Report

1-1. Code Explanation: Kernel Eigenfaces

- Part 1: PCA (Eigenfaces) and LDA (Fisherfaces)

    - Read Dataset

pca_lda.py

```python
def read_data(root_path, is_train):

    if is_train:
        root_path = os.path.join(root_path, "Training")
    else:
        root_path = os.path.join(root_path, "Testing")

    name = []
    data = []
    label = []

    for pgm_file in os.listdir(root_path):

        # image name
        name.append(pgm_file)

        # image data
        img = Image.open(os.path.join(root_path, pgm_file))
        img = img.resize((50, 50), Image.ANTIALIAS)
        img = np.array(img)
        data.append(img.ravel().astype(np.float64))

        # image label
        label.append(int(pgm_file[7:9]))

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

pca_lda.py

```python
    train_name, train_data, train_label = read_data(root_path='Yale_Face_Database', is_train=True)
    test_name, test_data, test_label = read_data(root_path='Yale_Face_Database', is_train=False)

    # train_data shape = (135, 2500)
    # test_data shape = (30, 2500)
    # all_data shape = (165, 2500)
    all_data = np.vstack((train_data, test_data))

    # train_name shape = (135, )
    # test_name shape = (30, )
    # all_name shape = (165, )
    all_name = np.hstack((train_name, test_name))

    # train_label shape = (135, )
    # test_label shape = (30, )
    # all_label shape = (165, )
    all_label = np.hstack((train_label, test_label))
```

At first, I load training and testing data from Yale Face Database with read_data() function. In this function, each image is resized to 50 by 50, and flattened to one-dimensional array. After reading training and testing data, I merge them with np.vstack() function.

- Part 1 Pipeline

pca_lda.py

```python
        # randomly select 10 images
        target_idx = np.random.choice(all_data.shape[0], 10)
        target_data = all_data[target_idx]
        target_name = all_name[target_idx]

        # PCA eigenfaces: https://laid.delanover.com/explanation-face-recognition-using-eigenfaces/
        print("PCA eigenfaces")
        # eigenfaces shape = (2500, 25)
        # mean shape = (2500, )
        eigenfaces, mean = PCA(all_data, 25)
        visualize("PCA", target_data, target_name, eigenfaces, mean)

        # LDA fisherfaces
        print("LDA fisherfaces")
        # fisherfaces shape = (2500, 25)
        fisherfaces = LDA(all_data, all_label, 25)
        visualize("LDA", target_data, target_name, fisherfaces)
```

Above image demonstrates the computation flow of task in part 1. Firstly, I randomly choose ten images from all_data including both training and testing images. Secondly, I run two kinds of dimension reduction algorithms, PCA and LDA on all_data, and only keep the first twenty-five Eigenfaces and Fisherfaces with PCA() and LDA() function respectively. Finally, I visualize Eigenfaces and Fisherfaces, and reconstruct original images in visualize() function.

- PCA

pca_lda.py

```python
def PCA(imgs, keep_nums):
    # imgs shape = (165, 2500)

    # mean of all images
    # shape = (2500, )
    mean = np.mean(imgs, axis=0)

    # extract distinguished features
    # shape = (165, 2500)
    imgs_feature = imgs - mean

    # covariance
    # shape = (165, 165)
    covariance = imgs_feature @ imgs_feature.T

    # eigen-decomposition
    # eigenvalue shape = (165, )
    # eigenvector shape = (165, 165)
    eigenvalue, eigenvector = np.linalg.eigh(covariance)

    # calculate eigenfaces
    # shape = (2500, 165)
    eigenfaces = imgs_feature.T @ eigenvector

    # normalize eigenfaces
    for i in range(eigenfaces.shape[1]):
        eigenfaces[:, i] = eigenfaces[:, i] / np.linalg.norm(eigenfaces[:, i])

    # sort eigenfaces based on its corresponding eigenvalues
    idx = np.argsort(eigenvalue)[::-1]
    eigenfaces = eigenfaces[:, idx]

    # only keep first K eigenfaces
    eigenfaces = eigenfaces[:, :keep_nums].real
    return eigenfaces, mean
```

I use PCA() to compute Eigenfaces. The goal of PCA() is to orthogonally project original data x to z with maximum variance. The first step of PCA is to compute distinguished feature of each image. Therefore, each image is subtracted by the mean of all images. After that, the covariance matrix is built. I compute eigenvectors of covariance matrix. The projection matrix is composed of the first twenty-five eigenvectors. After finding projection matrix, I multiply it with imgs_feature which is the original image data subtracted by mean, and get Eigenfaces.

- LDA

pca_lda.py

```python
def LDA(imgs, labels, keep_nums):
    # imgs shape = (165, 2500)
    # labels shape = (165, )

    all_class = np.unique(labels)
    mean = np.mean(imgs, axis=0)

    # S_w: variance in class, shape = (2500, 2500)
    # S_b variance among classes, shape = (2500, 2500)
    S_w = np.zeros((imgs.shape[1], imgs.shape[1]), dtype=np.float64)
    S_b = np.zeros((imgs.shape[1], imgs.shape[1]), dtype=np.float64)

    # calculate S_w and S_b
    for c in all_class:
        imgs_subset = imgs[np.where(labels == c)[0], :]
        mean_subset = np.mean(imgs_subset, axis=0)
        S_w += (imgs_subset - mean_subset).T @ (imgs_subset - mean_subset)
        S_b += imgs_subset.shape[0] * ((mean_subset - mean).T @ (mean_subset - mean))

    # eigen-decomposition
    # eigenvalue shape = (2500, )
    # eigenvector shape = (2500, 2500)
    eigenvalue, eigenvector = np.linalg.eig(np.linalg.pinv(S_w) @ S_b)

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

    # sort eigenvectors based on its corresponding eigenvalues
    idx = np.argsort(eigenvalue)[::-1]
    fisherfaces = eigenvector[:, idx]

    # only keep first K eigenfaces
    fisherfaces = fisherfaces[:, :keep_nums].real
    return fisherfaces
```

I use LDA() to get Fisherfaces. The goal of LDA() is to maximize between-class scatter (S_b) and minimize within-class variance (S_w). S_b and S_w are calculated based on these equations:

$$Scatter_{between-classes} = S_b = \sum_{i=1}^{c} N_i(\mu_i - \mu)(\mu_i - \mu)^T$$

$$Scatter_{within-classes} = S_w = \sum_{i=1}^{c} \sum_{x_j \in X_c} (x_j - \mu_i)(x_j - \mu_i)^T$$

After that, we find the eigenvectors of multiplication of “inverse of S_w” and “S_B”. The projection matrix is composed of first twenty-five eigenvectors.

- Visualize & Reconstruction

```python
# pca_lda.py
    # save 25 eigenfaces in one image
    fig, axs = plt.subplots(5, 5)
    for i in range(5):
        for j in range(5):
            axs[i, j].imshow(eigenfaces[:, i*5 + j].reshape((50, 50)), cmap='gray')
            axs[i, j].axis('off')

    if title == "PCA":
        plt.savefig('PCA eigenfaces/eigenfaces/all.png')
    else:
        plt.savefig('LDA fisherfaces/fisherfaces/all.png')
```

In visualize() function, I save all twenty-five Eigenfaces/Fisherfaces in a single image.

```python
# pca_lda.py
    if mean is None:
        mean = np.zeros(target_data.shape[1])
    projection = (target_data - mean) @ eigenfaces
    reconstruction = projection @ eigenfaces.T + mean
```

Additionally, I project ten testing images from original space to low-dimensional space, and project them back to high-dimensional space for reconstruction.

- Part 2: Face Recognition

  - Part 2 Pipeline

pca_lda.py

```python
        print("Face Recognition: PCA")
        eigenfaces, mean = PCA(all_data, 25)
        train_projection = (train_data - mean) @ eigenfaces
        test_projetion = (test_data - mean) @ eigenfaces
        face_recognition(train_projection, train_label, test_projetion, test_label)

        print("Face Recognition: LDA")
        fisherfaces = LDA(all_data, all_label, 25)
        train_projection = train_data @ fisherfaces
        test_projetion = test_data @ fisherfaces
        face_recognition(train_projection, train_label, test_projetion, test_label)
```

In part 2, I have to apply face recognition to test data in low-dimensional space. There are two methods to project test data to low-dimensional space. One is PCA and the other is LDA. In PCA, train and test data are both subtracted by mean of all data, and then multiply with projection matrix. In LDA, train and test data are directly multiplied with projection matrix. After projecting data into low-dimensional space, the face_recognition() function will recognizes face in images with KNN.

- Face Recognition

pca_lda.py

```python
    test_distance = []
    for i in range(30):

        # calculate distance of a testing image between all training images
        dist_lst = []
        for j in range(135):
            dist = np.sum((test_projection[i] - train_projection[j]) ** 2)
            dist_lst.append([dist, train_label[j]])

        # sort data_lst based on distance
        dist_lst.sort(key=lambda x: x[0])
        test_distance.append(dist_lst)
```

The first step of KNN is to calculate the distance between each test image and all train images in low-dimensional space.

```python
# k-nearest neighbor
for k in range(1, 11):
    correct = 0
    for i in range(30):
        neighbors, count = np.unique(np.array([x[1] for x in test_distance[i][:k]]), return_counts=True)
        predict = neighbors[np.argmax(count)]
        if predict == test_label[i]:
            correct += 1
    print(f'[k={k}] {correct}/{30} => acc: {round(correct / 30, 2):.2f}')
```

After that, I try different “k” in KNN to predict the label of each test image.

- Part 3: Face Recognition with Kernel PCA and Kernel LDA

  - Part 3 Pipeline

```python
kernel_type = "rbf"

print(f"Face Recognition: Kernel PCA ({kernel_type})")
kernel_coord = kernel_PCA(all_data, 25, kernel_type)
train_coord = kernel_coord[:135, :]
test_coord = kernel_coord[135:, :]
face_recognition(train_coord, train_label, test_coord, test_label)

print(f"Face Recognition: Kernel LDA ({kernel_type})")
kernel_coord = kernel_LDA(all_data, all_label, 25, kernel_type)
train_coord = kernel_coord[:135]
test_coord = kernel_coord[135:]
face_recognition(train_coord, train_label, test_coord, test_label)
```

The pipeline in part 3 is really similar to it in part 2. The difference is that instead of using PCA and LDA, I use kernel PCA and kernel LDA.

  - Kernel PCA

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

In kernel PCA, I calculate eigenvectors of kernel instead of covariance matrix. In this assignment, I implement two kinds of kernels, linear and RBF. Their implementation is shown below.

pca_lda.py

```python
    if kernel_type == "linear":
        kernel = imgs @ imgs.T
    elif kernel_type == "rbf":
        kernel = np.exp(-1e-7 * scipy.spatial.distance.cdist(imgs, imgs, 'sqeuclidean'))
```

In order to make sure the data is centered in feature space, we have to centralize kernel with following formula and its implementation.

$$K^C = K - 1_N K - K 1_N + 1_N K 1_N$$

pca_lda.py

```python
    one_mat = np.ones((kernel.shape[0], kernel.shape[0]), dtype=np.float64) / kernel.shape[0]
    kernel = kernel - one_mat @ kernel - kernel @ one_mat + one_mat @ kernel @ one_mat
```

Because the remaining operations are same as original PCA, their explanation is ignored.

- Kernel LDA

pca_lda.py

```python
    # calculate S_w and S_b
    for c in all_class:
        imgs_subset = kernel[np.where(labels == c)[0], :]
        mean_subset = np.mean(imgs_subset, axis=0)
        S_w += imgs_subset.T @ (np.eye(imgs_subset.shape[0]) - \
            (np.ones((imgs_subset.shape[0], imgs_subset.shape[0]), \
                dtype=np.float64) / imgs_subset.shape[0])) @ imgs_subset
        S_b += imgs_subset.shape[0] * ((mean_subset - mean).T @ (mean_subset - mean))
```

In LDA, between-class scatter and within-class variance are calculated based on image data. However, in kernel LDA, they are calculated based on kernel, and follow below formula in which the symbol M represents between-class scatter (S_b) and the symbol N represents within-class variance (S_w).

$$M = \sum_{m=1}^{c} l_m (M_m - M_*)(M_m - M_*)^T$$

$$N = \sum_{m=1}^{c} K_m (I - 1_{l_m}) K_m^T$$

1-2. Code Explanation: t-SNE

- Part 1: Symmetric SNE and t-SNE

  - Main Difference
    The main difference between symmetric SNE and t-SNE is type of distribution used in low-dimensional space. In symmetric SNE, both high-dimensional and low-dimensional space use Gaussian distribution. However, in t-SNE, Gaussian distribution is used in high-dimensional space, and Student-T distribution is used in low-dimensional space, which makes it successfully alleviates crowded problem in low-dimensional space.

  - Pairwise Affinities in t-SNE and Symmetric SNE

tsne.py

```
        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)
```

In t-SNE, pairwise affinities in low-dimensional space are calculated based on Student-T distribution and the corresponding formula is shown below.

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

However, in symmetric SNE, pairwise affinities in low-dimensional space are calculated based on Gaussian distribution and the corresponding formula is shown below.

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

From the original implementation from https://lvdmaaten.github.io/tsne/, I find that the key is how we calculate "num" variable. Therefore, I revise the code to support

tsne.py

```
        if method == 'T-SNE':
            num = 1 / (1 + scipy.spatial.distance.cdist(Y, Y, 'sqeuclidean'))
        else:
            num = np.exp(-1 * scipy.spatial.distance.cdist(Y, Y, 'sqeuclidean'))
        num[range(num_sample), range(num_sample)] = 0
        Q = num / np.sum(num)
        Q = np.maximum(Q, 1e-12)
```

symmetric SNE.

- Gradient in t-SNE and Symmetric SNE
  Because the type of distribution in low-dimensional space is difference between t-SNE and symmetric SNE, their gradient is undoubtedly different.

  The two equations below demonstrate the calculation of gradient in t-SNE and symmetric SNE.

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

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

I add a line of code to calculate gradient in symmetric SNE.

```python
    for i in range(num_sample):
        if method == 'T-SNE':
            dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (target_dim, 1)).T * (Y[i, :] - Y), axis=0)
        else:
            dY[i, :] = np.sum(np.tile(PQ[:, i], (target_dim, 1)).T * (Y[i, :] - Y), axis=0)
```

- Part 2: Visualize Embedding

  - Scatter Plot

```python
def visualize(Y, labels, itr, method, perplexity):
    plt.clf()
    scatter = plt.scatter(Y[:, 0], Y[:, 1], 10, labels)
    plt.legend(*scatter.legend_elements(), loc="upper left")
    plt.title(f'[{method}] perplexity: {perplexity} iter: {itr}')
    plt.savefig(os.path.join(method, str(perplexity), f"{itr}.png"))
```

I use visualize() function to plot the data point in two-dimensional space, and save the figure as a png file.

```python
imgs = [Image.open(os.path.join(PATH, f"{i}.png")) for i in range(START_IDX, END_IDX, STEP)]

imgs[0].save(
    os.path.join(PATH, "result.gif"),
    save_all=True,
    append_images=imgs[1:],
    duration=150,
    loop=0
)
```

All the png files can be converted into a gif file with PIL package.

- Part 3: Visualize Pairwise Similarities

- Histogram Plot

tsne.py

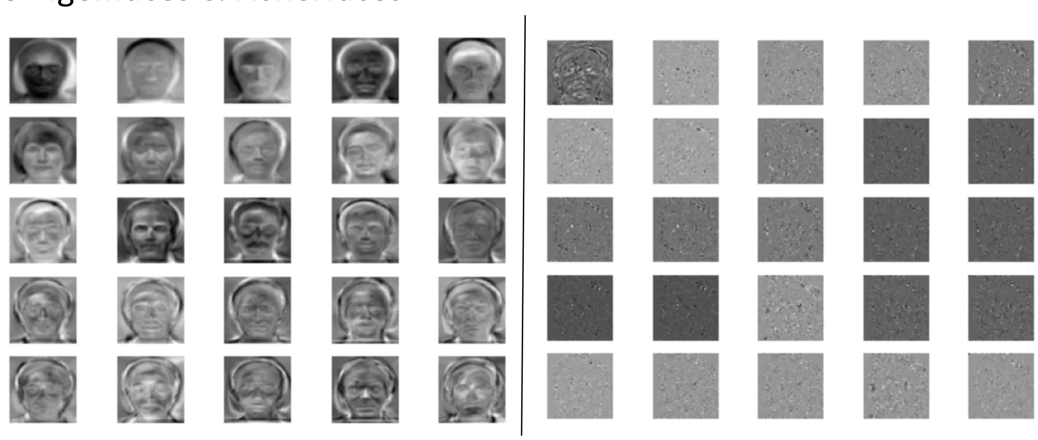
```python
def plot_similarity(P, Q, perplexity, method):
    plt.clf()
    plt.title('[High Dimension] Similarity Distribution')
    plt.hist(x=P.flatten(), bins=100, log=True)
    plt.savefig(os.path.join(method, str(perplexity), "high.png"))

    plt.clf()
    plt.title('[Low Dimension] Similarity Distribution')
    plt.hist(x=Q.flatten(), bins=100, log=True)
    plt.savefig(os.path.join(method, str(perplexity), "low.png"))
```

I visualize the distribution of pairwise similarity in plot_similarity() function. Both high-dimensional and low-dimensional pairwise similarity matrices are flattened into one-dimensional array, and shown as histogram plot.

2-1. Experiment Result & Discussion: Kernel Eigenfaces

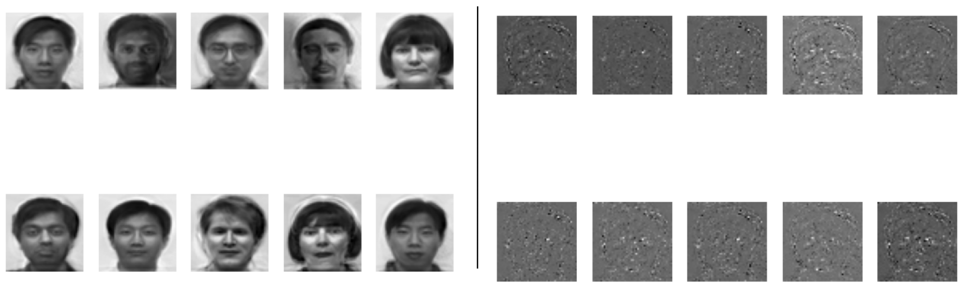
- Part 1: PCA (Eigenfaces) and LDA (Fisherfaces)

  - 25 Eigenfaces & Fisherfaces



We can see clear face image in Eigenfaces, but only see very rough outline of face in Fisherfaces. The reason is that PCA aims to maximize variance, which allow us to capture features among images, and LDA focus on maximizing the ratio of between-class scatter and within-class variance, which makes it capture features to distinguish objects from different classes.

- 10 Test Image Reconstruction



Because Eigenfaces capture the features among all images, we can reconstruct original images from Eigenfaces. However, because the features captured by Fisherfaces are used to differentiate objects from different classes, it is difficult for us to reconstruct original images from Fisherfaces.

- Part 2: Face Recognition

Face Recognition: PCA	Face Recognition: LDA
[k=1] 25/30 => acc: 0.83 [k=2] 25/30 => acc: 0.83 [k=3] 25/30 => acc: 0.83 [k=4] 25/30 => acc: 0.83 [k=5] 27/30 => acc: 0.90 [k=6] 26/30 => acc: 0.87 [k=7] 27/30 => acc: 0.90 [k=8] 26/30 => acc: 0.87 [k=9] 25/30 => acc: 0.83 [k=10] 24/30 => acc: 0.80 Average = 0.849	[k=1] 26/30 => acc: 0.87 [k=2] 27/30 => acc: 0.90 [k=3] 28/30 => acc: 0.93 [k=4] 26/30 => acc: 0.87 [k=5] 25/30 => acc: 0.83 [k=6] 24/30 => acc: 0.80 [k=7] 25/30 => acc: 0.83 [k=8] 25/30 => acc: 0.83 [k=9] 24/30 => acc: 0.80 [k=10] 24/30 => acc: 0.80 Average = 0.846

I use different number of neighbors to test the accuracy of PCA and LDA on face recognition task. The average accuracy of PCA is 84.9% and LDA is 84.6%. The performance of them is similar.

- Part 3: Face Recognition with Kernel PCA and Kernel LDA

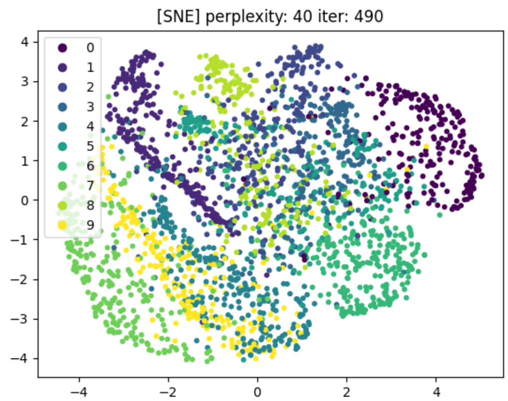
Face Recognition: PCA (Linear Kernel)	Face Recognition: LDA (Linear Kernel)
[k=1] 24/30 => acc: 0.80 [k=2] 24/30 => acc: 0.80 [k=3] 25/30 => acc: 0.83 [k=4] 25/30 => acc: 0.83 [k=5] 25/30 => acc: 0.83 [k=6] 25/30 => acc: 0.83 [k=7] 24/30 => acc: 0.80 [k=8] 25/30 => acc: 0.83 [k=9] 25/30 => acc: 0.83 [k=10] 24/30 => acc: 0.80 Average = 0.818	[k=1] 22/30 => acc: 0.73 [k=2] 18/30 => acc: 0.60 [k=3] 20/30 => acc: 0.67 [k=4] 21/30 => acc: 0.70 [k=5] 22/30 => acc: 0.73 [k=6] 22/30 => acc: 0.73 [k=7] 22/30 => acc: 0.73 [k=8] 21/30 => acc: 0.70 [k=9] 21/30 => acc: 0.70 [k=10] 21/30 => acc: 0.70 Average = 0.699

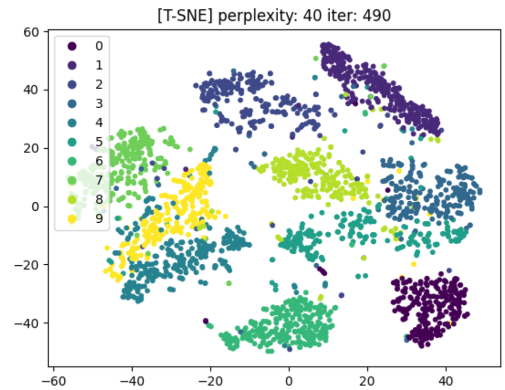
Face Recognition: PCA (RBF Kernel)	Face Recognition: LDA (RBF Kernel)
[k=1] 25/30 => acc: 0.83 [k=2] 25/30 => acc: 0.83 [k=3] 25/30 => acc: 0.83 [k=4] 25/30 => acc: 0.83 [k=5] 24/30 => acc: 0.80 [k=6] 23/30 => acc: 0.77 [k=7] 23/30 => acc: 0.77 [k=8] 24/30 => acc: 0.80 [k=9] 25/30 => acc: 0.83 [k=10] 24/30 => acc: 0.80 Average = 0.809	[k=1] 23/30 => acc: 0.77 [k=2] 22/30 => acc: 0.73 [k=3] 21/30 => acc: 0.70 [k=4] 22/30 => acc: 0.73 [k=5] 21/30 => acc: 0.70 [k=6] 22/30 => acc: 0.73 [k=7] 21/30 => acc: 0.70 [k=8] 20/30 => acc: 0.67 [k=9] 20/30 => acc: 0.67 [k=10] 20/30 => acc: 0.67 Average = 0.707

I find that performance of PCA is better than LDA regardless of kernel used. Additionally, original PCA and LDA are better than their kernel ones.

2-2. Experiment Result & Discussion: t-SNE

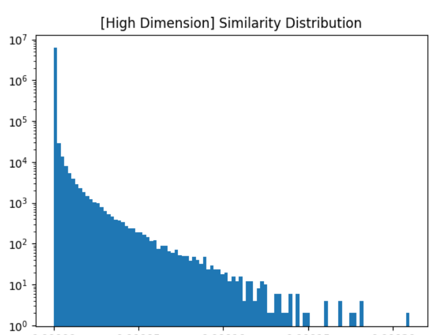
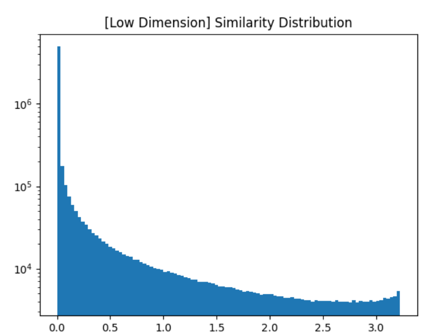
- Part 1: Symmetric SNE and t-SNE & Part 2: Visualize Embedding

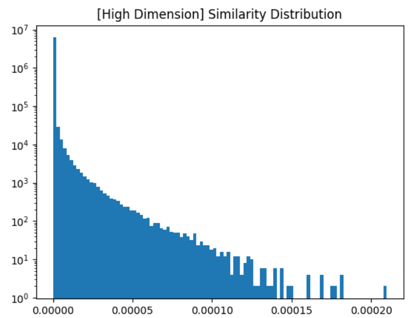
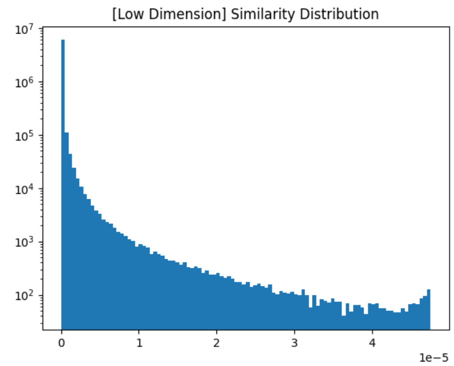




I find that SNE method suffers from severe crowded problem, and the coordinates of data points are between -4 and 4. With using Student-T distribution in low-dimensional space, t-SNE does not have crowded problem, and the coordinates of data points are between -60 and 60, which is larger than the range in SNE.

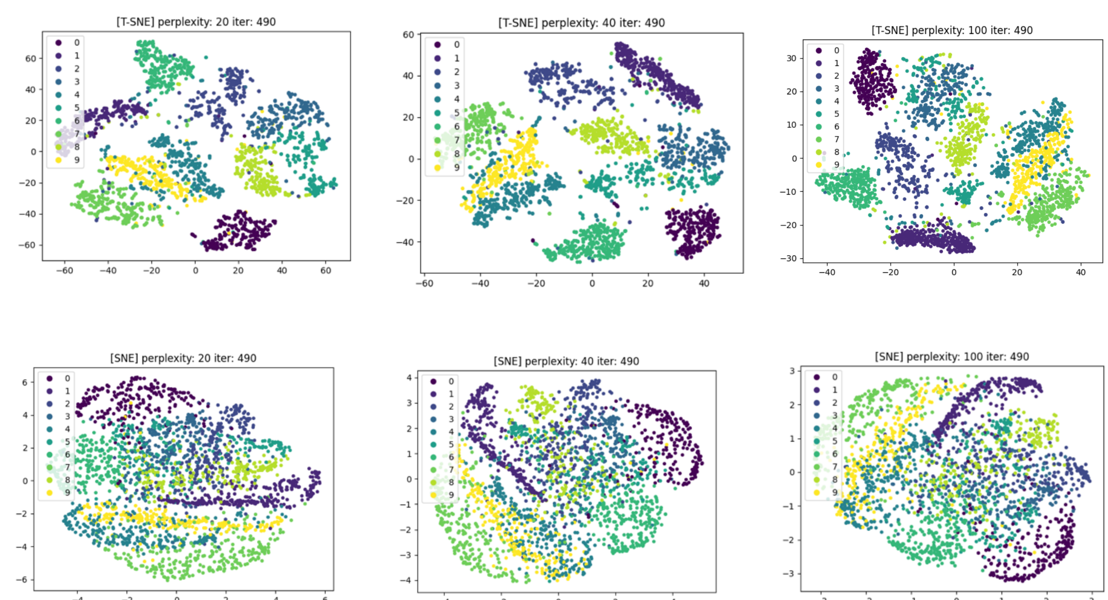
- Part 3: Visualize Pairwise Similarities

[SNE] High Dimension	[SNE] Low Dimension
	

[t-SNE] High Dimension	[t-SNE] Low Dimension
	

The similarity distribution in high-dimensional space is same in SNE and t-SNE. However, the similarity distribution in t-SNE is larger than the one in SNE in low-dimensional space. I think the reason is Student-T distribution is a long-tail distribution. When the distance between two data points is large enough, the probability of Student-T distribution is larger than normal distribution.

- Part 4: Try Different Perplexity



The first row in above image shows the result of t-SNE with different perplexity values. From the course, I learn that the perplexity is related to the number of neighbor data points of one data point. Hence, if the perplexity is small, the group in low-dimensional space is loose, and if the perplexity is large, the group is tight.

The second row in above image shows the result of SNE with different perplexity values. However, because SNE suffers from crowded problem, I cannot find the difference given different perplexity values.

3-1. Observation and Discussion

- Meaning of Eigenfaces
  Eigenfaces is a set of “standard faces” extracted from a large set of face images. Each face image in original dataset can be viewed as a combination of these standard faces. In other words, we can use the small set of standard faces to represent original face images dataset. Eigenfaces can be calculated by performing PCA on images. The key step is to calculate covariance of original image dataset, and conduct eigen-decomposition on this covariance matrix.

- Crowded Problem in Symmetric SNE
  In symmetric SNE, the different groups of data points are crowded in lo-dimensional space, which make us difficult to differentiate data points from different classes. The solution to this problem is replace gaussian distribution with Student-T distribution in low-dimensional space.