1. Code with detailed explanations (40%)

Kernel Eigenfaces (25%)

Part1 (10%)

PCA:

In PCA, derive the projection matrix W by computing the eigenvectors of the covariance matrix in the high-dimensional space, selecting the q eigenvectors corresponding to the q largest eigenvalues. Display the first 25 eigenfaces based on these eigenvectors and reconstruct 10 randomly chosen images.

```
def PCA(train_images, train_labels, test_images, test_labels):
    """
    train_images : (# of images, features)
    """
    # get the covariance matrix of the training dataset(high-dim)
    small_cov = get_covariance(train_images)

    # get the eigenvectors of the covariance matrix
    normalized_eigenvectors = get_eigenvectors(train_images, small_cov)

    # show the first 25 eigenfaces and use the eigenvectors to reconstruct the images
    transfrom_to_faces(normalized_eigenvectors[:, :25])
    show_reconstruction(train_images, normalized_eigenvectors)

    # transfer the dataset into low-dimensional space and do the face recognition
    z_train = train_images @ normalized_eigenvectors
    z_test = test_images @ normalized_eigenvectors
    face_recognition(z_train, train_labels, z_test, test_labels, k)
```

get_covariance():

When the covariance matrix is too large, computing its eigenvectors directly becomes time- and memory-intensive. Instead of calculating the eigenvectors for a (features,features)-sized matrix, we compute the eigenvectors of a (# of datapoints, # of datapoints)-sized matrix. This approach, often referred to as the dual space trick in PCA, is more efficient for datasets with many features but fewer data points.

```
def get_covariance(data):
    num_datapoints = data.shape[0]
    mean = np.mean(data, axis=0)
    diff = data - mean
    # Perform PCA with memory-efficient covariance matrix computation. origin: diff_train.T @ diff_train
    small_cov = diff @ diff.T / num_datapoints
    return small_cov
```

get_eigenvectors():

To compute the eigenvalues and eigenvectors efficiently, we use the smaller matrix. Once the eigenvectors of the small matrix are obtained, they are mapped back to the original feature space using the formula:

$$\mathbf{u}_i = \mathbf{X}^T \mathbf{v}_i$$

where u_i is the eigenvector of the origin covariance. X^T is the transpose of the data matrix, v_i is the eigenvector of the small matrix. Finally, sort the eigenvalues in descending order to rearrange the corresponding eigenvectors.

```
def get_eigenvectors(data, small_cov):
    eigenvalues, eigenvectors = np.linalg.eig(small_cov)

    # turn the eigenvectors of the small matrix back to origin feature space's eigenvectors
    normalized_eigenvectors = data.T @ eigenvectors / np.linalg.norm(data.T @ eigenvectors, axis=0)

    # sort eigenvalue from large to small
    indices = np.argsort(eigenvalues)[::-1]
    normalized_eigenvectors = normalized_eigenvectors[:, indices]
    return normalized_eigenvectors
```

transform_to_faces():

In this function we reshaped the first 25 eigenvectors to the size of images and showed all of them.

```
def transfrom_to_faces(vectors):
    eigenfaces = np.copy(vectors.T)
    fig, axes = plt.subplots( nrows: 5,  ncols: 5, figsize=(15, 5))
    axes = axes.flat
    for i, ax in enumerate(axes):
        eigenface = eigenfaces[i].reshape((image_height, image_width))
        eigenface = (eigenface - eigenface.min()) / (eigenface.max() - eigenface.min())
        ax.imshow(eigenface, cmap='gray')
        ax.axis('off')
    plt.tight_layout()
    plt.show()
```

show_reconstruction():

First random pick 10 images and then for each image it transfers to low dimension first and then transfer back into high dimension to get the reconstruction of it by using this formula:

$$xWW^\top$$

Finally, show the original images and corresponding reconstruction images.

```python
def show_reconstruction(images, normalized_eigenvectors):
    pick = set()
    while len(pick) < 10:
        pick.add(random.randint( a: 0, images.shape[0] - 1))

    fig, axes = plt.subplots( nrows: 10,  ncols: 2, figsize=(5, 15))
    for i, idx in enumerate(pick):
        # reconstruct the image by transfer to low dimension first and then transfer back into high dimension
        reconstruct_image = images[idx] @ normalized_eigenvectors @ normalized_eigenvectors.T
        axes[i, 0].imshow(images[idx].reshape(image_height, image_width), cmap='gray')
        axes[i, 0].axis("off")
        axes[i, 1].imshow(reconstruct_image.reshape(image_height, image_width), cmap='gray')
        axes[i, 1].axis('off')
    plt.tight_layout()
    plt.show()
```

LDA:

First, apply PCA to reduce the dimensionality, as the goal of PCA is to lower dimensions and reduce computational complexity. Next, use the dimensionality-reduced data from PCA to compute the LDA eigenvectors. Once the LDA eigenvectors are obtained, display the first 25 eigenfaces and use these eigenvectors to reconstruct the images.

```python
def LDA(train_images, train_labels, test_images, test_labels):
    """
    train_images : (# of images, features)
    """
    # do PCA first to reduce the dimension since the number of features are too large
    small_cov = get_covariance(train_images)
    pca_eigenvectors = get_eigenvectors(train_images, small_cov)
    pca_matrix = train_images @ pca_eigenvectors

    # get LDA eigenvectors
    num_group = 15
    eigenvectors = get_LDA_eigenvectors(pca_matrix, train_labels)
    eigenvectors = pca_eigenvectors @ eigenvectors

    # show the first 25 eigenfaces and use the eigenvectors to reconstruct the images
    transfrom_to_faces(eigenvectors[:, :25])
    show_reconstruction(train_images, eigenvectors[:, :num_group - 1])

    # use the eigenvectors to transfer to low dimension and then do the recognition
    z_train = train_images @ eigenvectors[:, :num_group - 1]
    z_test = test_images @ eigenvectors[:, :num_group - 1]
    face_recognition(z_train, train_labels, z_test, test_labels, k)
```

get_LDA_eigenvectors():

To get LDA eigenvectors, we need to calculate the S_W and S_B , the formula of S_W and S_B are as follows,

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

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

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

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

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

After obtaining these two matrix, the way to get the eigenvectors of projection matrix W is let

$$S_W^{-1} S_B \text{ as } W$$

Finally, sort the eigenvalues in descending order to rearrange the corresponding eigenvectors.

```python
def get_LDA_eigenvectors(matrix, labels):
    num_group = 15
    num_features = matrix.shape[1]
    S_W = np.zeros((num_features, num_features))
    S_W += 1e-6 * np.eye(S_W.shape[0])
    S_B = np.zeros((num_features, num_features))
    overall_mean = np.mean(matrix, axis=0)
    for i in range(num_group):
        groupI = matrix[labels == i]
        mean = np.mean(groupI, axis=0)
        S_W += (groupI - mean).T @ (groupI - mean)
        num_groupI = groupI.shape[0]
        mean_diff = (mean - overall_mean).reshape(1, -1)
        S_B += num_groupI * mean_diff.T @ mean_diff

    W = np.linalg.inv(S_W) @ S_B
    eigenvalues, eigenvectors = np.linalg.eig(W)
    eigenvalues = eigenvalues.real
    eigenvectors = eigenvectors.real.astype('float')
    # print(eigenvectors.dtype)
    indices = np.argsort(eigenvalues)[::-1]
    eigenvectors = eigenvectors[:, indices]
    return eigenvectors
```

Part 2 (5%)

First, construct the low-dimensional representations of the training and testing datasets using

$$z = Wx$$

Then, perform classification on the testing dataset using the k-Nearest Neighbors (k-NN) algorithm based on the low-dimensional features. In LDA, the dimensionality reduction must result in a number of dimensions fewer than the number of classes; otherwise, LDA may not effectively capture all the discriminative directions between classes.

In PCA:

```python
# transfer the dataset into low-dimensional space and do the face recognition
z_train = train_images @ normalized_eigenvectors
z_test = test_images @ normalized_eigenvectors
face_recognition(z_train, train_labels, z_test, test_labels, k)
```

In LDA:

```
# use the eigenvectors to transfer to low dimension and then do the recognition
z_train = train_images @ eigenvectors[:, :num_group - 1]
z_test = test_images @ eigenvectors[:, :num_group - 1]
face_recognition(z_train, train_labels, z_test, test_labels, k)
```

```
def face_recognition(z_train, train_labels, z_test, test_labels, k=1):
    correct = 0
    for i in range(z_test.shape[0]):
        distances = np.linalg.norm(z_test[i] - z_train, axis=1)
        dis = np.argsort(distances)
        unique, count = np.unique(train_labels[dis[:k]], return_counts=True)
        idx = np.argmax(count)
        if test_labels[i] == unique[idx]:
            correct += 1
    print(f'performance:{correct / z_test.shape[0]}')
```

Part 3 (10%)

kernelPCA:

First, compute the kernel matrix for the training dataset based on the specified kernel type. Next, derive the eigenvectors of the kernel matrix to form the projection matrix. Finally, use the projection matrix to obtain the low-dimensional representations of both the training and testing datasets in the kernel space, and perform classification on the testing dataset using the kernel-based features.

```
def kernelPCA(train_images, train_labels, test_images, test_labels, kernel_type):
    # get the kernel matrix of training dataset
    train_kernel = get_kernel_function(train_images, train_images, kernel_type)

    # calculate the eigenvectors
    eigenvalues, eigenvectors = np.linalg.eig(train_kernel)
    indices = np.argsort(eigenvalues)[::-1]
    eigenvectors = eigenvectors[:, indices]

    # use the eigenvectors to transfer to low dimension and then do the recognition
    test_kernel = get_kernel_function(test_images, train_images, kernel_type)
    z_train = train_kernel @ eigenvectors[:, :100]
    z_test = test_kernel @ eigenvectors[:, :100]
    face_recognition(z_train, train_labels, z_test, test_labels, k)
```

get_kernel_function():

We use two kernel matrix for the kernelPCA and kernelLDA, one is RBF matrix, another is polynomial kernel matrix.

RBF: $\exp(-\gamma \|x-y\|^2)$

Polynomial: $(x^T y)^d$

For easy computation we need to center the kernel matrix by using this formula:

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

1_N is NxN matrix with every element $1/N$

However, for the testing kernel, X should be testing data Y should be training data, this cause the kernel matrix is not symmetry. Therefore, 1_N need some modification according to their size.

```python
def get_kernel_function(X, Y, kernel_type):
    # num_points = images.shape[0]
    # construct kernel matrix
    K = np.zeros((X.shape[0], Y.shape[0]))
    if kernel_type == 'rbf':
        # gamma = 1.0 / X.shape[1]
        gamma = 0.00001
        for i in range(X.shape[0]):
            for j in range(Y.shape[0]):
                diff = X[i] - Y[j]
                K[i, j] = np.exp(-gamma * diff.dot(diff))
    else:
        degree = 3
        for i in range(X.shape[0]):
            for j in range(Y.shape[0]):
                K[i, j] = np.power(X[i].dot(Y[j]), degree)

    # center kernel
    n_samples_X = X.shape[0]
    n_samples_Y = Y.shape[0]

    one_n_X = np.ones((n_samples_X, n_samples_X)) / n_samples_X
    one_n_Y = np.ones((n_samples_Y, n_samples_Y)) / n_samples_Y

    K_centered = K - one_n_X @ K - K @ one_n_Y + one_n_X @ K @ one_n_Y
    return K_centered
```

kernelLDA:

First, compute the kernel matrix for the training dataset based on the specified kernel type. Next, derive

the eigenvectors of the kernel matrix to form the projection matrix. The function get_LDA_eigenvectors is as same as in LDA. Finally, use the projection matrix to obtain the low-dimensional representations of both the training and testing datasets in the kernel space, and perform classification on the testing dataset using the kernel-based features.

```
def kernelLDA(train_images, train_labels, test_images, test_labels, kernel_type):
    # get the kernel matrix of training dataset
    train_kernel = get_kernel_function(train_images, train_images, kernel_type)

    # get LDA eigenvectors
    eigenvectors = get_LDA_eigenvectors(train_kernel, train_labels)

    # use the eigenvectors to transfer to low dimension and then do the recognition
    z_train = train_kernel @ eigenvectors
    test_kernel = get_kernel_function(test_images, train_images, kernel_type)
    z_test = test_kernel @ eigenvectors
    face_recognition(z_train, train_labels, z_test, test_labels, k)
```

t-SNE (15%)

Part 1 (10%)

The parts that need to be modified are the calculation of q_{ij} and the gradient of y

q_{ij}:

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

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

```
num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y)) # (1+||a-b||^2)^-1 tsne
num = np.exp(-(sum_Y + num + sum_Y.reshape(-1, 1)))  # exp(-||a-b||^2) ssne
```

y:

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

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

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

Part 2 (2%)

For every 50 or 100 iterations, save the images of the scatter plot of 2D space data according to their labels.

```
if (iter + 1) % 50 == 0:
    pylab.clf()
    # Draw a scatter plot
    pylab.scatter(Y[:, 0], Y[:, 1], s: 20, labels)
    pylab.title(f'Iteration {iter + 1}')
    pylab.savefig(f'symSNE_progress/iter_{iter + 1}.png')
```

After that, use these images to construct the gif of the whole process

```
def create_gif():
    images = []
    filenames = sorted(os.listdir('symSNE_progress'),
                       key=lambda x: int(x.split('_')[1].split('.')[0]))

    for filename in filenames:
        if filename.endswith('.png'):
            file_path = os.path.join('symSNE_progress', filename)
            images.append(Image.open(file_path))

    images[0].save( fp: 'symSNE_animation.gif',
                   save_all=True,
                   append_images=images[1:],
                   duration=500,
                   loop=0)

    print("GIF animation created successfully!")
```

Part 3 (2%)

```
def visualize_similarity_distributions(P, Q):
    """
    Visualize and compare the distributions of pairwise similarities
    in both high-dimensional (P) and low-dimensional (Q) spaces.

    Parameters:
    -----------
    P : numpy array
        Pairwise similarities in high-dimensional space
    Q : numpy array
        Pairwise similarities in low-dimensional space
    """
    # Flatten the matrices and remove diagonal elements
    P_flat = P.flatten()
    Q_flat = Q.flatten()

    # Create figure with two subplots
    fig, (ax1, ax2) = plt.subplots( nrows: 2, ncols: 1, figsize=(10, 8))

    # Plot P distribution (high-dimensional)
    ax1.hist(P_flat, bins=50, color='blue', log=True)
    ax1.set_xlabel('Similarity (log scale)')
    ax1.set_ylabel('Count')
    ax1.set_title('High-dimensional (P) Distribution')
    ax1.grid(True)
```

```
# Plot Q distribution (low-dimensional)
ax2.hist(Q_flat, bins=50, color='red', log=True)
ax2.set_xlabel('Similarity (log scale)')
ax2.set_ylabel('Count')
ax2.set_title('Low-dimensional (Q) Distribution')
ax2.grid(True)

# Add statistics text
stats_text = f"""
    High-dimensional (P):
    Mean: {P_flat.mean():.2e}
    Std: {P_flat.std():.2e}

    Low-dimensional (Q):
    Mean: {Q_flat.mean():.2e}
    Std: {Q_flat.std():.2e}

    KL Divergence: {stats.entropy(P_flat, Q_flat):.2e}
    """
plt.figtext( x: 1.02,  y: 0.5, stats_text, fontsize=10, va='center')

# Set overall title
title = "Distribution of Pairwise Similarities"
fig.suptitle(title, y=1.02)

plt.tight_layout()
return fig
```

Part 4 (1%)

just change the variable of perplexity

```
perplexity = 20.0
Y = symSNE(X,  no_dims: 2,  initial_dims: 50, perplexity)
```

```
perplexity = 5.0
Y = tsne(X,  no_dims: 2,  initial_dims: 50, perplexity)
```

2. Experiments and Discussion (50%)

Kernel Eigenfaces (20%)

Part 1 (5%)

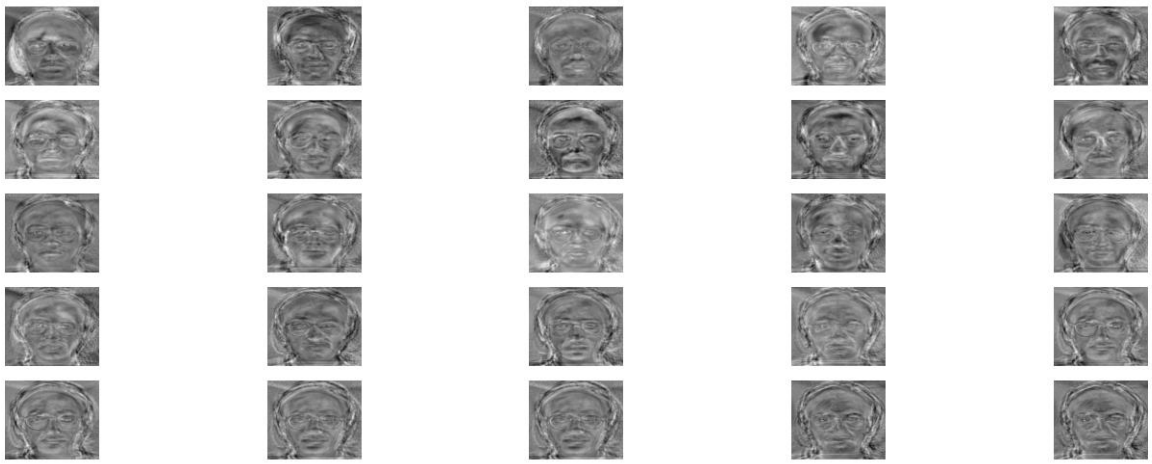
First 25 eigenfaces

Reconstructed faces of PCA

The left column represents the original images, while the right column shows the corresponding

reconstructed faces. These reconstructed faces were obtained by reducing the dimensionality of the features to 100.

First 25 fisher faces:



Reconstructed faces of LDA

The left column represents the original images, while the right column shows the corresponding reconstructed faces. These reconstructed faces were obtained by reducing the dimensionality of the features to number of classes -1 (=14).

Part 2 (5%)

parameter for k nearest neighbors: k=3

PCA:

reducing the dimensionality of the features to 100.

```
accuracy:0.8333333333333334
```

LDA:

reducing the dimensionality of the features to 14(# of classes -1).

```
accuracy:1.0
```

Part 3 (5%)

parameter for k nearest neighbors: k=3

kernelPCA:

reducing the dimensionality of the features to 100.

RBF : gamma = 1/number of features = 0.0000222

```
accuracy:0.13333333333333333
```

Polynomial: degree = 5

```
accuracy:0.3
```

kernelLDA:

reducing the dimensionality of the features to 14(number of classes -1).

RBF: gamma = 1/number of features = 0.0000222

```
accuracy:0.13333333333333333
```

Polynomial: degree = 3

```
accuracy:0.13333333333333333
```

Discussion:

The performance of the kernel method is significantly worse than that of simple PCA and LDA. I experimented with different kernel parameters, but the results did not improve. I see three possible explanations for this:

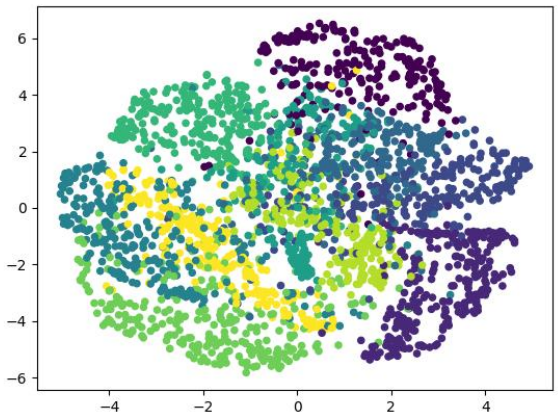
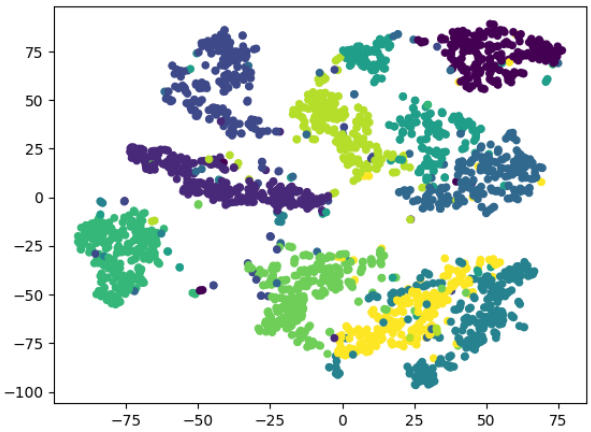
1. Implementation Error: There may be an error in my kernel method implementation.
2. Inappropriate Kernel Choice: The chosen kernel might not be suitable for this dataset, as it may fail to accurately represent the similarities between data points.
3. Linear Nature of the Data: Kernels are generally more effective for non-linear data distributions. If the dataset has a predominantly linear distribution, methods like PCA and LDA might perform better.

This is supported by my test results. For example, simple LDA achieved an accuracy of 1.0, indicating that standard LDA was already sufficient to preserve the critical features of the dataset. This suggests that the dataset is likely closer to a linear distribution.

Additionally, there was little difference in performance between the RBF and polynomial kernels. At best, the polynomial kernel performed slightly better in kernel PCA, but it still fell significantly short of the performance of standard PCA.

t-SNE (30%)

Part 1 (5%) & (5%) Please discuss the observation in this part.

Symmetry SNE	t-SNE
	

The key difference between SSNE and t-SNE lies in the method of calculating Q. SSNE uses a normal distribution to convert the distances between data points into probabilities, while t-SNE employs a t-distribution. The t-distribution has longer tails, meaning that in the low-dimensional space, data points need to be further apart to achieve a low probability. This characteristic allows t-SNE to address the crowding problem encountered in symmetric SNE.

Part 2 (5%)

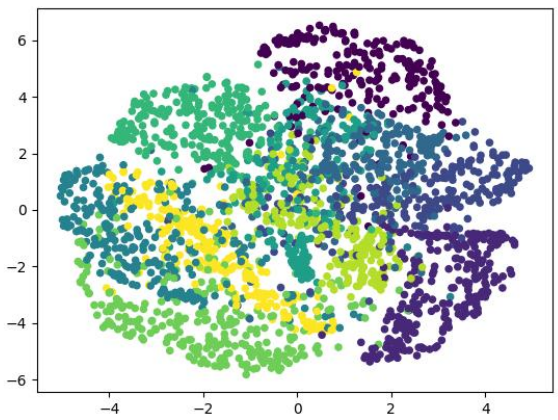
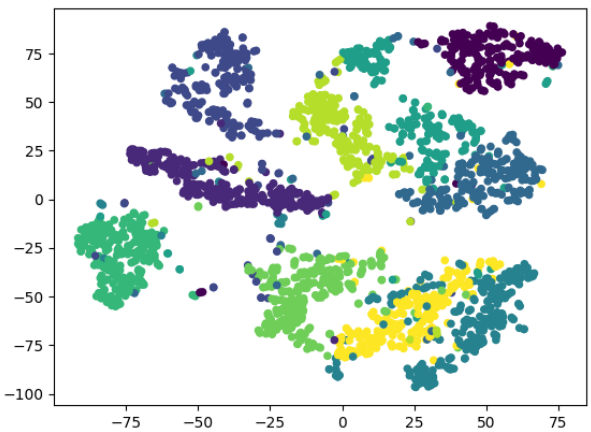
Perplexity = 20

Target dimension = 2

initial dimension(reduce by PCA) = 50

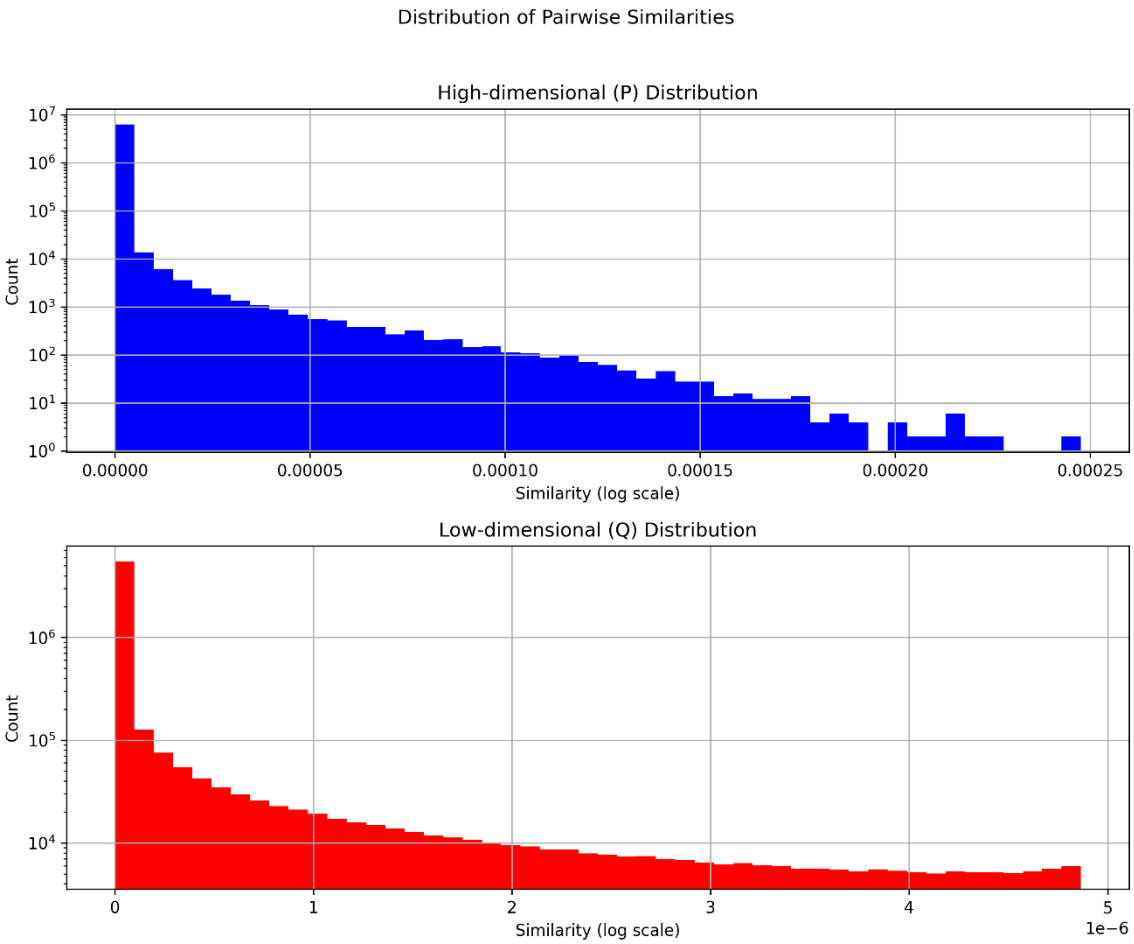
iterations: SSNE:500, t-SNE:1000

The reason that SSNE use smaller iteration is because it has higher convergence speed. **(Observations and Discussion)**

Symmetry SNE	t-SNE
	

Part 3 (5%)

Symmetry SNE:

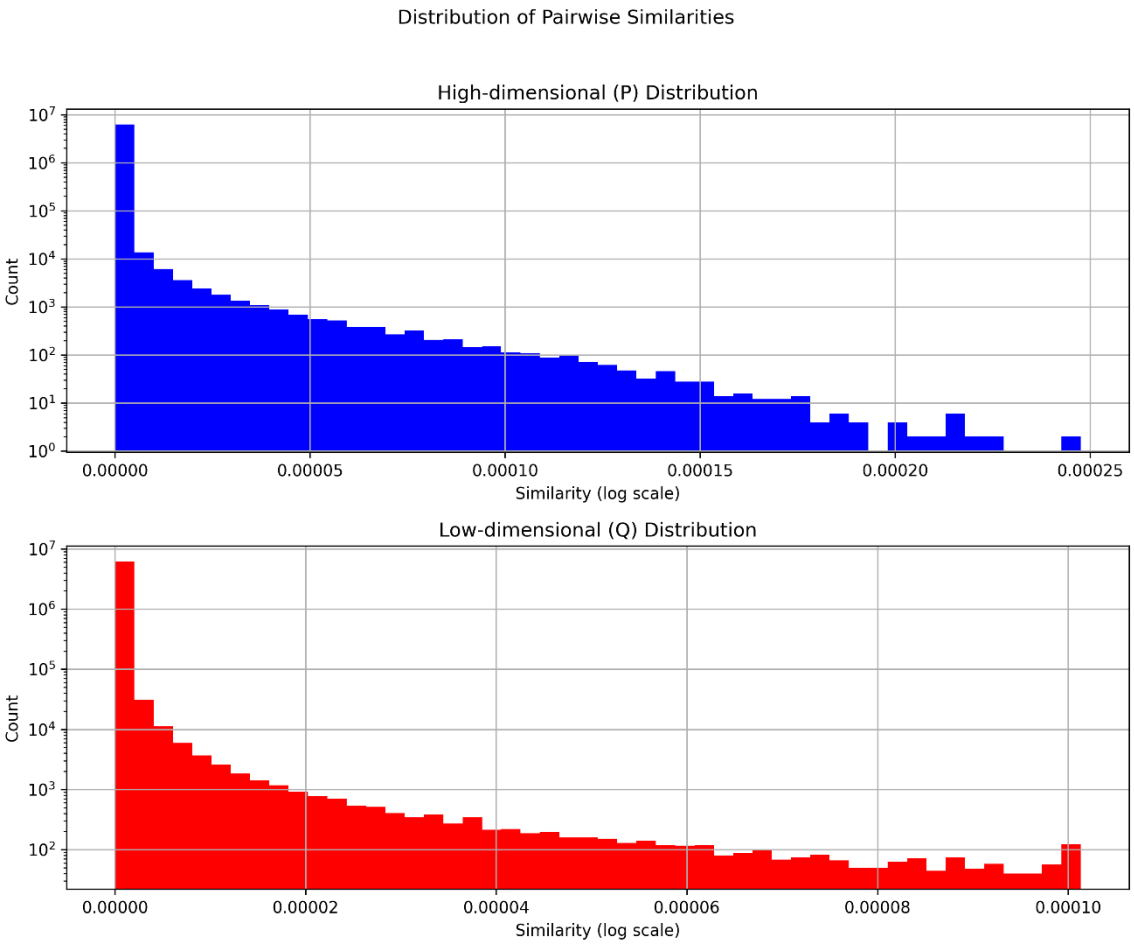


High-dimensional (P):
Mean: 1.60e-07
Std: 2.54e-06

Low-dimensional (Q):
Mean: 1.60e-07
Std: 6.17e-07

KL Divergence: 2.11e+00

t-SNE:



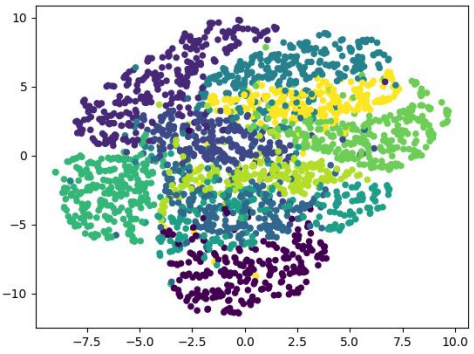
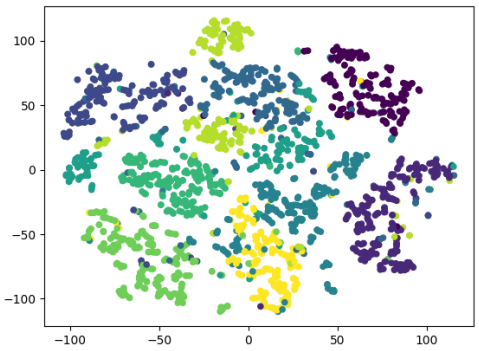
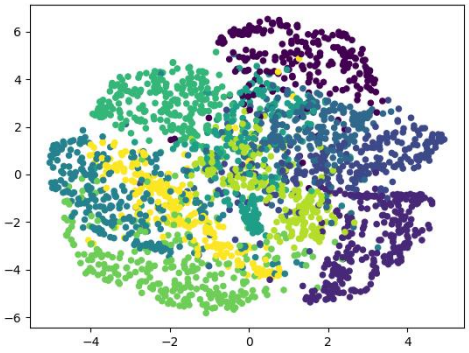
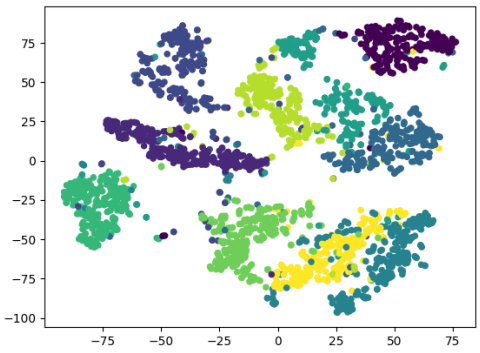
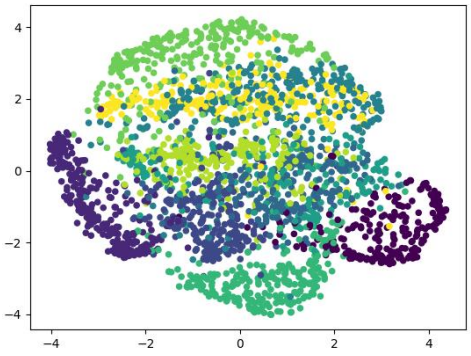
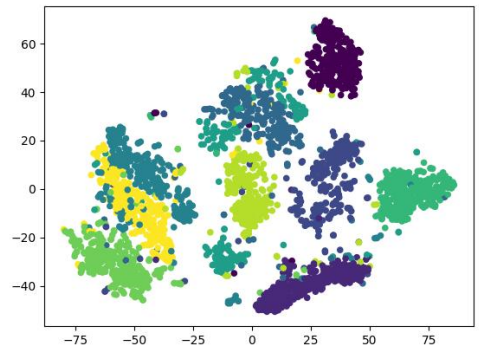
High-dimensional (P):
Mean: 1.60e-07
Std: 2.54e-06

Low-dimensional (Q):
Mean: 1.60e-07
Std: 1.76e-06

KL Divergence: 1.02e+00

It is clear that t-SNE effectively separates similarities corresponding to distances, whereas the similarity distribution in SSNE appears more scattered. **(Observations and Discussion)**

Part 4 (5%) & (5%) Please discuss the observation in this part.

Perplexity	Symmetry SNE	t-SNE
5		
20		
50		

It can be observed that when the perplexity is higher, points within the same group tend to distribute more evenly. Conversely, when the perplexity is lower, the same group forms smaller, tighter local clusters. This happens because perplexity measures the effective neighborhood size, representing the approximate number of neighbors for each data point. A lower perplexity means fewer effective neighbors, causing closer points to be grouped together, while points farther away, even with the same label, may not be considered part of the same local cluster. Additionally, due to SSNE's crowding problem, the effects of different perplexity values are less noticeable in its visualizations.

Observations and Discussion (10%)

The reconstructed faces in PCA are better than those in LDA. I believe this is because the maximum dimensionality reduction in LDA is constrained by the number of classes. As a result, when reducing the dimensions and mapping back to the original feature space, much of the finer detail is lost.

PCA reconstruction	LDA reconstruction
	

Each eigenface represents a basis vector in a lower-dimensional subspace that captures the most significant variations in the dataset. These variations often correspond to features such as the general shape of the face, shadows, or other patterns in facial structure.

Summary of SSNE and t-SNE

Features	SSNE	t-SNE
Low-Dimensional Similarity	Gaussian distribution	Student's t-distribution (heavier tails)
Crowding Problem	Poor handling	Better handling
Preservation	Emphasizes local structure	Balances local and global structure
Visual Quality	Less interpretable	Highly interpretable
Parameter Sensitivity	**Parameter Sensitivity**	**Parameter Sensitivity**