

# HW7 311552055 蕭育泓

## Kernel Eigenfaces

### Part a. code explanations

#### Main function

```
if __name__ == "__main__":  
    print("loading...")  
    train_data, train_label, train_name = load("./Yale_Face_Database/Training/")  
    test_data, test_label, test_name = load("./Yale_Face_Database/Testing/")  
  
    kernel_mode = ["simple", "polynomial", "RBF"]  
    # kernel_mode = ["RBF"]  
  
    for i in kernel_mode:  
        print("PCA...")  
        PCA(train_data, train_label, test_data, test_label, i)  
        print("LDA...")  
        LDA(train_data, train_label, test_data, test_label, i)
```

#### Load data

First, loading the data and resizing the picture ( $231 * 195$ ) to the picture ( $21 * 15$ ) by using the mean of the picture pixel. And then resize the picture again (resize  $21 * 15$  to  $315$ ). Finally, we will get the all training data ( $135 * 315$ ) and return it.

```
def load(path):
    data = []
    label = []
    name=[]
    allFileList = os.listdir(path)
    for file in allFileList:
        filename = f'{path}{file}'
        name.append(filename)
        image = Image.open(filename)
        data_buf = np.array(image)
        # print(data_buf)
        data_resize = np.zeros((21,15))
        for i in range(data_buf.shape[0]):
            for j in range(data_buf.shape[1]):
                data_resize[int(i/11)][int(j/13)] += data_buf[i][j]
        for i in range(data_resize.shape[0]):
            for j in range(data_resize.shape[1]):
                data_resize[i][j] = int(data_resize[i][j]/143)

        data_resize = data_resize.reshape(1,-1)
        data.append(data_resize[0])
        label.append(int(re.findall('\d+', filename)[0]))

    return np.array(data),np.array(label),np.array(name)
```

## Part 1:

### PCA function

For part1, I used the kernel\_type = “simple” to implement the kernel linear. First, I standardize the training data and do the select kernel. Using the kernel to get the first 25 eigenfaces.

**orthogonal projection  $W$**  is composed of  
 $k$  first largest eigenvectors of covariance matrix of  $x$   
**principal components**

For the reconstruction, I using the formula below.

$xWW^T$  **linear combination of principal components**  
**which tries to reconstruct original  $x$**

Because of the training data is standardization, so I need to adjust my training data for fit the original real type.

```
def PCA(train_data, train_label, test_data, test_label, kernel_type):
    mean = np.mean(train_data, axis=0)

    if kernel_type == "polynomial":
        gram = polynomial_kernel((train_data - mean)/np.std(train_data), 5, 10, 2)

    if kernel_type == "RBF":
        gram = rbf_kernel((train_data - mean)/np.std(train_data), (train_data - mean)/np.std(train_data), 0.01)

    if kernel_type == "simple":
        gram = (((train_data - mean)/np.std(train_data)).T @ (train_data - mean)/np.std(train_data))

    eigenvalue, eigenvector = np.linalg.eig(gram)
    targetIndex = np.argsort(-eigenvalue)
    eigenvalue = eigenvalue[targetIndex]
    eigenvector = eigenvector[:,targetIndex]
    eigenvector = eigenvector.real
    eigenvector = eigenvector[:, :25]
```

```
for i in range(25):
    plt.figure("PCA 25 eigenfaces")
    plt.subplot(5, 5, i+1)
    plt.axis("off")
    plt.imshow(eigenvector[:,i].reshape(21,15), cmap="gray")
plt.show()
reconstruct_num = np.random.choice(train_data.shape[0],10, replace=False)
reconstruct = np.zeros((10, 315)) # 45045 = 231 * 195
for i in range(10):
    reconstruct[i,:] = (train_data[reconstruct_num[i],:] - mean) @ eigenvector @ eigenvector.T + mean
for i in range(10):
    plt.figure("PCA 10 reconstruction.")
    # Original image.
    plt.subplot(2, 10, i + 1)
    plt.axis('off')
    plt.imshow(train_data[reconstruct_num[i], :].reshape((21, 15)), cmap='gray')

    # Reconstructed image.
    plt.subplot(2, 10, i + 11)
    plt.axis('off')
    plt.imshow(reconstruct[i, :].reshape((21, 15)), cmap='gray')
plt.show()
acc = prediction(train_data, train_label, test_data, test_label, eigenvector, mean)
print(f"{kernel_type} PCA accuracy : {acc}")
```

## LDA function

For LDA, some concepts are as same as PCA function.

The different between two functions is only about the gram matrix generate. In the LDA, we need to using the standardize training data to do the formula below and find out the SW and SB.

between-class scatter:

$$S_B = \sum_{j=1}^k S_{B_j} = \sum_{j=1}^k n_j (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^\top$$

where  $\mathbf{m} = \frac{1}{n} \sum x$

within-class scatter:  $S_W = \sum_{j=1}^k S_j$ , where  $S_j = \sum_{i \in \mathcal{C}_j} (x_i - \mathbf{m}_j)(x_i - \mathbf{m}_j)^\top$

and  $\mathbf{m}_j = \frac{1}{n_j} \sum_{i \in \mathcal{C}_j} x_i$

And using the below formula to calculate the gram matrix.

get first  $q$  largest eigenvectors of  $S_W^{-1} S_B$  as  $W$

But because my data resize type is different with the class

sample, so I need to swap the function  $S_W^{-1} S_B$  to  $S_B S_W^{-1}$ .

```
def LDA(train_data, train_label, test_data, test_label, kernel_type):
    mean = np.mean(train_data, axis=0)
    training = copy.deepcopy(train_data)
    training = (train_data - mean)/np.std(train_data)
    mean_Standard = np.mean(training, axis=0)

    count = len(np.unique(train_label))
    SW = np.zeros((training.shape[1], training.shape[1]))
    SB = np.zeros((training.shape[1], training.shape[1]))

    for i in range(count): # 15
        Xi = training[np.where(train_label == i+1)[0], :]

        mj = np.mean(Xi, axis=0)

        if kernel_type == "simple":
            SW += (Xi - mj).T @ (Xi - mj)
            SB += 9 * (mj - mean_Standard).reshape(-1,1) @ (mj - mean_Standard).reshape(-1,1).T
        if kernel_type == "polynomial":
            SW += polynomial_kernel((Xi - mj), 5, 10, 2)
            SB += 9 * polynomial_kernel((mj - mean_Standard).reshape(-1,1).T, 5, 10, 2)
        if kernel_type == "RBF":
            SW += rbf_kernel((Xi - mj), (Xi - mj), 0.01)
            SB += 9 * rbf_kernel((mj - mean_Standard).reshape(-1,1).T, (mj - mean_Standard).reshape(-1,1).T, 0.01)

    gram = SB @ np.linalg.inv(SW)
```

The remaining parts are as same as PCA function.

(include eigenvector generate and reconstruct part)

## Part 2:

### Function prediction

Using the k nearest neighbor to classify the testing image belongs to. I set the  $k = 3$  to find the class. Find out the distance between the projection of training data and the projection of testing data. The k smallest distance will be select to be which cluster and predict.

```
def prediction(train_data, train_label, test_data, test_label, eigenvector, mean):
    k = 3
    acc = 0

    train_proj = (train_data - mean) @ eigenvector
    test_proj = (test_data - mean) @ eigenvector
    distance = np.zeros(135)
    for testIndex, test in enumerate(test_proj):
        for trainIndex, train in enumerate(train_proj):
            distance[trainIndex] = np.linalg.norm(test - train)
        minDistances = np.argsort(distance)[:k]
        predict = np.argmax(np.bincount(train_label[minDistances]))
        if predict == test_label[testIndex]:
            acc += 1
    return acc/test_label.shape[0]
```

## Part 3:

In the function PCA and the function LDA, both of them, I select the kernel RBF and kernel polynomial.

## Function polynomial\_kernel

First, I will find out the kernel K by using the formula from previous homework. And using the formula below to get the Kernel PCA.

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

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

```
def polynomial_kernel(x, gamma, coef, degree):
    K = np.power(gamma * (x.T @ x) + coef, degree)
    N = np.ones((x.T.shape[0], x.T.shape[0]))/x.shape[0] #1N is NxN matrix with every element 1/N
    KC = K - N @ K - K @ N + N @ K @ N
    return KC
```

## Function rbf\_kernel

The introduce is same as Function polynomial\_kernel.

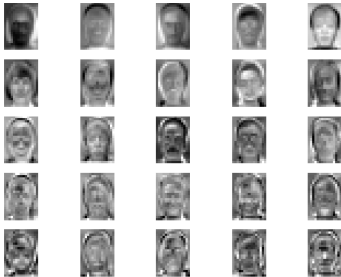
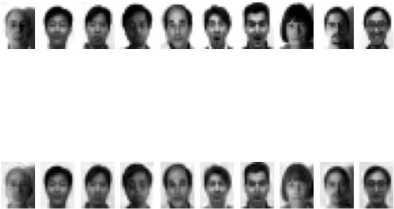
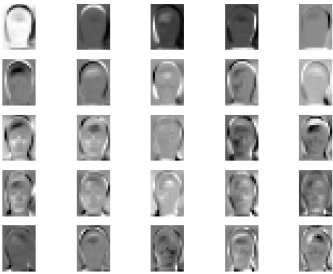
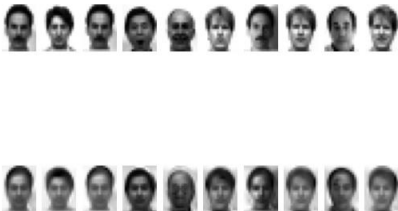


```
def rbf_kernel(x, y, gamma):
    K = np.exp(-gamma * cdist(x.T, y.T, 'sqeuclidean'))
    # print(K.shape)
    N = np.ones((x.T.shape[0], x.T.shape[0]))/x.T.shape[0] #1N is NxN matrix with every element 1/N
    KC = K - N @ K - K @ N + N @ K @ N
    return KC
```

In PCA, I will return the KC to the original function to be the gram matrix.

In LDA, I will use kernel to calculate the SW and SB.

Part b. Result




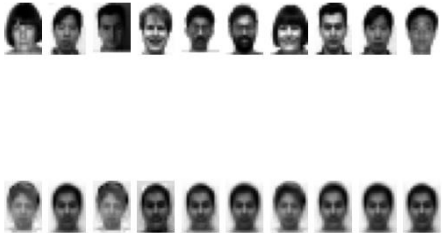

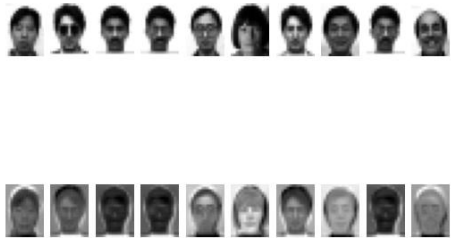
PCA

kernel	Eigenfaces	reconstruction.
simple		
Poly-nomial		
RBF		

Performance:

```
loading...
PCA...
simple PCA accuracy : 0.8333333333333334
PCA...
polynomial PCA accuracy : 0.8333333333333334
PCA...
RBF PCA accuracy : 0.8666666666666667
```

## LDA

kernel	Fisherfaces	reconstruction.
simple		
Poly-nomial		
RBF		

Performance:

```
loading...
LDA...
simple LDA accuracy : 0.8666666666666667
LDA...
polynomial LDA accuracy : 0.7
LDA...
RBF LDA accuracy : 0.9
```



Observing in the part3:

1. The reconstruct images have the better resolution in PCA, which is more clear.
2. Both of them are having the better performance in RBF kernel.
3. The kernel polynomial in LDA, the reconstructions are similar.
4. The performance in polynomial kernel is the worst in both of them.
5. The simple performance is well in both of them.

# t-SNE

## Part a. code explanations

### Part 1:

To modify the code a little bit and make it back to symmetric SNE. The difference of the s-SNE and t-SNE are about the q and gradient.

In the t-SNE, we calculate the q and gradient by the formula below.

$$p_{ij} = \frac{\exp(-\|x_i - x_j\|^2 / (2\sigma^2))}{\sum_{k \neq l} \exp(-\|x_l - x_k\|^2 / (2\sigma^2))}$$
$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|y_i - y_j\|^2)^{-1}}$$

gradient:

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

In the s-SNE, we calculate the q and gradient by the formula

below.

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

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

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

Calculate the different q :

```
if mode == "t-sne":
    num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))

if mode == "s-sne":
    num = np.exp((-1) * np.add(np.add(num, sum_Y).T, sum_Y))
```

Calculate the different gradient :

```
if mode == "t-sne":
    dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)

if mode == "s-sne":
    dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

I select the mode to control, which SNE is doing.

```
modes = ["t-sne", "s-sne"]
perplexitys = [10.0, 20.0, 30.0, 40.0, 50.0]
for mode in modes:
    for perplexity in perplexitys:
        Y, gif_pic = sne(labels, X, 2, 50, perplexity, mode)
        make_gif(gif_pic, mode, perplexity)
```

## Part 2:

Visualize the embedding of both t-SNE and symmetric SNE.

I will store the image of each 10 iterations and using the function `make_gif` to make the .gif.

```
# 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))
    pylab.clf()
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    if mode == "t-sne":
        pylab.xlim([-100, 100])
        pylab.ylim([-100, 100])
    if mode == "s-sne":
        pylab.xlim([-25, 25])
        pylab.ylim([-25, 25])
    pylab.tight_layout()
    canvas = pylab.get_current_fig_manager().canvas
    canvas.draw()
    gif_pic.append(Image.frombytes('RGB', canvas.get_width_height(), canvas.tostring_rgb()))

# Stop lying about P-values
if iter == 100:
    P = P / 4.
gif_pic[-1].save(f'./tsne_python/{mode}_{perplexity}_pic.png')

def make_gif(gif_pic, mode, perplexity):
    gif_pic[0].save(f'./tsne_python/{mode}_{perplexity}.gif', format='GIF',
        append_images = gif_pic[1:], save_all=True, duration=100, loop=0)
    return
```

## Part 3:

Using function `drawSimilarities` to show the high-dimensional space and low-dimensional space. The  $p$  and  $q$  are using the last iteration result.

```
def drawSimilarities(p, q, labels, mode, perplexity):
    # Get sorted index.
    index = np.argsort(labels)
    plt.clf()
    plt.figure(1)

    # Plot p.
    log_p = np.log(p)
    sorted_p = log_p[index][:, index]
    plt.subplot(121)
    img = plt.imshow(sorted_p, cmap='gray', vmin=np.min(log_p), vmax=np.max(log_p))
    plt.colorbar(img)
    plt.title('High dim space')

    # Plot q.
    log_q = np.log(q)
    sorted_q = log_q[index][:, index]
    plt.subplot(122)
    img = plt.imshow(sorted_q, cmap='gray', vmin=np.min(log_q), vmax=np.max(log_q))
    plt.colorbar(img)
    plt.title('Low dim space')
    plt.show()
```

## Part 4:

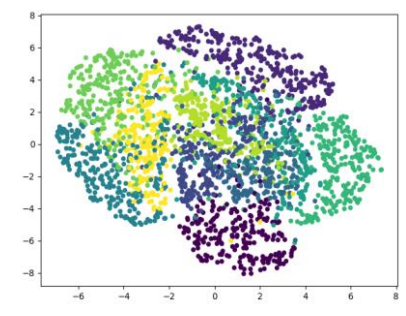
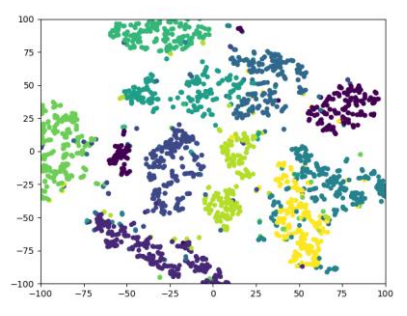
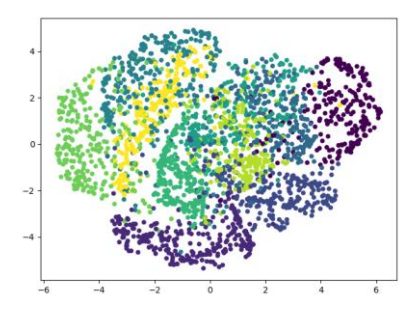
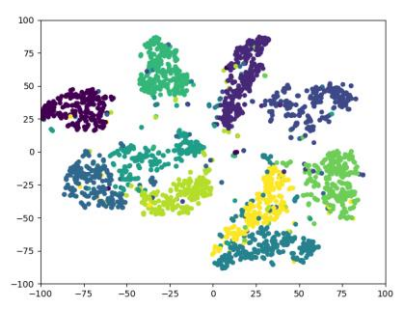
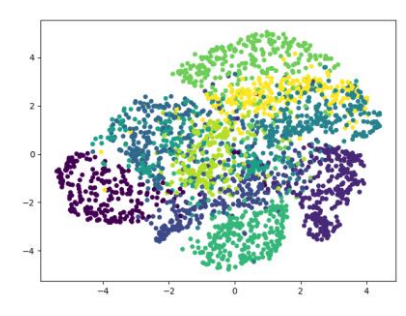
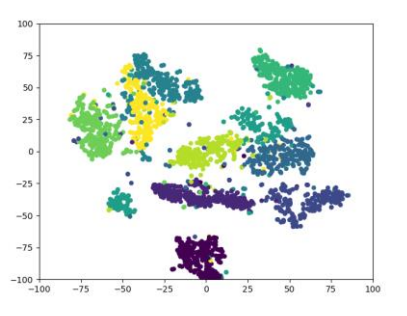
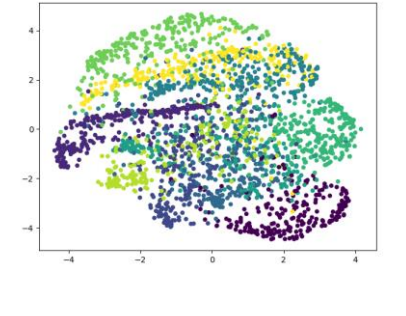
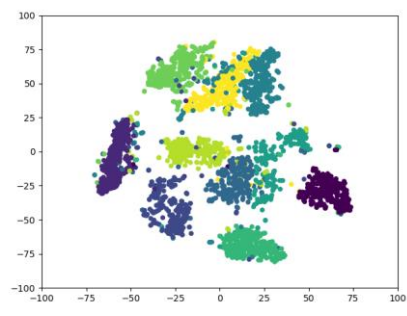
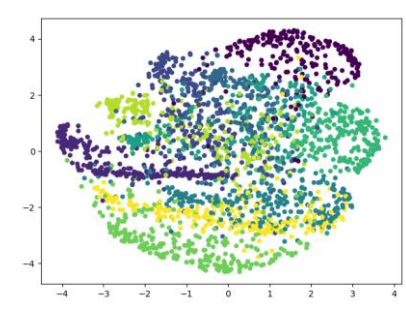
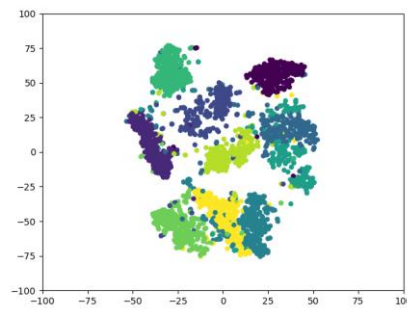
I Try to play with different perplexity values with

[10 20 30 40 50] for the input.

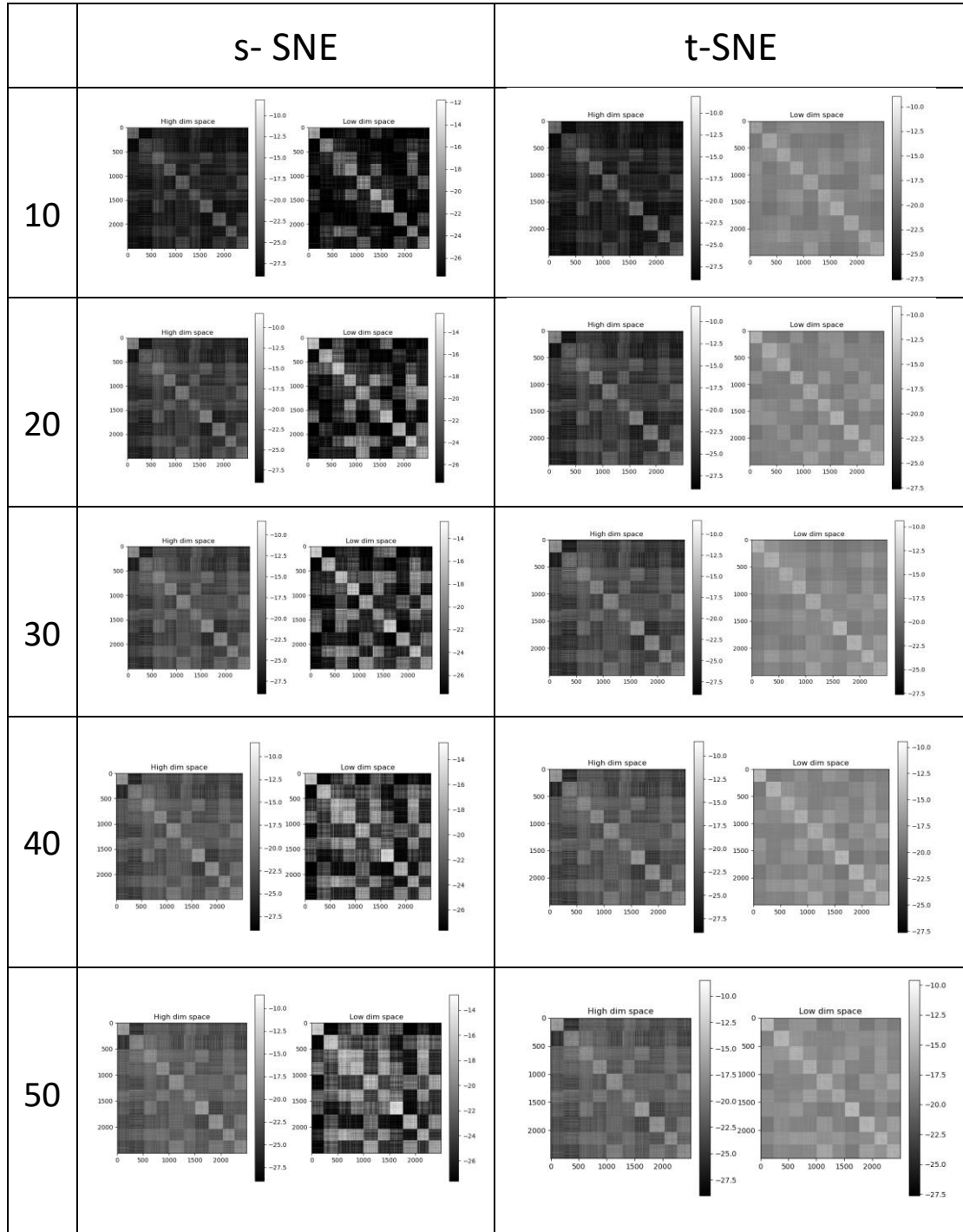
```
modes = ["t-sne", "s-sne"]
perplexitys = [10.0, 20.0, 30.0, 40.0, 50.0]
for mode in modes:
    for perplexity in perplexitys:
        Y, gif_pic = sne(labels, X, 2, 50, perplexity, mode)
        make_gif(gif_pic, mode, perplexity)
```

## Part b. Result

Visualize the embedding of both t-SNE and symmetric SNE.

perplexity	s- SNE	t-SNE
10		
20		
30		
40		
50		

## Visualize the distribution of pairwise similarities



Observing in the part1:

1. Both will have better performance after the 100 iteration and error will drop down from around 20 to about 2
2. Training speed in s-SNE is faster than t-SNE.

Observing in the part4:

1. Using the bigger perplexity will get the all data closer.
2. Using the bigger perplexity will get each clusters closer.
3. Using the bigger perplexity will get smaller group classify.

## Part c. observations and discussion

1. Eigenfaces is the set of eigenvectors.
2. The meaning of eigenfaces, I think is going to get the weight to know which places are more important.
3. The eigenfaces is more clear than fisherfaces.
4. The reconstruct data will similar to the original image.
5. If we did not resize to the small size, the calculate time will be horrible.



6. The result of the s-SNE is more crowded than t-SNE and mix together.
7. t-SNE can classify better.
8. The range of pairwise similarities are smaller in t-SNE, so that s-SNE has more crowded problems.