Machine Learning HW7

311551094 資科工碩一 廖昱瑋

I. Code

a. Kernel Eigenfaces

1. Part1

(1) Compress image

Because the dimensionality is too high in original data. Resize image to 77 * 65 first.

```
def resize_img(data):
    num_imgs = data.shape[0]
    img_compress = np.zeros((num_imgs, 77 * 65))

    for img in range(num_imgs):
        for row in range(77):
            for col in range(65):
                tmp = 0
                for i in range(3):
                    for j in range(3):
                        tmp += data[img][(row*3 + i) * 195 + (col*3 + j)]
                img_compress[img][row * 65 + col] = tmp // 9

    return img_compress
```

(2) PCA

Compute the covariance matrix of data. Then, do eigen decomposition of the covariance matrix. Get 25 first largest eigenvectors and normalize them. They are stored as matrix W.

$$\underset{w^1}{\operatorname{argmax}} (w^1)^\top S(w^1) \text{ , subject to } \left\|w^1\right\| = 1$$

```
def PCA(data):
    covariance = np.cov(data.T)
    eigenvalue, eigenvector = np.linalg.eigh(covariance)

    index = np.argsort(-eigenvalue)
    eigenvector = eigenvector[:, index]

    W = eigenvector[:, :25]
    for i in range(W.shape[1]):
        W[:, i] = W[:, i] / np.linalg.norm(W[:, i])

    return W
```

(3) LDA

Compute S_W, S_B by the following formula.

$$S_W = \sum_{j=1}^{k} S_j, \text{ where } S_j = \sum_{i \in C_j} (x_i - \mathbf{m}_j)(x_i - \mathbf{m}_j)^\top \quad S_B = \sum_{j=1}^{k} S_{B_j} = \sum_{j=1}^{k} n_j (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^\top$$

$$\text{and } \mathbf{m}_j = \frac{1}{n_j} \sum_{i \in C_j} x_i \qquad \text{where } \mathbf{m} = \frac{1}{n} \sum x$$

Solve the equation $S_B w_l = \lambda_l S_W w_l$ by doing eigen decomposition on ${S_W}^{-1} S_B$ and get first 25 largest eigenvectors. Note that S_W in un-invertible when $n < D$, so use pseudo inverse on S_W. Normalize the eigenvectors and store them as matrix W.

```python
def LDA(data, label):
    dimension = data.shape[1]
    S_w = np.zeros((dimension, dimension))
    S_b = np.zeros((dimension, dimension))
    m = np.mean(data, axis = 0)

    for subject in trange(1, 16):
        id = np.where(label == subject)
        scatter = data[id]
        mj = np.mean(scatter, axis = 0)
        within_diff = scatter - mj
        S_w += within_diff.T @ within_diff
        between_diff = mj - m
        S_b += 9 * between_diff.T @ between_diff

    S_w_S_b = np.linalg.pinv(S_w) @ S_b
    eigenvalue, eigenvector = np.linalg.eigh(S_w_S_b)

    index = np.argsort(-eigenvalue)
    eigenvector = eigenvector[:, index]

    W = eigenvector[:, :25]
    for i in range(W.shape[1]):
        W[:, i] = W[:, i] / np.linalg.norm(W[:, i])

    return W
```

(4) Eigenfaces & Fisherfaces

Print eigenfaces and fisherfaces by W.

```python
def print_eigen_fisher_face(W, face):
    fig = plt.figure(figsize=(5, 5))
    for i in range(25):
        img = W[:, i].reshape(77, 65)
        ax = fig.add_subplot(5, 5, i+1)
        ax.axis('off')
        ax.imshow(img, cmap='gray')
    fig.savefig(f'./output/eigen_fisher_faces/{face}.jpg')
    plt.show()
```

(5) Reconstruct faces

Randomly choose 10 images and reconstruct images by xWW^T.

```python
def reconstruct_face(W, data, face):
    id = np.random.choice(135, 10, replace=False)
    fig = plt.figure(figsize=(8, 2))
    for i in range(10):
        img = data[id[i]].reshape(77, 65)
        ax = fig.add_subplot(2, 10, i + 1)
        ax.axis('off')
        ax.imshow(img, cmap='gray')

        x = img.reshape(1, -1)
        reconstruct_img = x @ W @ W.T
        reconstruct_img = reconstruct_img.reshape(77, 65)
        ax = fig.add_subplot(2, 10, i + 11)
        ax.axis('off')
        ax.imshow(reconstruct_img, cmap='gray')
    fig.savefig(f'./output/eigen_fisher_faces/reconstruct_{face}.jpg')
    plt.show()
```

2. Part2

Compute coordinates in the space built upon principal components by xW. Then, compute distance between each test coordinates between every train coordinates. Get the prediction by 5 nearest neighbors.

```python
def predict(train_img, train_label, test_img, test_label, W):
    k = 5
    error = 0

    xW_train = train_img @ W
    xW_test = test_img @ W
    for test in range(30):
        distance = np.zeros(135)
        for train in range(135):
            distance[train] = np.sum((xW_test[test] - xW_train[train]) ** 2)
        neighbors = np.argsort(distance)[:k]
        prediction = np.argmax(np.bincount(train_label[neighbors]))
        if test_label[test] != prediction:
            error += 1

    print(f'error rate: {error / 30 * 100}%')
```

3. Part3

(1) Center the data

```python
mean = np.mean(train_img_compress, axis=0)
centered_train = train_img_compress - mean
centered_test = test_img_compress - mean
```

(2) Kernel

Define 2 kernels, linear kernel and RBF kernel.

```python
def linear_kernel(u, v):
    return u @ v.T

def RBFkernel(u, v):
    gamma = 1e-10
    dist = np.sum(u ** 2, axis=1).reshape(-1, 1) + np.sum(v ** 2, axis=1) - 2 * u @ v.T

    return np.exp(-gamma * dist)
```

(3) Kernel PCA

Compute the kernel matrix of data. Then, do eigen decomposition of the kernel matrix to solve $m\lambda\alpha = K\alpha$. Get 25 first largest normalized eigenvectors.

```python
def kernelPCA(data, kernel_type):
    if kernel_type == 'rbf':
        kernel = RBFkernel(data, data)
    elif kernel_type == 'linear':
        kernel = linear_kernel(data, data)
    else:
        print('False kernel type input!')

    eigenvalue, eigenvector = np.linalg.eigh(kernel)

    index = np.argsort(-eigenvalue)
    eigenvector = eigenvector[:, index]

    W = eigenvector[:, :25]
    for i in range(W.shape[1]):
        W[:, i] = W[:, i] / np.linalg.norm(W[:, i])

    return W, kernel
```

(4) Kernel LDA

Compute the kernel matrix of data. Compute S_W, S_B by the following formula.

$$S_B^\Phi = \sum_{i=1}^{c} l_i \boldsymbol{\mu}_i^\Phi (\boldsymbol{\mu}_i^\Phi)^T, \quad S_W^\Phi = \sum_{i=1}^{c} \sum_{j=1}^{l_i} \Phi(\mathbf{x}_{ij}) \Phi(\mathbf{x}_{ij})^T$$

Then, do eigen decomposition of the kernel matrix to solve $\lambda KK\alpha = KZK\alpha$. Get 25 first largest normalized eigenvectors.

```
def kernelLDA(data, kernel_type):
    Z = np.ones((data.shape[0], data.shape[0])) / 9

    if kernel_type == 'rbf':
        kernel = RBFkernel(data, data)
    elif kernel_type == 'linear':
        kernel = linear_kernel(data, data)
    else:
        print('False kernel type input!')

    S_w = kernel @ kernel
    S_b = kernel @ Z @ kernel

    S_w_S_b = np.linalg.pinv(S_w) @ S_b
    eigenvalue, eigenvector = np.linalg.eigh(S_w_S_b)

    index = np.argsort(-eigenvalue)
    eigenvector = eigenvector[:, index]

    W = eigenvector[:, :25]
    for i in range(W.shape[1]):
        W[:, i] = W[:, i] / np.linalg.norm(W[:, i])

    return W, kernel
```

(5) Compute performance

Compute kernel matrix of test data. Then, compute coordinates in the space built upon principal components by xW. Get the prediction by 5 nearest neighbors.

```
def predict_kernel(train_img, train_label, test_img, test_label, W, train_kernel, kernel_type):
    k = 5
    error = 0

    if kernel_type == 'rbf':
        test_kernel = RBFkernel(test_img, train_img)
    elif kernel_type == 'linear':
        test_kernel = linear_kernel(test_img, train_img)
    else:
        print('False kernel type input!')

    xW_train = train_kernel @ W
    xW_test = test_kernel @ W

    for test in range(30):
        distance = np.zeros(135)
        for train in range(135):
            distance[train] = np.sum((xW_test[test] - xW_train[train]) ** 2)
        neighbors = np.argsort(distance)[:k]
        prediction = np.argmax(np.bincount(train_label[neighbors]))
        if test_label[test] != prediction:
            error += 1

    print(f'error rate: {error / 30 * 100}%')
```

b. t-SNE

1. Part1

   t-SNE:

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

   Symmetric SNE:

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

```
num = np.exp(-np.add(np.add(num, sum_Y).T, sum_Y))
dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

2. Part2

   Save embedding images every 40 iterations.

```
if (iter + 1) % 40 == 0:
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    if mode == 1:
        pylab.savefig(f'../output/SNE/t_SNE/perplexity{int(perplexity)}/iteration{iter+1:04d}.png')
    else:
        pylab.savefig(f'../output/SNE/symmetric/perplexity{int(perplexity)}/iteration{iter+1:04d}.png')
    pylab.cla()
```

3. Part3

   Use histogram to plot pairwise similarities in both high-dimensional space and low-dimensional space.

```
def plot_similarity(P, Q, mode, perplexity):
    pylab.hist(P.flatten(),bins = 30,log = True)
    pylab.savefig(f'../output/SNE/{mode}/perplexity{int(perplexity)}/high_D.png')
    pylab.cla()
    pylab.hist(Q.flatten(),bins = 30,log = True)
    pylab.savefig(f'../output/SNE/{mode}/perplexity{int(perplexity)}/low_D.png')
```

4. Part4

   Try different perplexity values with 5, 20, 35, 50, 65.

```
PERPLEXITY = [5, 20, 35, 50, 65]
```

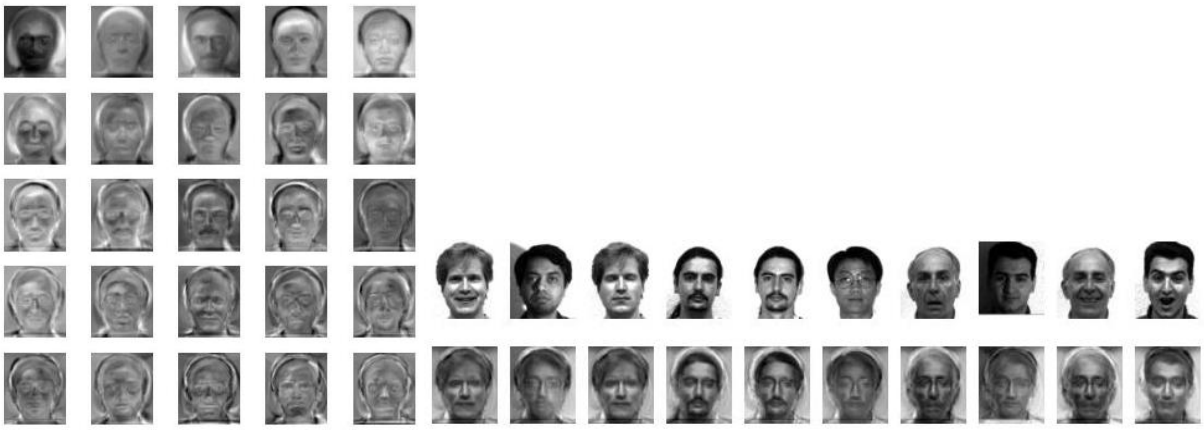
```
for perplexity in PERPLEXITY:
    print(f'Running t-SNE with perlexity {perplexity}.')
    sne(X, labels, 2, 50, perplexity, mode)
```

II. Result & Discussion

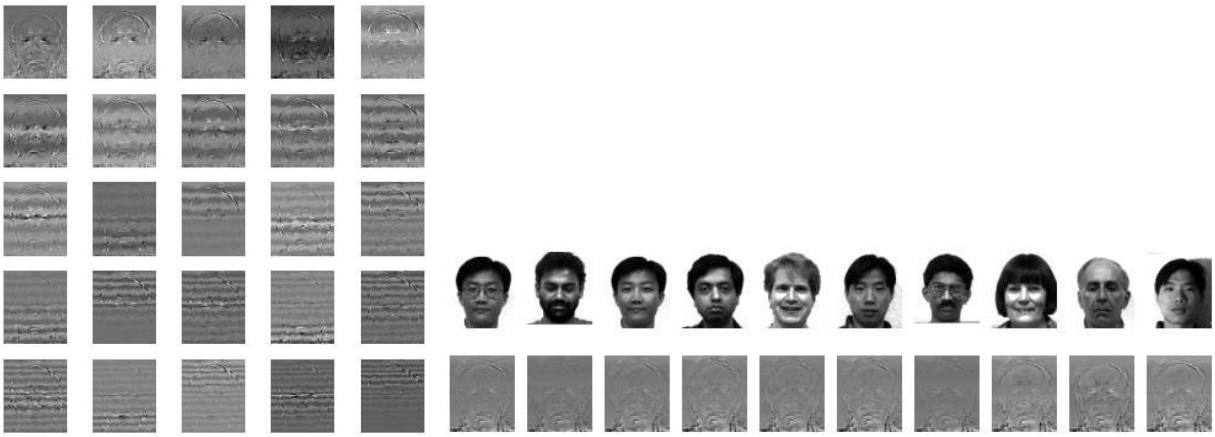
a. Kernel Eigenfaces

1. Part1

Eigenfaces and its reconstruct faces:



Fisherfaces and its reconstruct faces:



Eigenfaces and its reconstruct faces provide a good contour of original face. However, fisherfaces and its reconstruct faces are quite blurred. It's because dimensionality of lower-dimension space should be smaller than number of classes. We have 15 classes in the dataset but set dimensionality of lower-dimension space as 25. Therefore, the remaining fisherfaces are literally grey.

2. Part2

Algorithm	Error rate
PCA	10.00%
LDA	16.67%

3. Part3

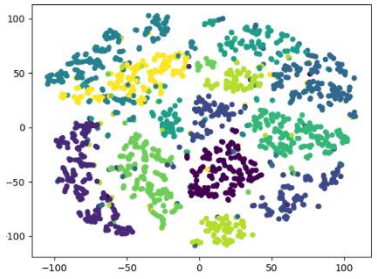
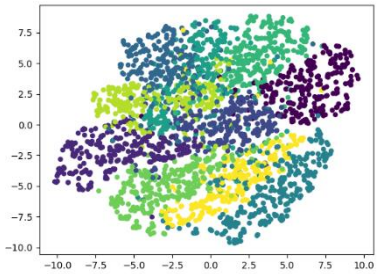
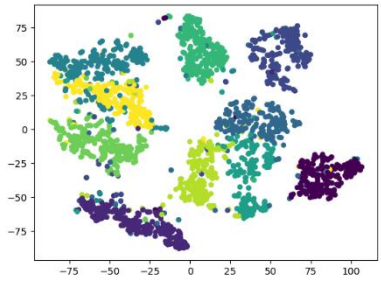
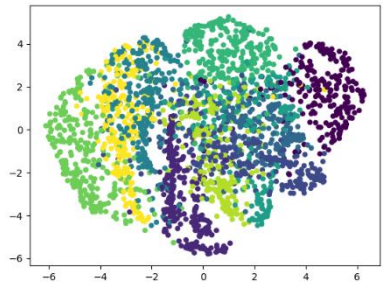
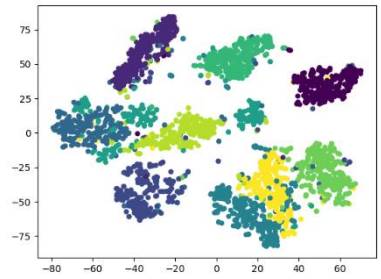
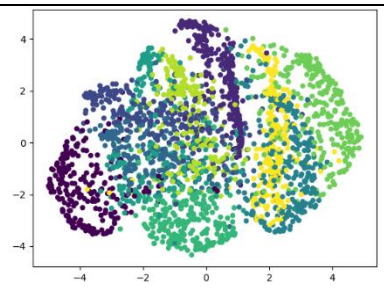
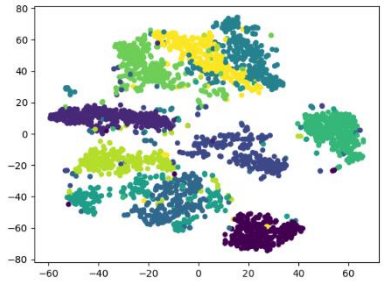
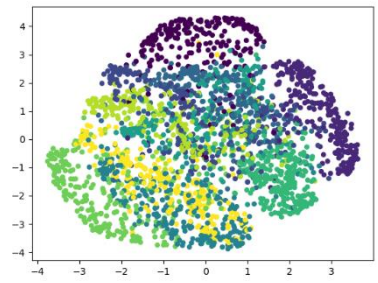
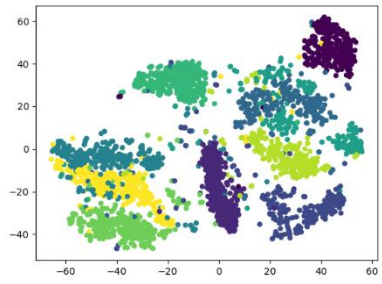
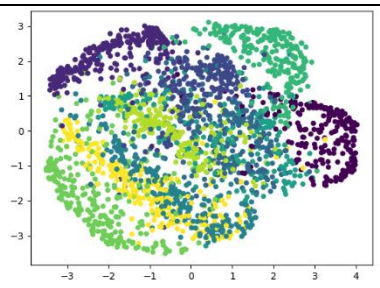
Algorithm		Error rate
PCA	Linear kernel	16.67%
	RBF kernel	20.00%
LDA	Linear kernel	23.33%
	RBF kernel	26.67%

Roughly, PCA performs better than LDA because dimensionality of lower-dimension space we

choose in LDA is not adequate. Surprisingly, kernel method doesn't perform better than original PCA and LDA. In my opinion, it's because kernel trick is used to extract nonlinear features but the dimensionality in original dataset is high enough. Therefore, we don't need to project data to higher-dimension space by kernel trick. In this experiment, linear kernel performs better than RBF kernel.

b. t-SNE

1. Project data onto 2D space

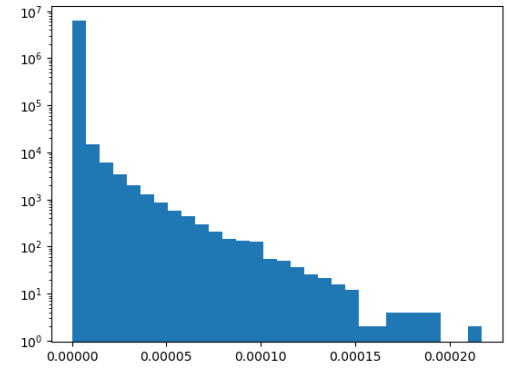
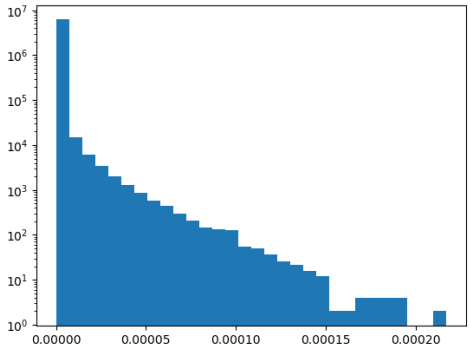
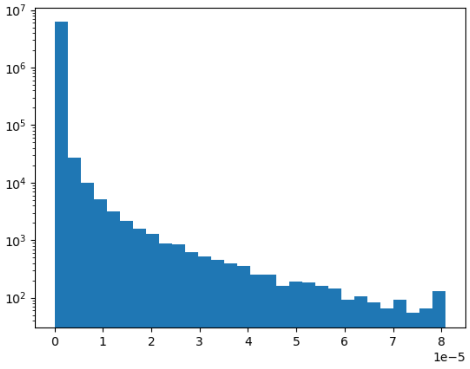
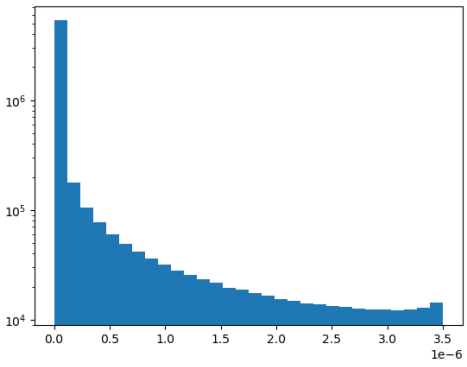
Perplexity	t-SNE	Symmetric SNE
5		
20		
35		
50		
65		

We can see that symmetric SNE has crowded problem which t-SNE doesn't have. If perplexity

is small, the embedding will tend to be fluffy because repulsive forces will dominate. If perplexity is large, the embedding will tend to be denser structure.

2. Pairwise similarities in high-D low-D space

Result of pairwise similarities in high-D low-D space with perplexity set as 35.

Dimension	t-SNE	Symmetric SNE
High-D		
Low-D		

The pairwise similarity of t-SNE in low-D is range from 0 to 8e-5 and the pairwise similarity of symmetric SNE in low-D is range from 0 to 3.5e-6. Therefore, points do not need to be too far to achieve low probability in symmetric SNE which may lead to crowded problem.

III. Discussion

a. The meaning of eigenfaces

Eigenfaces are a specific case of doing PCA on human faces. In PCA, we want to preserve maximum the variance when doing projection. It allows us to drop unnecessary information by selecting only the principal components. The principal components are all uncorrelated and we get fewer vectors than the whole space that allow us to reconstruct any possible face.

b. The crowded problem of symmetric SNE

In symmetric SNE, we always use Gaussian to transform Euclidean distance into probability, no matter in high-dimensional or low-dimensional space. However, small probability can be achieved by using not-so-far distance in Gaussian. We may have two data that are far away in high-dimensional space but the corresponding data in low-dimensional space are not-so-far.

c. Anything you want to discuss

1. The value of gamma in RBF kernel is important. Inadequate value may lead to bad result.
2. The larger the perplexity is, the quicklier the pairwise similarities converge.
3. Symmetric SNE converges quicklier than t-SNE.
4. Perplexity has greater effect on t-SNE than symmetric SNE because symmetric SNE has crowded problem.