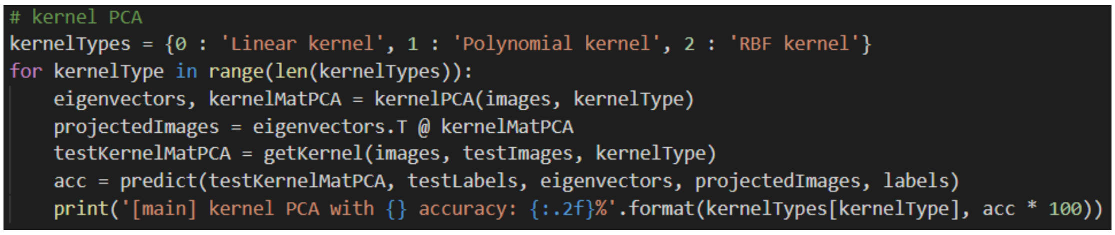
<u>Kernel PCA</u>

We use the formula $K^C = K - \mathbf{1}_N K - K\mathbf{1}_N + \mathbf{1}_N K \mathbf{1}_N$ and then compute its eigenvectors.



```
def kernelPCA(data, kernelType=0):
    kernel = getKernel(data, data, kernelType)
    numOfImages = (data.shape)[1]
    oneN = np.ones((numOfImages, numOfImages)) / numOfImages
    kernelT = kernel - oneN @ kernel - kernel @ oneN + oneN @ kernel @ oneN
    eigenvalues, eigenvectors = np.linalg.eig(kernelT)
    sortedIndex = np.argsort(eigenvalues.real)[::-1]
    eigenvalues = eigenvalues[sortedIndex]
    eigenvectors = eigenvectors[:, sortedIndex]
    return eigenvectors, kernel
```

By obtaining the kernel matrix, we can project the image without explicitly calculating the feature vector and just simply multiply the kernel matrix by the eigenvector matrix. To predict, we need to calculate the kernel value of the test image and the training images.



```
# kernel PCA
kernelTypes = {0 : 'Linear kernel', 1 : 'Polynomial kernel', 2 : 'RBF kernel'}
for kernelType in range(len(kernelTypes)):
    eigenvectors, kernelMatPCA = kernelPCA(images, kernelType)
    projectedImages = eigenvectors.T @ kernelMatPCA
    testKernelMatPCA = getKernel(images, testImages, kernelType)
    acc = predict(testKernelMatPCA, testLabels, eigenvectors, projectedImages, labels)
    print('[main] kernel PCA with {} accuracy: {:.2f}%'.format(kernelTypes[kernelType], acc * 100))
```

Linear kernel accuracy: 72.73%

Polynomial kernel accuracy: 78.79%

RBF kernel accuracy: 84.85%

II. t-SNE

tsne.py and ssne.py

(1) 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).

The main difference is that they use different distributions in the low-dimensional space respectively, and thus the gradient formulas are also different. Symmetric SNE uses Gaussian distribution in low-dimensional space, while t-SNE uses Student-t distribution in low-dimensional space. The low-dimensional pairwise similarities and the corresponding gradient formulas in symmetric SNE and t-SNE are shown below respectively:

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

<u>Symmetric SNE</u>

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

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

<u>t-SNE</u>

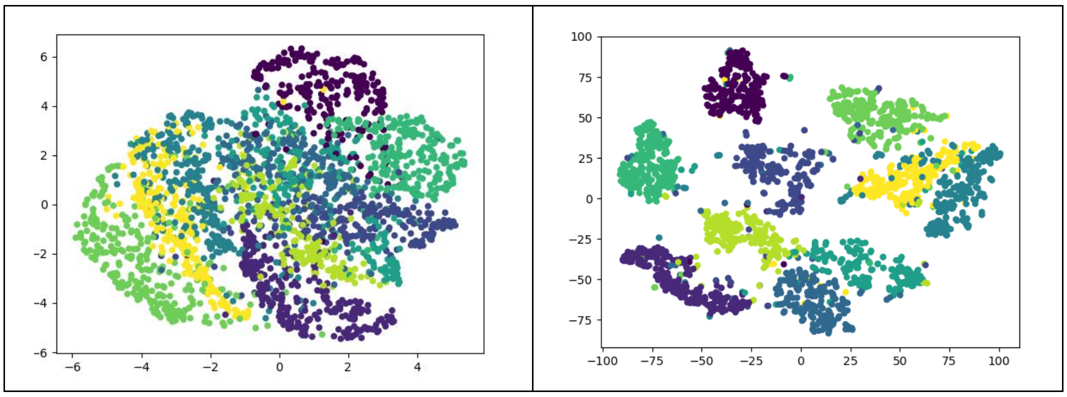
$$q_{ij} = \frac{\left(1 + \left\|y_i - y_j\right\|^2\right)^{-1}}{\sum_{k\neq l}(1 + \|y_k - y_l\|^2)^{-1}}$$

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

The main issue in symmetric SNE is the crowding problem. The following figures show the results of s
. It is obvious the result of symmetric SNE is more crowded than the one of t-SNE.

(Perplexity = 20)

| Symmetric SNE | t-SNE |
|---|---|



As mentioned above, all we need to modify are the low-dimensional pairwise similarity and the gradient formula:

t-SNE

```python
# 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)
```

```python
# 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)
```

Symmetric SNE

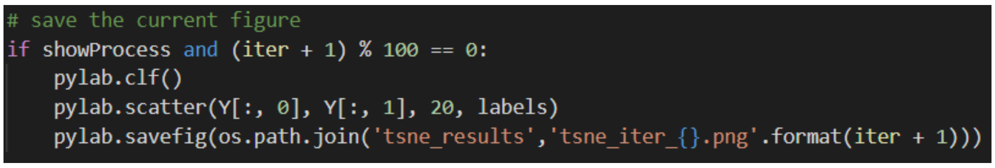
```python
# Compute pairwise affinities
num = np.exp(-1 * dist.cdist(Y, Y, 'sqeuclidean'))
num[range(n), range(n)] = 0.
Q = num / np.sum(num)
Q = np.maximum(Q, 1e-12)
```

```python
# Compute gradient
PQ = P - Q
for i in range(n):
    dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

(2) Visualize the embedding of both t-SNE and symmetric SNE.
Details of the visualization:

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

In order to show the optimization process, we add a parameter *showProcess* for **tsne** and **ssne** functions respectively.



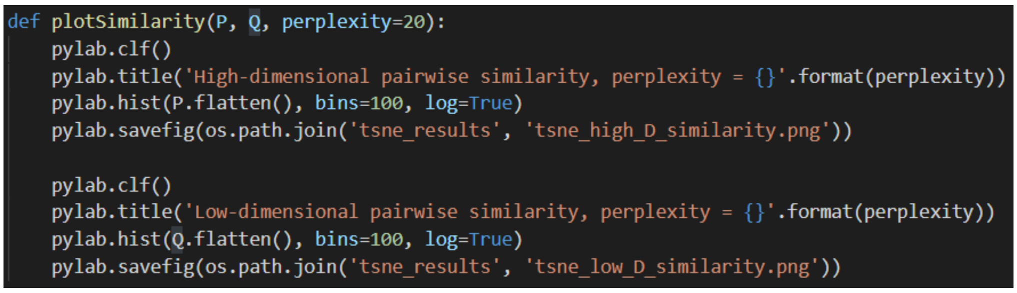
```python
# save the current figure
if showProcess and (iter + 1) % 100 == 0:
    pylab.clf()
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    pylab.savefig(os.path.join('tsne_results','tsne_iter_{}.png'.format(iter + 1)))
```

(Number of iterations = 1000)

The GIF files are **tsne.gif**, and **ssne.gif,** respectively.

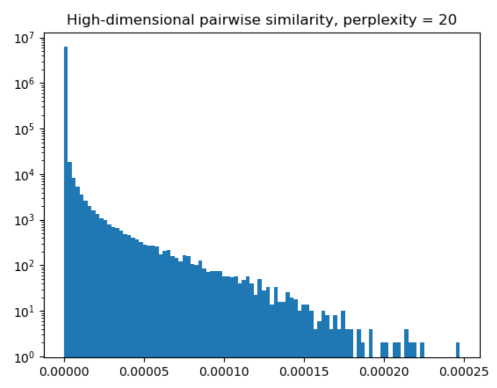
(3) Visualize the distribution of pairwise similarities in both high-dimensional space and low-dimensional space, based on both t-SNE and symmetric SNE.

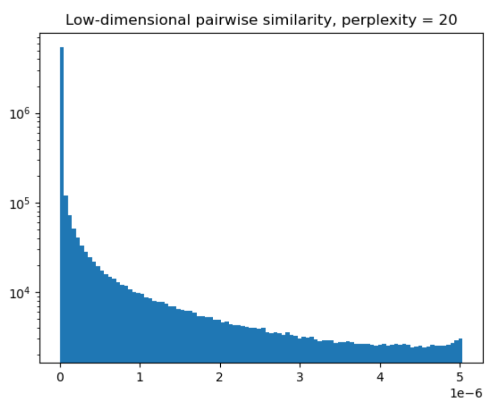
We can get the distribution shape by using the histogram.



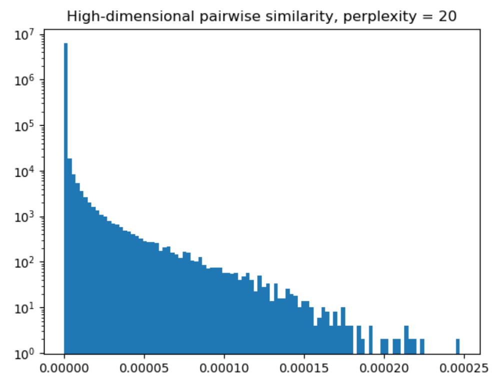
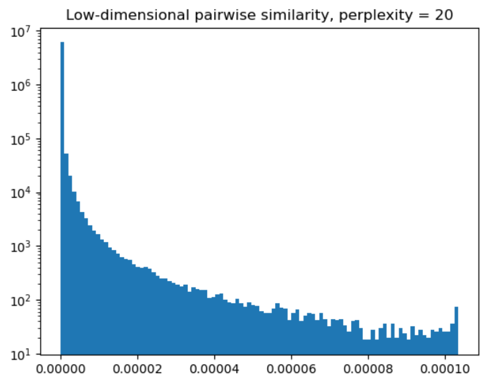
```python
def plotSimilarity(P, Q, perplexity=20):
    pylab.clf()
    pylab.title('High-dimensional pairwise similarity, perplexity = {}'.format(perplexity))
    pylab.hist(P.flatten(), bins=100, log=True)
    pylab.savefig(os.path.join('tsne_results', 'tsne_high_D_similarity.png'))

    pylab.clf()
    pylab.title('Low-dimensional pairwise similarity, perplexity = {}'.format(perplexity))
    pylab.hist(Q.flatten(), bins=100, log=True)
    pylab.savefig(os.path.join('tsne_results', 'tsne_low_D_similarity.png'))
```

<u>Symmetric SNE</u>





<u>t-SNE</u>

| | |
|---|---|
|  |  |

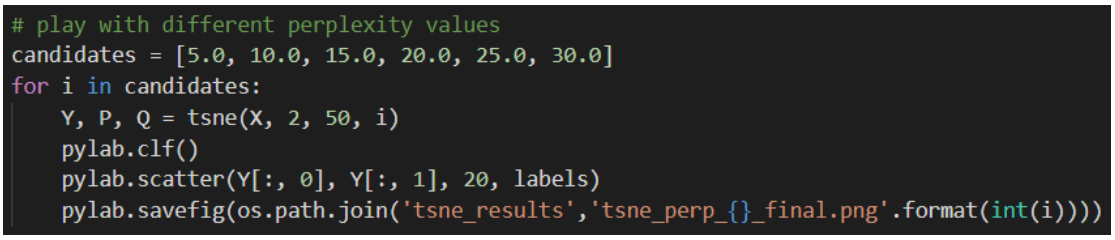
x-axis: distance

y-axis: frequency

We can confirm that t-SNE indeed alleviate the crowding problem in the low-dimensional space by observing the pairwise similarities.

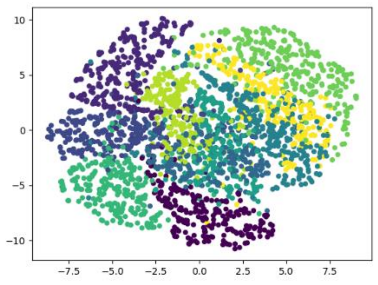
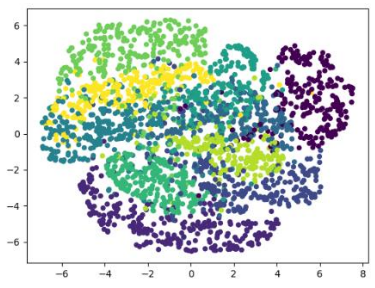
(4) Try to play with different perplexity values. Observe the change in visualization and explain it in the report.

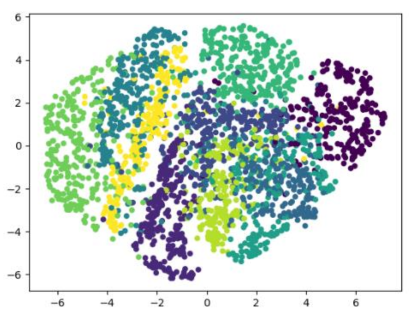
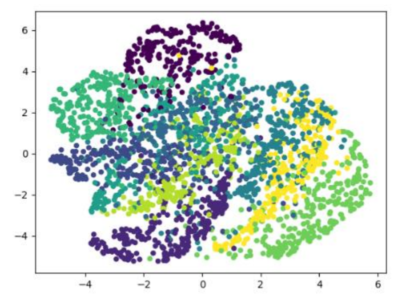
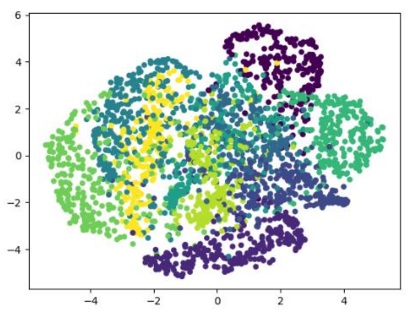
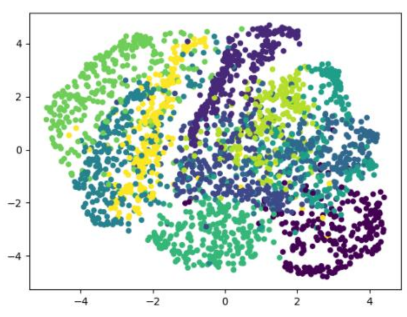
We set perplexity values as 5, 10, 15, 20, 25, and 30, and save the results.



```
# play with different perplexity values
candidates = [5.0, 10.0, 15.0, 20.0, 25.0, 30.0]
for i in candidates:
    Y, P, Q = tsne(X, 2, 50, i)
    pylab.clf()
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    pylab.savefig(os.path.join('tsne_results','tsne_perp_{}_final.png'.format(int(i))))
```

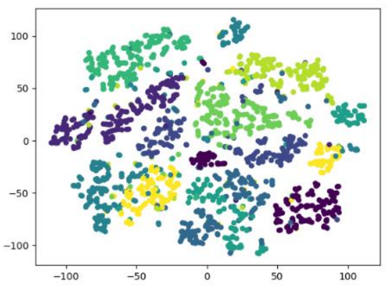
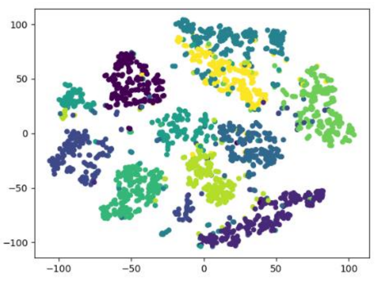
<u>Symmetric SNE</u>

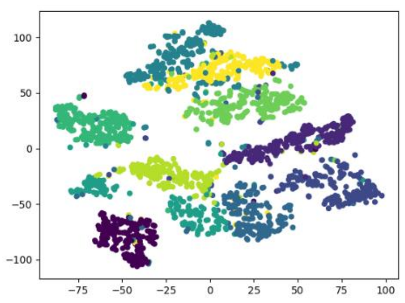
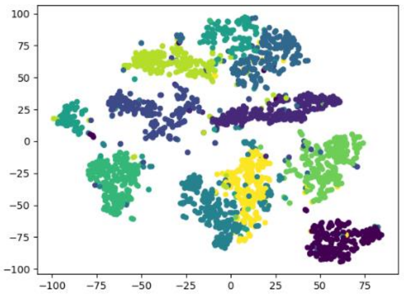
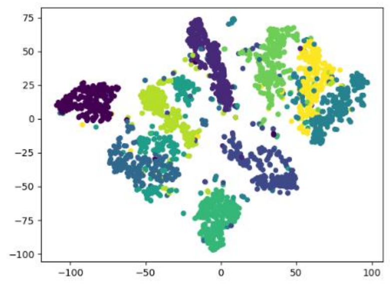
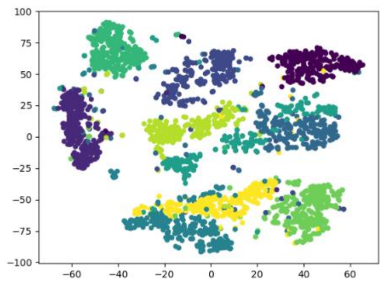
| Perplexity = 5 | Perplexity = 10 |
|---|---|
|  |  |
| Perplexity = 15 | Perplexity = 20 |

| | |
|---|---|
|  |  |
| Perplexity = 25 | Perplexity = 30 |
|  |  |

For symmetric SNE, when we increase the perplexity value, the distance becomes a little bit closer.

<u>t-SNE</u>

| Perplexity = 5 | Perplexity = 10 |
|---|---|
|  |  |
| Perplexity = 15 | Perplexity = 20 |

| | |
|---|---|
|  |  |
| Perplexity = 25 | Perplexity = 30 |
|  |  |

We can easily find that, for t-SNE, the distance becomes wider as the perplexity value increases.