ML_HW6_report

309554027 鍾弈言

A. code with detailed explanations (20%)

a. Kernel Eigenfaces

1. main.py

I process all input parameters and read the data in main.py read_dataset

```
def read_dataset(mode: str) -> Tuple[np.ndarray, np.ndarray]:
    """
    Read the data according to the corresponding mode, compress the data into np.ndarray and return.
    """
    path = f'./Yale_Face_Database/{mode}'
    files = os.listdir(path)
    images = np.zeros((len(files), config.WIDTH * config.HEIGHT))
    labels = np.zeros(len(files), dtype=int)
    for idx, file in enumerate(files):
        image = Image.open(f'{path}/{file}')
        images[idx] = np.asarray(image.resize(
            (config.WIDTH, config.HEIGHT))).flatten()
        # record the person's index
        labels[idx] = int(file[7:9])

    return images, labels
```

2. kernel_eigenfaces() in kernel_eigenfaces.py

main function in kernel_eigenfaces.py, check should run PCA or LDA.

```
def kernel_eigenfaces(train_images: np.ndarray, train_labels: np.ndarray, test_images: np.ndarray,
                      test_labels: np.ndarray, method: int, mode: int, num_k: int, kernel_type: int, gamma: float) -> None:
    if method == 0:
        # PCA
        print(f'\n===== Compute PCA start =====')
        computePCA(
            train_images,
            train_labels,
            test_images,
            test_labels,
            mode,
            num_k,
            kernel_type,
            gamma)
    else:
        # LDA
        print(f'\n===== Compute LDA start =====')
        computeLDA(
            train_images,
            train_labels,
            test_images,
            test_labels,
            mode,
            num_k,
            kernel_type,
            gamma)
```

3. computePCA()

The main function of compute PCA. First, according to the given method (simple or kernel) to compute the covariance matrix. Then, find the principal eigenvectors corresponding to 25 first largest eigenvalues of the covariance matrix as principal eigenvectors. Next, compute the eigenfaces and reconstruct the eigenfaces and visualize it. Finally, use principal eigenvectors to classify and recognize the test images.

```python
def computePCA(train_images: np.ndarray, train_labels: np.ndarray, test_images: np.ndarray,
               test_labels: np.ndarray, mode: int, num_k: int, kernel_type: int, gamma: float) -> None:
    """
    compute Principal Components Analysis
    """
    # Compute mean of train_images
    mean_images = np.sum(train_images, axis=0) / len(train_images)

    if mode == 0:
        # Simple PCA
        cov_matrix = computeSimplePCACov(train_images, mean_images)

        # Recorded the input config
        print(f'Simple PCA with {num_k}-NN')
    else:
        # Kernel PCA
        cov_matrix = computeKernelPCACov(train_images, kernel_type, gamma)

        # Recorded the input config
        print(
            f'{"RBF" if kernel_type else "Linear"} Kernel PCA with {num_k}-NN',
            end='')
        if kernel_type:
            print(f' gamma {gamma}')
        else:
            print('')

    # Find 25 first largest principal eigenvectors
    principal_eigenvectors = findPrincipalEigenvectors(cov_matrix)
    # Get eigenfaces and show it
    eigenfaces = principal_eigenvectors.T
    drawEigenfaces(
        eigenfaces.reshape(
            25,
            config.HEIGHT,
            config.WIDTH),
        'Original PCA Eigenfaces')
    saveFigure(0, mode, kernel_type, 'PCA_Original_Eigenfaces')

    # Randomly choose 10 train eigenfaces to reconstruction and show it
    reconstruction_eigenfaces = reconstructEigenfaces(
        train_images, principal_eigenvectors)
    drawEigenfaces(
        reconstruction_eigenfaces.reshape(
            10,
            config.HEIGHT,
            config.WIDTH),
        'Reconstruct PCA Eigenfaces')
    saveFigure(0, mode, kernel_type, 'PCA_Reconstruct_Eigenfaces')

    # Classify test_images for face recognize.
    classifyAndRecognize(
        train_images,
        train_labels,
        test_images,
        test_labels,
        principal_eigenvectors,
        num_k)
```

computeSimplePCACov() in computePCA

According to the formula taught from the course, using a simple method to compute the covariance matrix of images.

$$S = \left[\frac{1}{N} \sum_{x} (x - \bar{x})(x - \bar{x})^{\top} \right]$$

```
def computeSimplePCACov(train_images: np.ndarray,
                        mean_images: np.ndarray) -> np.ndarray:
    """
    Compute the covariance matrix of Simple PCA
    """
    difference_images = train_images - mean_images
    cov_matrix = difference_images.T.dot(difference_images)

    return cov_matrix
```

computeKernelPCACov() in computePCA

According to the kernel type(Linear or RBF) to compute the kernel function of training images. Then use formula taught from the course to compute the covariance matrix of images.

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

```
def computeKernelPCACov(train_images: np.ndarray,
                        kernel_type: int, gamma: float) -> np.ndarray:
    """
    Compute the covariance matrix of kernel PCA
    """
    if kernel_type == 0:
        # Linear
        kernel = train_images.T.dot(train_images)
    else:
        # RBF
        kernel = np.exp(-gamma * cdist(train_images.T,
                        train_images.T, 'sqeuclidean'))

    matrix_n = np.ones((config.WIDTH * config.HEIGHT,
                        config.WIDTH * config.HEIGHT),
                       dtype=float) / (config.WIDTH * config.HEIGHT)
    cov_matrix = kernel - \
        matrix_n.dot(kernel) - kernel.dot(matrix_n) + \
        matrix_n.dot(kernel).dot(matrix_n)

    return cov_matrix
```

4. **computeLDA()**

Basically, it does the same things as PCA. Only one difference between LDA and PCA is the method(Simple and kernel) of computing covariance matrix.

```python
def computeLDA(train_images: np.ndarray, train_labels: np.ndarray, test_images: np.ndarray,
               test_labels: np.ndarray, mode: int, num_k: int, kernel_type: int, gamma: float) -> None:
    """
    compute Linear Discriminative Analysis
    """
    _, count_num_of_class = np.unique(train_labels, return_counts=True)

    if mode == 0:
        # Simple LDA
        cov_matrix = computeSimpleLDACov(
            count_num_of_class, train_images, train_labels)

        # Recorded the input config
        print(f'Simple LDA with {num_k}-NN')
    else:
        # Kernel LDA
        cov_matrix = computeKernelLDACov(
            count_num_of_class,
            train_images,
            train_labels,
            kernel_type,
            gamma)

        # Recorded the input config
        print(
            f'{"RBF" if kernel_type else "Linear"} Kernel LDA with {num_k}-NN',
            end='')
        if kernel_type:
            print(f' gamma {gamma}')
        else:
            print('')

    # Find 25 first largest principal eigenvectors
    principal_eigenvectors = findPrincipalEigenvectors(cov_matrix)
    # Get eigenfaces and show it
    eigenfaces = principal_eigenvectors.T
    drawEigenfaces(
        eigenfaces.reshape(
            25,
            config.HEIGHT,
            config.WIDTH),
        'Original LDA Eigenfaces')
    saveFigure(1, mode, kernel_type, 'LDA_Original_Eigenfaces')

    # Randomly choose 10 train eigenfaces to reconstruction and show it
    reconstruction_eigenfaces = reconstructEigenfaces(
        train_images, principal_eigenvectors)
    drawEigenfaces(
        reconstruction_eigenfaces.reshape(
            10,
            config.HEIGHT,
            config.WIDTH),
        'Reconstruct LDA Eigenfaces')
    saveFigure(1, mode, kernel_type, 'LDA_Reconstruct_Eigenfaces')

    # Classify test_images for face recognize.
    classifyAndRecognize(
        train_images,
        train_labels,
        test_images,
        test_labels,
        principal_eigenvectors,
        num_k)
```

computeSimpleLDACov() in computeLDA

According to the formula taught from the course, first compute the between-class scatter B. Then compute within-class scatter W.

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

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

$$\textit{between-class scatter:}$$

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

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

Finally get the covariance matrix by compute Sw^-1 * Sb.

$$S_B \mathrm{w}_l = \lambda_l S_W \mathrm{w}_l$$

```python
# Get eigenfaces and show it
eigenfaces = principal_eigenvectors.T
drawEigenfaces(
    eigenfaces.reshape(
        25,
        config.HEIGHT,
        config.WIDTH),
    'Original LDA Eigenfaces')
saveFigure(1, mode, kernel_type, 'LDA_Original_Eigenfaces')

# Randomly choose 10 train eigenfaces to reconstruction and show it
reconstruction_eigenfaces = reconstructEigenfaces(
    train_images, principal_eigenvectors)
drawEigenfaces(
    reconstruction_eigenfaces.reshape(
        10,
        config.HEIGHT,
        config.WIDTH),
    'Reconstruct LDA Eigenfaces')
saveFigure(1, mode, kernel_type, 'LDA_Reconstruct_Eigenfaces')

# Classify test_images for face recognize.
classifyAndRecognize(
    train_images,
    train_labels,
    test_images,
    test_labels,
    principal_eigenvectors,
    num_k)
```

```python
def computeSimpleLDACov(count_num_of_class: int, train_images: np.ndarray,
                        train_labels: np.ndarray) -> np.ndarray:
    # Compute global mean
    global_mean = np.mean(train_images, axis=0)
    image_size = config.HEIGHT * config.WIDTH

    # Compute mean of each class
    class_num = len(count_num_of_class)
    class_mean = np.zeros((class_num, image_size))
    for label_idx in range(class_num):
        class_mean[label_idx, :] = np.mean(
            train_images[train_labels == label_idx + 1], axis=0)

    # Compute between-class scatter B
    scatter_b = np.zeros((image_size, image_size), dtype=float)
    for idx in range(len(count_num_of_class)):
        # difference = mj - m
        difference = (class_mean[idx] - global_mean).reshape((image_size, 1))
        scatter_b += count_num_of_class[idx] * difference.dot(difference.T)

    # Compute within-class scatter
    scatter_w = np.zeros((image_size, image_size), dtype=float)
    for idx in range(len(class_mean)):
        # difference = xi - mj, where i is belongs to Cj
        difference = train_images[train_labels == idx + 1] - class_mean[idx]
        scatter_w += difference.T.dot(difference)

    # Compute lambda = Sw^-1 * Sb
    cov_matrix = np.linalg.pinv(scatter_w).dot(scatter_b)

    return cov_matrix
```

computeKernelLDACov() in computeLDA

When compute the covariance matrix of kernel LDA, first I use the given kernel type(Linear or RBF) to compute the kernel function of training images. Then use the formula searched from wikipedia to compute matrix M and N. Finally get the covariance matrix by compute N^-1 * M.

$$(\mathbf{M}_i)_j = \frac{1}{l_i}\sum_{k=1}^{l_i} k(\mathbf{x}_j, \mathbf{x}_k^i). \quad \mathbf{N} = \sum_{j=1,2} \mathbf{K}_j(\mathbf{I} - \mathbf{1}_{l_j})\mathbf{K}_j^{\mathrm{T}}$$

$$\alpha = \mathbf{N}^{-1}(\mathbf{M}_2 - \mathbf{M}_1).$$

```python
def computeKernelLDACov(count_num_of_class: int, train_images: np.ndarray,
                        train_labels: np.ndarray, kernel_type: int, gamma: float):
    """
    reference: https://en.wikipedia.org/wiki/Kernel_Fisher_discriminant_analysis
    matrix N: sigma Kk(I - 1/num_k * I)Kk^T, Kk means k-th kernel
    """
    class_num = len(count_num_of_class)
    image_num = len(train_images)
    image_size = config.HEIGHT * config.WIDTH

    # Compute kernel
    if kernel_type == 0:
        # Linear
        kernel_per_class = np.zeros((class_num, image_size, image_size))
        for idx in range(class_num):
            tmp_image = train_images[train_labels == idx + 1]
            kernel_per_class[idx] = tmp_image.T.dot(tmp_image)
        kernel_all = train_images.T.dot(train_images)
    else:
        # RBF
        kernel_per_class = np.zeros((class_num, image_size, image_size))
        for idx in range(class_num):
            tmp_image = train_images[train_labels == idx + 1]
            kernel_per_class[idx] = np.exp(-gamma *
                                           cdist(tmp_image.T, tmp_image.T, 'sqeuclidean'))
        kernel_all = np.exp(-gamma * cdist(train_images.T,
                                           train_images.T, 'sqeuclidean'))

    # Compute matrix N
    matrix_n = np.zeros((image_size, image_size))
    identity_matrix = np.identity(image_size)
    for idx in range(class_num):
        matrix_n += kernel_per_class[idx].dot(identity_matrix - (
            1 / count_num_of_class[idx]) * identity_matrix).dot(kernel_per_class[idx].T)

    # Compute matrix M
    M_i = np.zeros((class_num, image_size))
    for idx, kernel in enumerate(kernel_per_class):
        for row_idx, row in enumerate(kernel):
            M_i[idx, row_idx] = np.sum(row) / count_num_of_class[idx]
    M_mean = np.zeros(image_size)
    for idx, row in enumerate(kernel_all):
        M_mean[idx] = np.sum(row) / image_num
    matrix_m = np.zeros((image_size, image_size))
    for idx, num in enumerate(count_num_of_class):
        difference = (M_i[idx] - M_mean).reshape((image_size, 1))
        matrix_m += num * difference.dot(difference.T)

    # Compute alpha = N^-1 * M
    cov_matrix = np.linalg.pinv(matrix_n).dot(matrix_m)

    return cov_matrix
```

The shared function of LDA and PCA.

5. **findPrincipalEigenvectors()**

After we get the covariance matrix, we compute the eigenvalues and eigenvectors of the covariance matrix and find the principal eigenvectors which correspond to 25 first largest eigenvalues.

```python
def findPrincipalEigenvectors(cov_matrix: np.ndarray) -> np.ndarray:
    """
    Find 25(in this work) first largest eigenvectors as principal components.
    """
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

    # Find 25 first largest eigenvectors
    sort_index = np.argsort(-eigenvalues)
    principal_eigenvectors = eigenvectors[:, sort_index[0:25]].real

    return principal_eigenvectors
```

6. drawEigenfaces()

We can simply get eigenfaces by computing the transpose of principal eigenvectors. Given eigenfaces, then use this function to draw the eigenfaces.

```python
def drawEigenfaces(eigenfaces: np.ndarray, title: str) -> None:
    """
    Show the results of eigenfaces.
    """
    plt.figure(title)
    # Five eigenfaces per row
    num_row = int(eigenfaces.shape[0] / 5)
    for idx in range(len(eigenfaces)):
        plt.subplot(num_row, int((len(eigenfaces) + 1) / num_row), idx + 1)
        plt.axis('off')
        plt.imshow(eigenfaces[idx], cmap='gray')
```

7. reconstructEigenfaces()

I randomly choose ten training images and use the formula below to reconstruct eigenfaces.

$$\underbrace{xW}_{z}W^{\top}$$

```python
def reconstructEigenfaces(train_images: np.ndarray,
                          principal_eigenvectors: np.ndarray) -> np.ndarray:
    """
    Randomly choose 10 train_images to reconstruct eigenfaces.
    """
    # Randomly choose 10 train_images
    chosen_index = random.sample(range(len(train_images)), 10)

    reconstruction_eigenfaces = train_images[chosen_index].dot(
        principal_eigenvectors).dot(principal_eigenvectors.T)

    return reconstruction_eigenfaces
```

8. classifyAndRecognize()

First, I decorrelated the training images and testing images according to the principal eigenvectors. Then, the function uses k-NN to classify the testing images it should belong to.

```python
def classifyAndRecognize(train_images: np.ndarray, train_labels: np.ndarray, test_images: np.ndarray,
                         test_labels: np.ndarray, principal_eigenvectors: np.ndarray, num_k: int) -> None:
    # Get the number of train_images and test_images
    num_of_train = len(train_images)
    num_of_test = len(test_images)

    decorrelated_train = decorrelateImages(
        num_of_train, train_images, principal_eigenvectors)
    decorrelated_test = decorrelateImages(
        num_of_test, test_images, principal_eigenvectors)
    error = 0
    distance = np.zeros(num_of_train)
    for test_idx in range(num_of_test):
        for train_idx in range(num_of_train):
            distance[train_idx] = np.linalg.norm(
                decorrelated_test[test_idx] - decorrelated_train[train_idx])
        min_distance = np.argsort(distance)[:num_k]
        predict = np.argmax(np.bincount(train_labels[min_distance]))
        if predict != test_labels[test_idx]:
            error += 1
    print(
        f'Error count: {error}, Accuracy: {(len(test_labels) - error) / num_of_test}')
```

decorrelatedImages()

This function project gives images to principal eigenspace.

```python
def decorrelateImages(num_of_images: int, images: np.ndarray,
                      principal_eigenvectors: np.ndarray) -> np.ndarray:
    """
    Decorrelate unnecessary components.
    """
    decorrelate_images = np.zeros((num_of_images, 25))
    for idx, image in enumerate(images):
        decorrelate_images[idx, :] = image.dot(principal_eigenvectors)

    return decorrelate_images
```

b. t-SNE

1. Modify

In the original t-SNE, the compute formula of gradient and Q(the similarity of lower dimension) is below.

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

gradient:

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

In the symmetric SNE, the compute formula of gradient and Q(the similarity of lower dimension) is below.

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

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

Based on the formula above, the difference between symmetric SNE and t-SNE is q and gradient. Thus, we only need to modify q and gradient.

In original tsne()

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

my ssne()

```
num = np.exp(-1. * 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. **drawCurrentImage()**

To produce the gif results, this function will be called every 10 iterations, draw the current results and appends to an image array.

```
# 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))
    image.append(drawCurrentImage(Y, labels, 't-SNE', perplexity))
```

```
def drawCurrentImage(Y: np.ndarray, labels: np.ndarray, title: str, perplexity: float) -> Image:
    plt.clf()
    plt.title(title)
    plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
    plt.tight_layout()
    canvas = plt.get_current_fig_manager().canvas
    canvas.draw()

    return Image.frombytes('RGB', canvas.get_width_height(), canvas.tostring_rgb())
```

3. **visualize()**

After t-SNE or symmetric SNE is finished, this function will draw the high-dimensional similarities and low-dimensional similarities of results.

```
def visualize(Y: np.ndarray, P: np.ndarray, Q: np.ndarray,
              labels: np.ndarray, title: str):
    plt.clf()
    plt.figure(title)

    dirname = './output_images/SNE'
    os.makedirs(dirname, exist_ok=True)
    plt.title(title)
    plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
    plt.savefig(f'{dirname}/{title}.png')

    sorted_idx = np.argsort(labels)

    plt.figure('similarity')
    # Plot P
    log_P = np.log(P)
    sorted_P = log_P[sorted_idx][:, sorted_idx]
    plt.subplot(121)
    plt.title('The similarity of High dimensions')
    plt.imshow(P, cmap='hot', interpolation='nearest')

    # Plot Q
    log_Q = np.log(Q)
    sorted_Q = log_Q[sorted_idx][:, sorted_idx]
    plt.subplot(122)
    plt.title('The similarity of Low dimensions')
    plt.imshow(Q, cmap='hot', interpolation='nearest')
    plt.savefig(f'{dirname}/{title}_similarity.png')
```

B. experiments settings and results (35%) & discussion (15%)

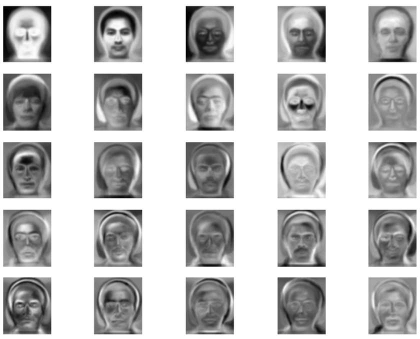
a. Kernel Eigenfaces

settings:

compress size: 98 x 116

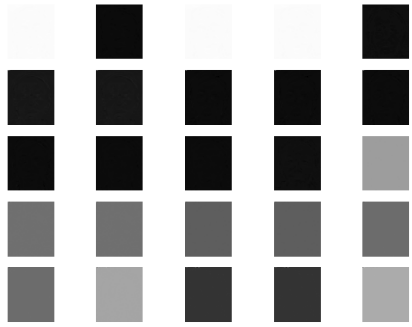
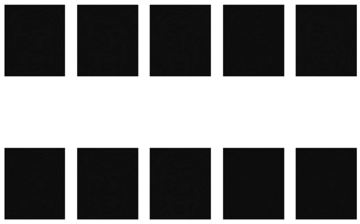
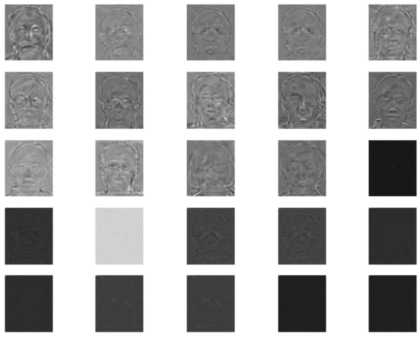
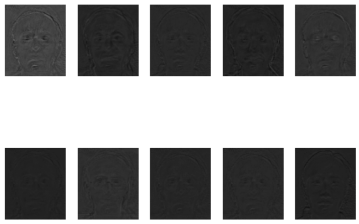
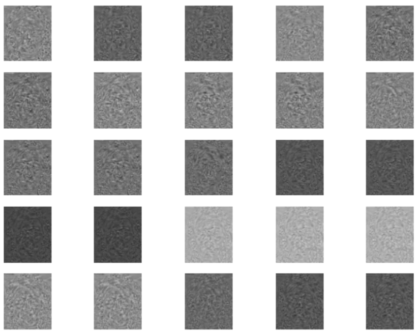
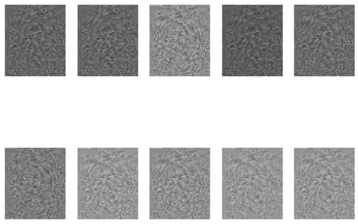
k_NN: 5

PCA

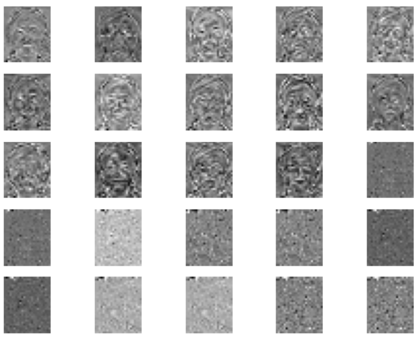
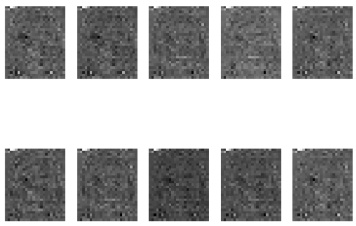
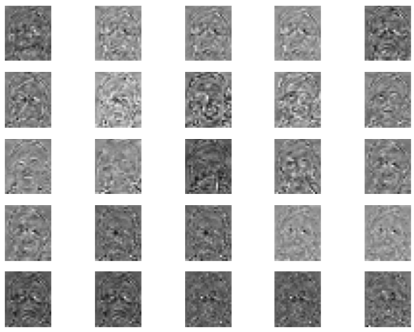
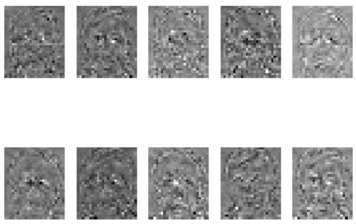
	Eigenfaces	Reconstruct faces	Accuracy
Simple			Error count: 3 Accuracy: 0.9
Linear kernel			Error count: 4 Accuracy: 0.866
RBF kernel			Error count: 4 Accuracy: 0.866

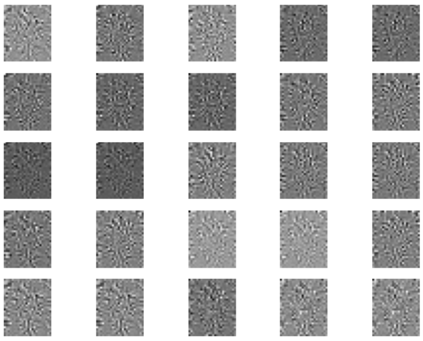
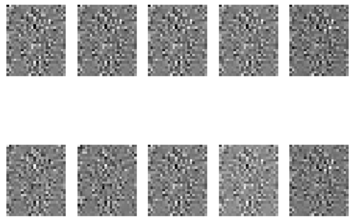
LDA

	Eigenfaces	Reconstruct faces	Accuracy

Simple			Error count: 5 Accuracy: 0.833
Linear kernel			Error count: 16 Accuracy: 0.4666
RBF kernel			Error count: 25 Accuracy: 0.1666

LDA (compress size: 24 x 29)

	Eigenfaces	**Reconstruct faces**	**Accuracy**
Simple			Error count: 1 Accuracy: 0.9666
Linear kernel			Error count: 22 Accuracy: 0.2666

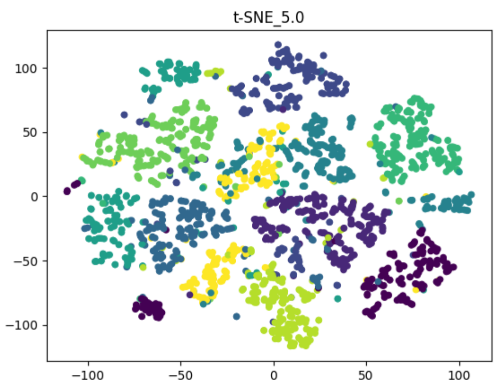
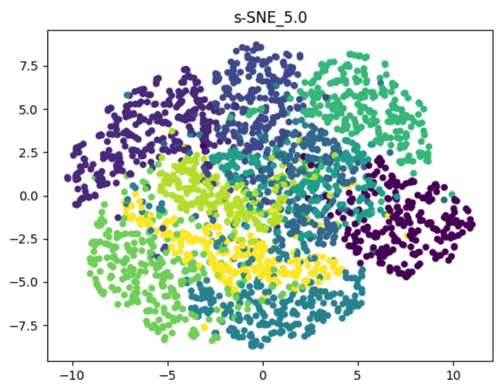
RBF kernel			Error count: 16 Accuracy: 0.4666(

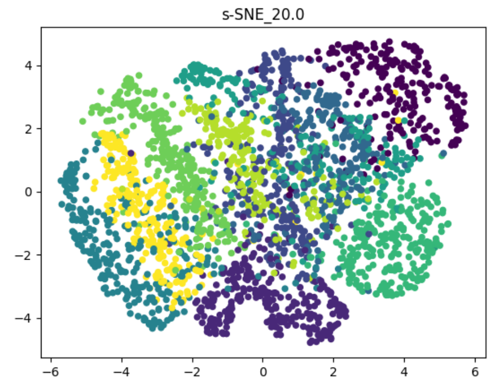
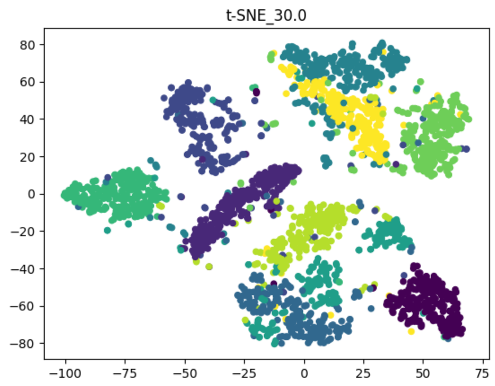
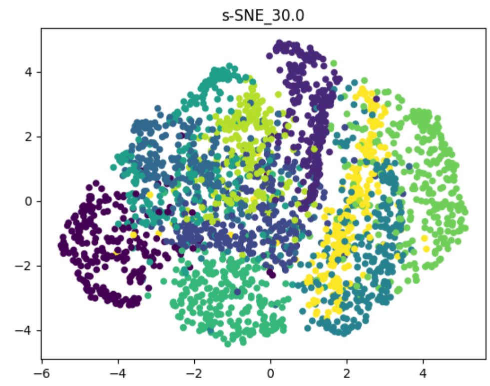
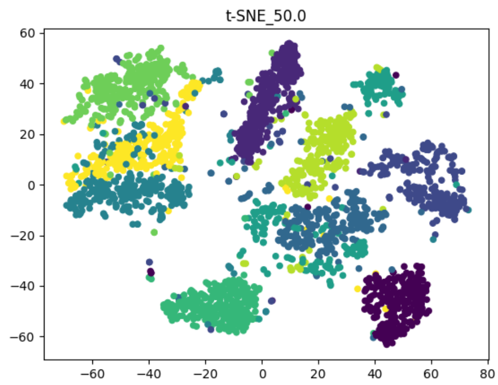
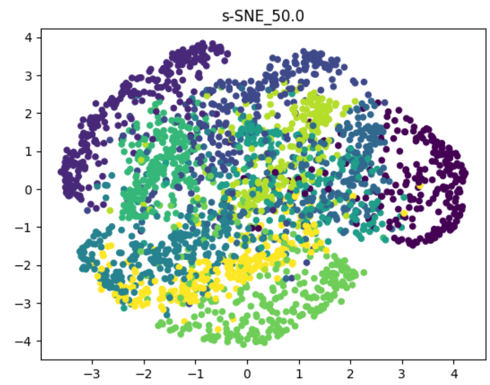
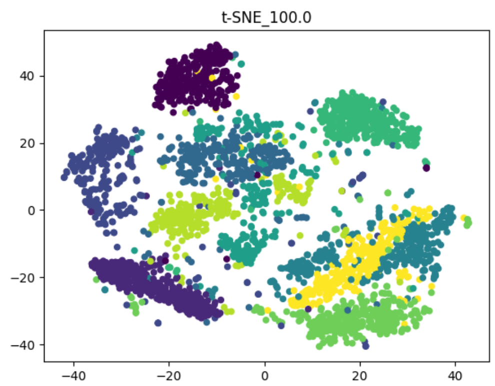
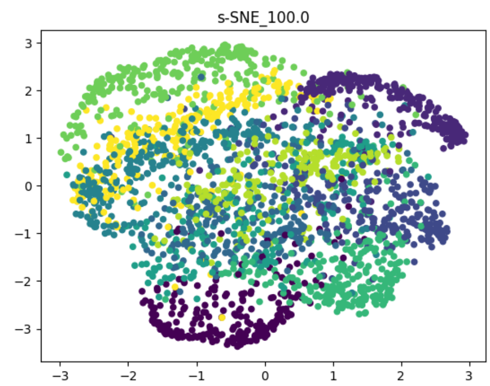
discussion:

In my experiments, you can see that the original eigenfaces and reconstructed eigenfaces of PCA are more clearly than LDA. And the overall accuracy of PCA is better than LDA, too. The second observation of my experiments is that if the compress size is too large(i.e. 98x116), the eigenfaces of LDA are pure black and cannot see any features. I modify the compress size by setting config.py and the eigenfaces of LDA are more clear than before, but still cannot seem clear like PCA. Overall, the results of simple methods are better than the kernel methods.

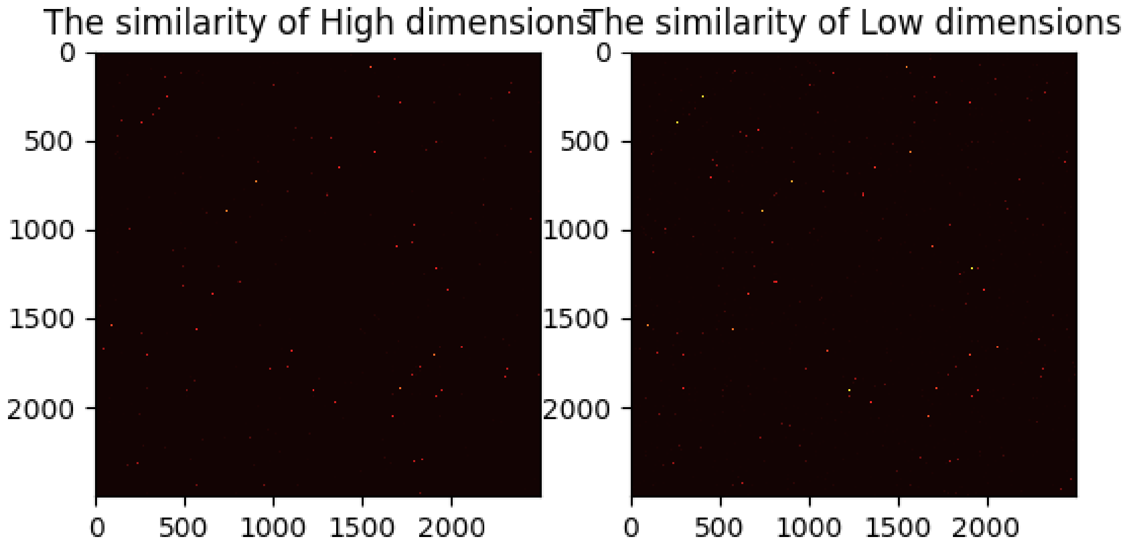
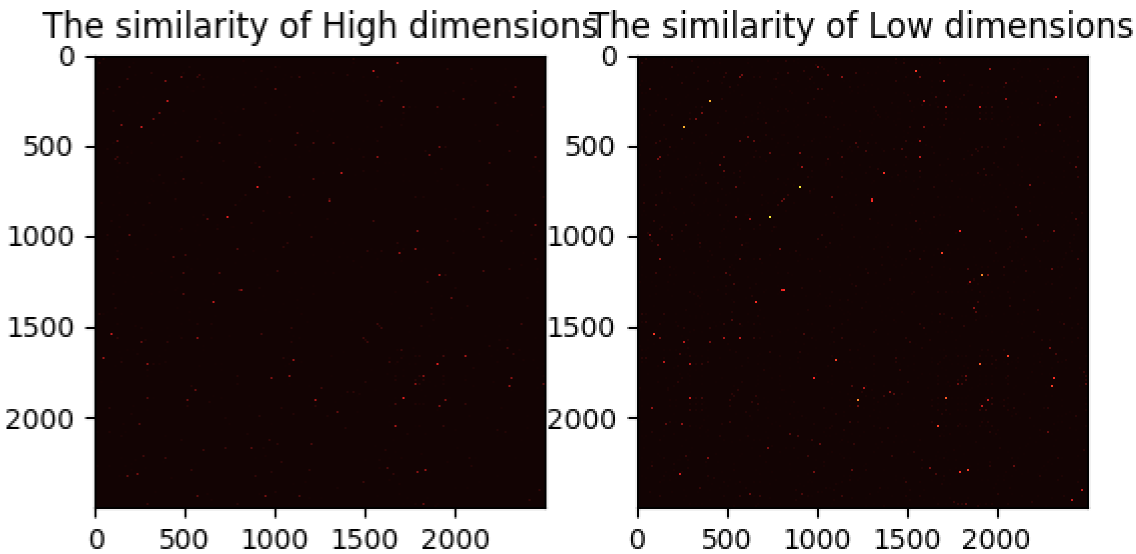
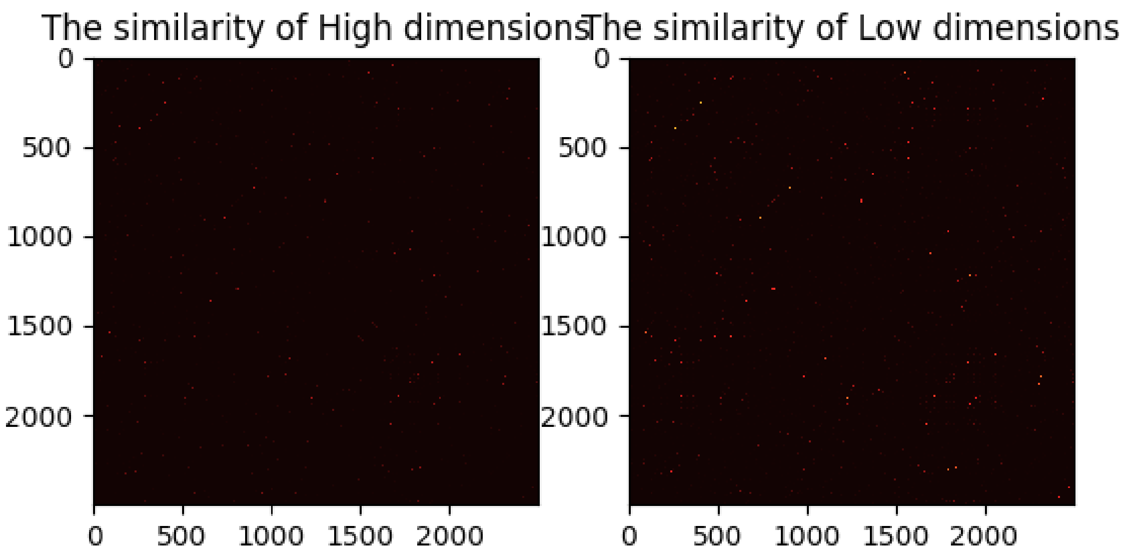
b. t-SNE

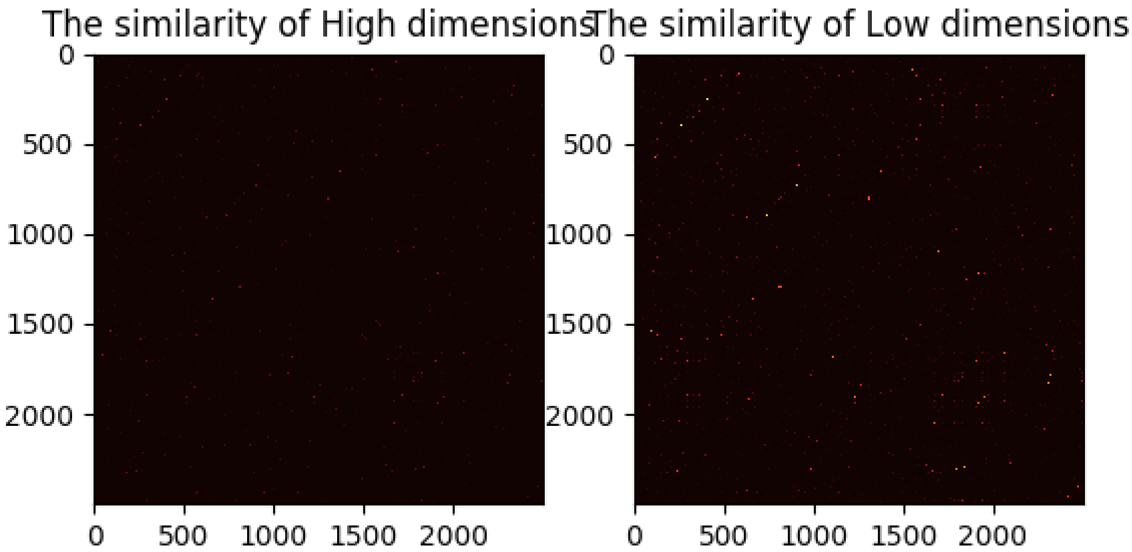
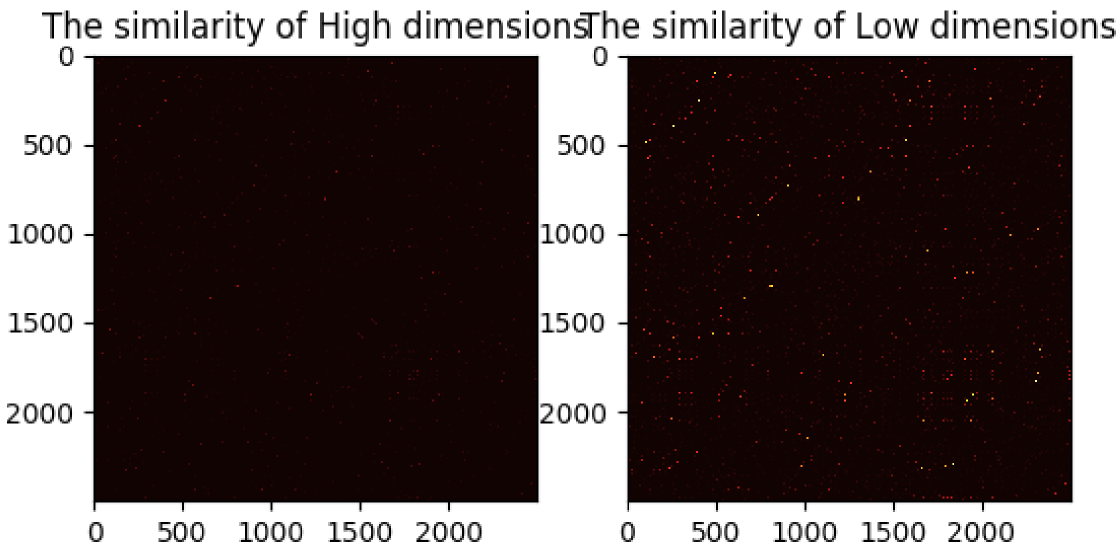
t-SNE vs symmetric SNE

perplexity	**t-SNE**	**symmetric SNE**
5		

20		
30		
50		
100		

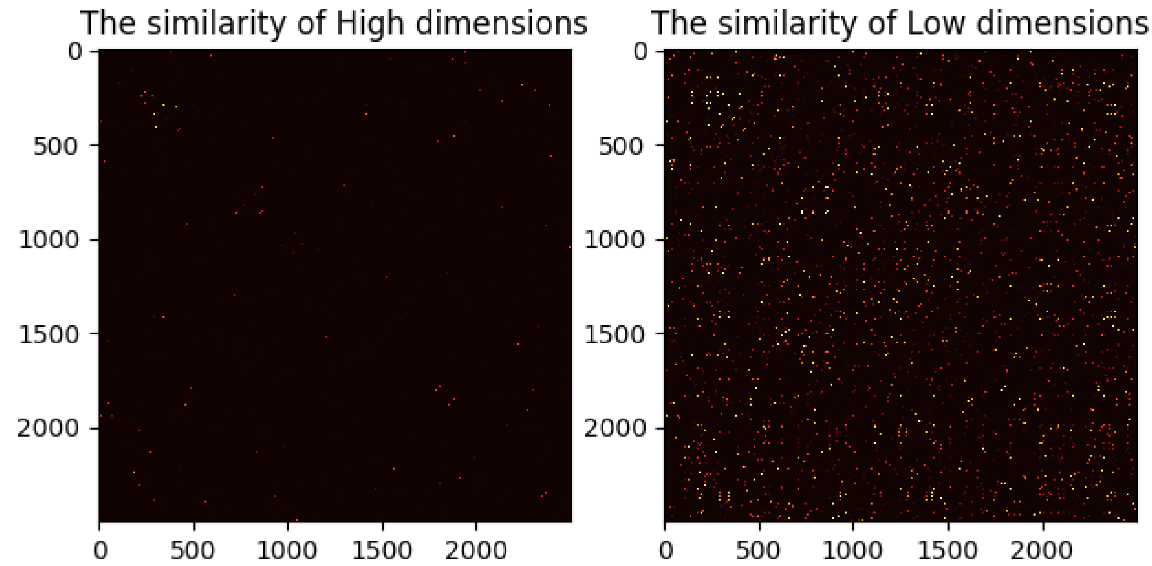
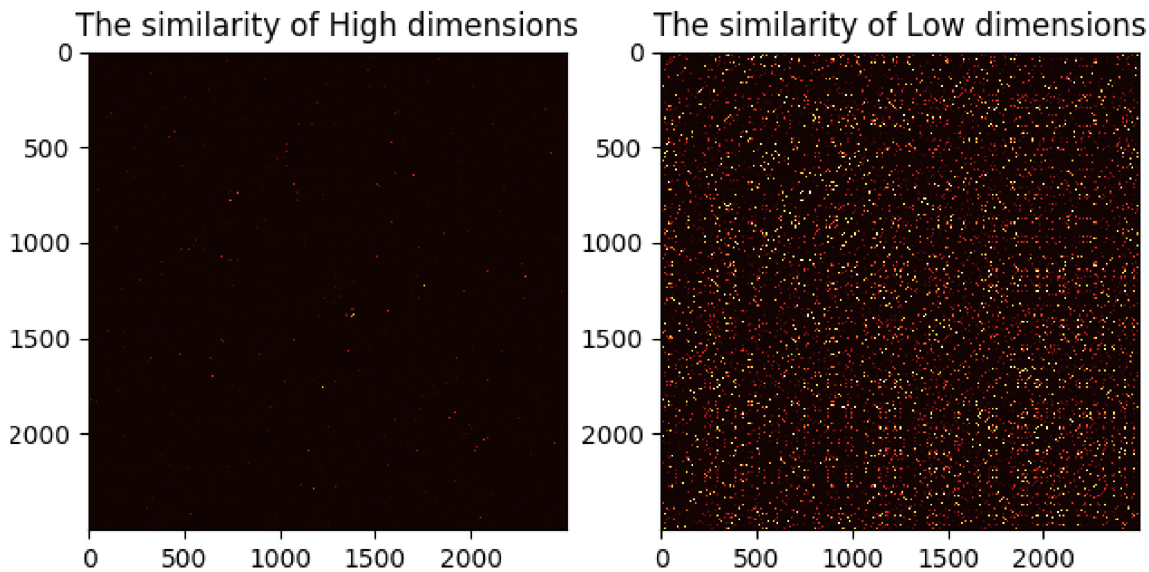
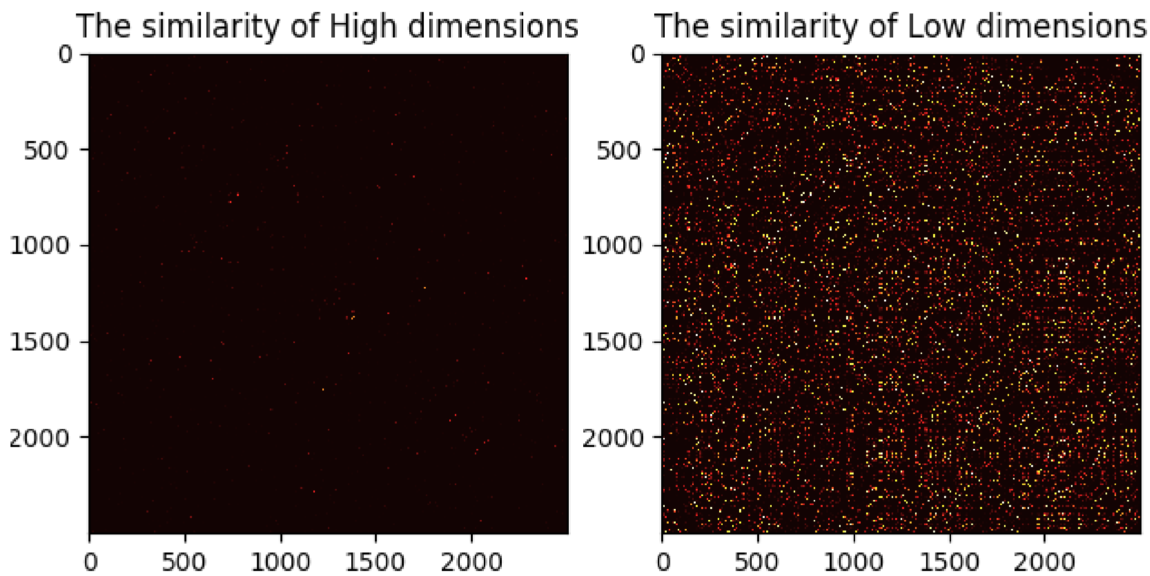
Similarity - t-SNE

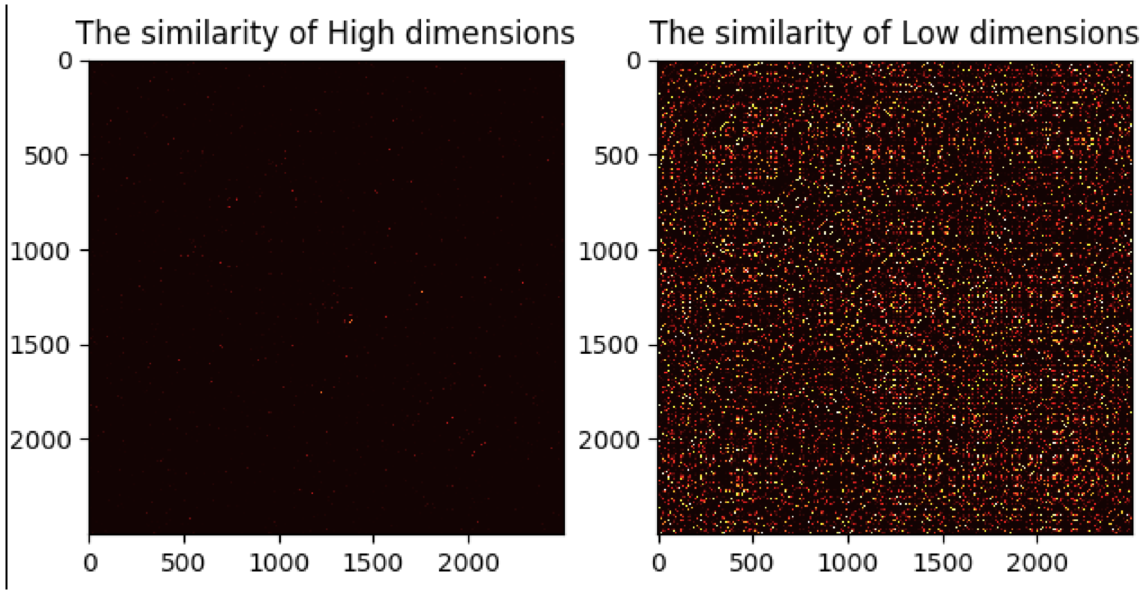
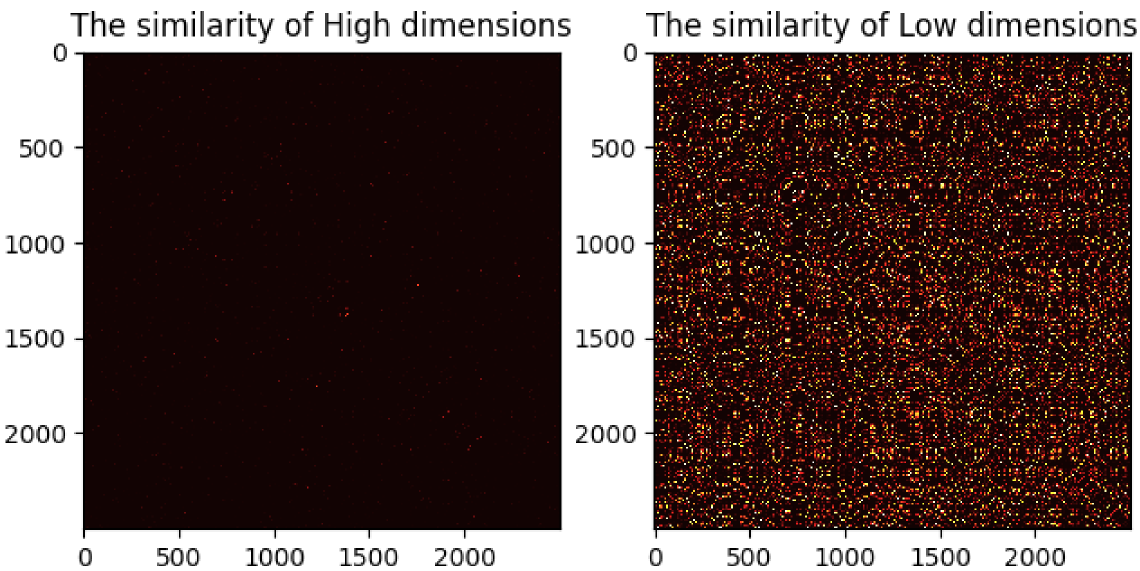
<table>
<tr><th>perplexity</th><th>similarity</th></tr>
<tr><td>5</td><td>
</td></tr>
<tr><td>20</td><td>
</td></tr>
<tr><td>30</td><td>
</td></tr>
</table>

50	
100	

Similarity - symmetric SNE

perplexity	**similarity**

5	The similarity of High dimensions The similarity of Low dimensions 0 500 1000 1500 2000
20	The similarity of High dimensions The similarity of Low dimensions 0 500 1000 1500 2000
30	The similarity of High dimensions The similarity of Low dimensions 0 500 1000 1500 2000

<table>
<tr><td>50</td><td>
</td></tr>
<tr><td>100</td><td>
</td></tr>
</table>

discussion

Part1. Compare the results of the t-SNE and symmetric-SNE shown above, we can easily observe that symmetric-SNE is more crowded than t-SNE.

Part4. No matter the perplexity is how much, the crowded problem in symmetric-SNE is not changed. In the t-SNE, change perplexity will influence the results graph. In my experience, setting perplexity to 30 will get best results.

C. Observation

   a. **Although the eigenfaces of LDA are not clear, the accuracy of LDA is better than the accuracy of PCA(only in simple cases.)**
   b. **In theory, the classification of supervised methods should be better than unsupervised methods. But in my**

experience, the results of kernel LDA are worse than kernel PCA.

c. Eigenfaces preserve the principal feature of training images. Thus, we can use eigenfaces to classify testing images.
d. The reconstructed faces of PCA look like the original faces.
e. The performance between simple method and kernel method are not big differences. And the kernel function between linear and RBF are not big differences, too.
f. The larger compress size we set, the more compute time we need. (e.g. 98x116 >>> 24x29)
g. In PCA, the larger compress size we set, the more clearly eigenfaces we get.
h. In LDA,if the compress size is too large(98x116), the eigenfaces are all black and cannot see any outline. Conversely, with the compress size set to 24x29, we can see a little bit of the outline of the eigenfaces.
i. According to my experience, the symmetric SNE suffers from the crowded problem. It almost cannot distinguish the different classes without color. Compared to symmetric SNE, t-SNE uses t-distribution to reduce the crowded problem.
j. Looking at the similarity graph of symmetric SNE, the high dimension graph is sparse and the low dimension graph is crowded. As the perplexity increases, the crowded situation of low dimension similarity graphs is intensified.
k. Looking at the similarity graph of t-SNE, the low dimension graph is more sparse than symmetric SNE.
l. The convergence speed of symmetric SNE is faster than t-SNE in the gif results.
m. The perplexity will influence the number of neighbors to be considered. The larger the perplexity we set, the less sensitive it is to classify the small group. i.e Small groups are easy to ignore. This situation can be observed clearly in t-SNE. Since the symmetric SNE suffers from a crowded problem, the effect of perplexity is not clear.