I. Kernel Eigenfaces / Fisherfaces

Part 1. Eigenfaces and Fisherfaces / Image reconstruction

PCA eigenfaces

```
PCA_file = './Result/PCA_LDA/PCA/'
W_PCA, mean_PCA = PCA(train_data, k=dim)
EigenVisualFaces(W_PCA, PCA_file + 'eigenfaces/', k=dim)
ReconstructFace(W_PCA, mean_PCA, train_data, PCA_file + 'reconstruct/')
for i in range(1, 20, 2):
    acc += faceRecongnition(W_PCA, mean_PCA, train_data, test_data, i)
```

I write **PCA** process involves centering the data, computing the covariance matrix, finding the eigenvalues and eigenvectors of the covariance matrix, projecting these eigenvectors back to the original data space, normalizing the eigenvectors, sorting them in descending order of their eigenvalues, and selecting the top k eigenvectors as the principal components. This process effectively reduces the dimensionality of the data while retaining the most important variance.

```
def PCA(data, k=25):
    mean = np.mean(data, axis=0)
    centered_data = data - mean
    cov = (centered_data) @ (centered_data).T

    eigenvalue, eigenvector = np.linalg.eig(cov)
    eigenvector = data.T @ eigenvector

    # Normalize W
    for i in range(len(eigenvector[0])):
        eigenvector[:,i] = eigenvector[:,i] / np.linalg.norm(eigenvector[:,i])

    # 按eigenvalues降序排列
    idx = np.argsort(eigenvalue)[::-1]
    eigenvector = eigenvector[:, idx]
    W = eigenvector[:, :k].real

    return W, mean
```

LDA fisherfaces

```
scalar = 3
labels = np.repeat(np.arange(1, subject_num + 1), train_num)
LDA_file = './Result/PCA_LDA/LDA/'
data = Reduction(train_data, scalar)
compress_test = Reduction(test_data, scalar)

W_LDA, mean_LDA = LDA(data, labels, k=dim)
EigenVisualFaces(W_LDA, LDA_file + 'fisherfaces/', k=dim, S=scalar)
ReconstructFace(W_LDA, None, data, LDA_file + 'reconstruct/', S=scalar)
for i in trange(1, 20, 2):
    acc += faceRecongnition(W_LDA, None, data, compress_test, i)
```

This **LDA** process involves computing the within-class and between-class scatter matrices, finding the eigenvalues and eigenvectors of the matrix *Sw^−1*Sb*, normalizing the eigenvectors, sorting them in descending order of their eigenvalues, and selecting the top k eigenvectors as the discriminant vectors. This process aims to maximize the separation between different classes while minimizing the scatter within each class.

$$S_w = \sum_{i=1}^{c} \sum_{x \in C_i} (x - \mu_i)(x - \mu_i)^T \qquad S_b = \sum_{i=1}^{c} n_i (\mu_i - \mu)(\mu_i - \mu)^T$$

```
def LDA(data, labels, k=25):
    (n, d) = data.shape
    labels = np.asarray(labels)
    c = np.unique(labels)
    mean = np.mean(data, axis=0)
    Sw = np.zeros((d, d), dtype=np.float64)
    Sb = np.zeros((d, d), dtype=np.float64)
    # Compute within-class and between-class scatter matrices
    for i in c:
        X_i = data[np.where(labels == i)[0], :]
        mu_i = np.mean(X_i, axis=0)
        Sw += (X_i - mu_i).T @ (X_i - mu_i)
        Sb += X_i.shape[0] * (mu_i - mean).T @ (mu_i - mean)
    # Compute eigenvalues and eigenvectors
    eigen_val, eigen_vec = np.linalg.eig(np.linalg.pinv(Sw) @ Sb)
    # Normalize eigenvectors
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])

    # Sort eigenvectors by eigenvalues in descending order
    idx = np.argsort(eigen_val)[::-1]
    W = eigen_vec[:, idx][:, :k].real

    return W, mean
```

The **Reduction** function aims to reduce the size of each image by averaging $S \, x \, S$ blocks of pixels. Essentially, it compresses each $S \, x \, S$ pixel block into a single pixel whose value is the average of the block, thus achieving dimensionality reduction. This helps to reduce the data size, making subsequent computations more efficient

```
def Reduction(data, S):
    num_images = len(data)
    reduced_height = height // S
    reduced_width = width // S

    d = np.zeros((num_images, reduced_height, reduced_width))

    for n in range(num_images):
        img = data[n].reshape(height, width)

        for i in range(0, height, S):
            for j in range(0, width, S):
                block = img[i:i+S, j:j+S]
                block_mean = np.mean(block)
                d[n, i // S, j // S] = block_mean

    return d.reshape(num_images, -1)
```

Visualization and reconstruction

For each of the first k faces, reshape the column vector into an image and save it. Create a larger figure and add each of the first k faces as subplots in a grid.

```
def EigenVisualFaces(W, file_path, k=25, S=1):
    # Save individual faces
    for i in range(k):
        img = W[:,i].reshape(height//S, width//S)
        plt.imsave(f'{file_path}eigenface_{i:02d}.jpg', img, cmap='gray')

    # Create and save composite figure
    fig, axes = plt.subplots(int(np.sqrt(k)), int(np.sqrt(k)), figsize=(12, 9))
    for i, ax in enumerate(axes.flat):
        if i < k:
            img = W[:,i].reshape(height//S, width//S)
            ax.imshow(img, cmap='gray')
            ax.axis('off')
    plt.tight_layout()
    fig.savefig(f'{file_path}../eigenfaces_{k}.jpg')
    plt.show()
```

Selecting random faces, reconstructing them, saving individual comparisons, and creating a composite image of all reconstructed faces.

$$\hat{x} = (x - \mu)WW^T + \mu$$

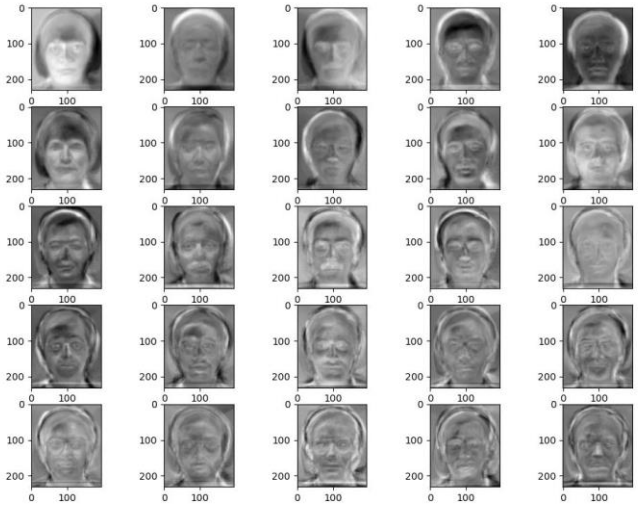
where x^ is the reconstructed face, x is the original face, μ is the mean vector, and W is the matrix of eigenfaces.

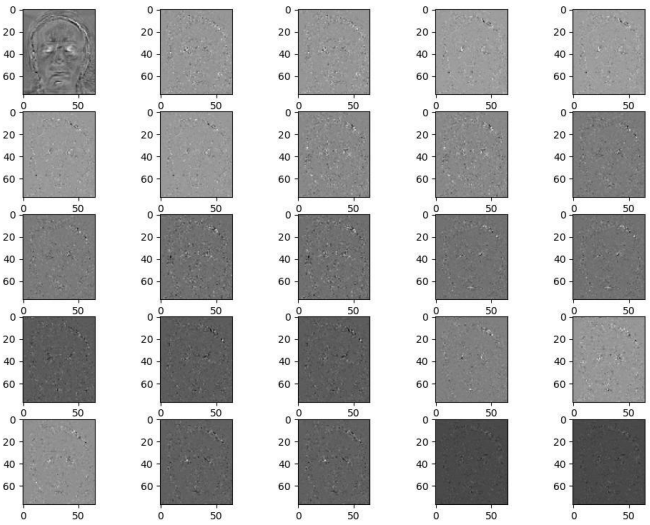
```python
def ReconstructFace(W, mean, data, file_path, S=1):
    if mean is None:
        mean = np.zeros(W.shape[0])

    sel = np.random.choice(subject_num * train_num, 10, replace=False)
    img = []

    for index in sel:
        x = data[index].reshape(1, -1)
        reconstruct = (x - mean) @ W @ W.T + mean
        img.append(reconstruct.reshape(height//S, width//S))

        # Plot and save original and reconstructed face side-by-side
        fig, ax = plt.subplots(1, 2)
        ax[0].imshow(x.reshape(height//S, width//S), cmap='gray')
        ax[1].imshow(reconstruct.reshape(height//S, width//S), cmap='gray')
        fig.tight_layout()
        fig.savefig(f'{file_path}reconfaces_{len(img)}.jpg')
        plt.close(fig)

    # Plot all reconstructed faces together
    fig, axes = plt.subplots(2, 5, figsize=(10, 4))
    for i, ax in enumerate(axes.flat):
        if i < len(img):
            ax.imshow(img[i], cmap='gray')
    fig.tight_layout()
    fig.savefig(f'{file_path}../reconfaces.jpg')
    plt.show()
```

Result

- **Eigenfaces / Fisherfaces**

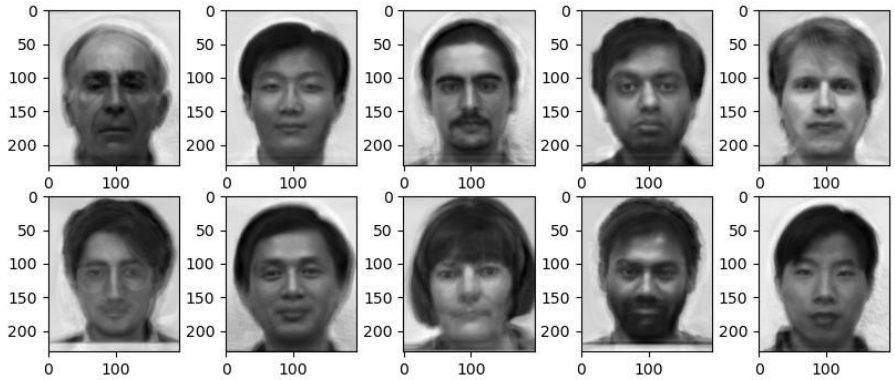
When we look at the eigenfaces, we can see that they capture not just the essential facial features, but also the lighting and shadows in the images. In contrast, the fisherfaces seem to highlight only the general outline of the faces. This because PCA (which we use to create eigenfaces) is designed to find the biggest differences across all the images, capturing a wide range of features, including lighting variations. LDA (which creates fisherfaces), however, focuses on finding the differences that matter the most for distinguishing between different people. So, fisherfaces are aimed to capture just those key features that help tell one person apart from another.

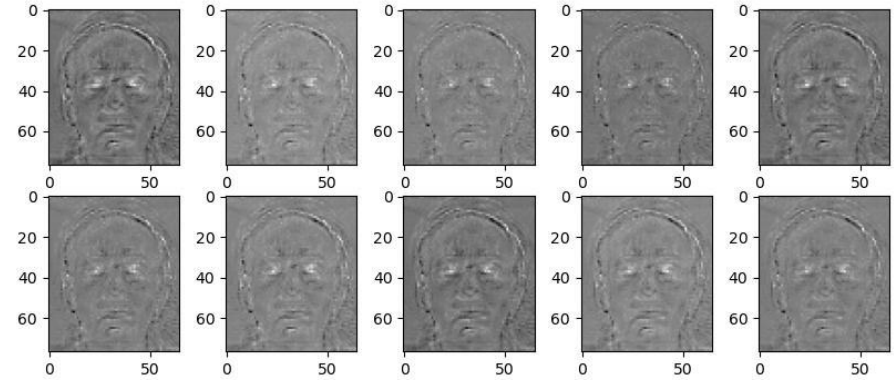




- **Reconstruction PCA / LDA**

  As we discussed above, eigenfaces do a great job of capturing the detailed facial features across all images. With just the first 25 eigenfaces, we can achieve a pretty good reconstruction of the original faces. However, with fisherfaces (from LDA), it's a different story. Since LDA is focused on finding features that distinguish one person from another, it doesn't capture the fine details as well. As a result, the reconstructed images using fisherfaces don't look as close to the original faces.





Part 2. Face recognition

faceRecongnition

Project the training and testing data onto the lower-dimensional subspace spanned by the eigenfaces. where X_low is the low-dimensional representation, X is the original data, μ is the mean vector, and W is the matrix of eigenfaces.

$$X_{\text{low}} = (X - \mu)W$$

Loop Through Each Test Sample

1. Compute the distance between the current test sample and all training samples.
2. Identify the K nearest neighbors based on the computed distances.
3. Perform majority voting to predict the class of the test sample.
4. Compare the predicted class with the true class and count errors.

```python
def faceRecongnition(W, mean, train_data, test_data, K):
    if mean is None:
        mean = np.zeros(W.shape[0])

    # Project training and testing data into the lower-dimensional space
    low_train = (train_data - mean) @ W
    low_test = (test_data - mean) @ W

    err = 0
    for i in range(len(low_test)):
        dist = np.linalg.norm(low_train - low_test[i], axis=1)
        nearest = np.argsort(dist)[:K]
        vote = np.bincount(nearest // train_num, minlength=subject_num)
        predict = np.argmax(vote) + 1
        if predict != i // 2 + 1:
            err += 1

    accuracy = 1 - err / len(low_test)
    print(f"K={K}: Accuracy:{accuracy:.4f} ({len(low_test) - err}/{len(low_test)})")
    return accuracy
```

Result

It's clear that PCA performs better overall compared to LDA. This is a bit unexpected because LDA is supposed to be good at distinguishing between different classes. One possible explanation is that LDA's feature space is too small with only 25 dimensions.

PCA and KNN

```
K=1: Accuracy:0.8333 (25/30)
K=3: Accuracy:0.8333 (25/30)
K=5: Accuracy:0.9000 (27/30)
K=7: Accuracy:0.9000 (27/30)
K=9: Accuracy:0.8000 (24/30)
K=11: Accuracy:0.8000 (24/30)
K=13: Accuracy:0.8333 (25/30)
K=15: Accuracy:0.8000 (24/30)
K=17: Accuracy:0.7667 (23/30)
K=19: Accuracy:0.7667 (23/30)
Average accuracy: 0.8233
```

LDA and KNN

```
K=1: Accuracy:0.8000 (24/30)
K=3: Accuracy:0.7333 (22/30)
K=5: Accuracy:0.7667 (23/30)
K=7: Accuracy:0.7333 (22/30)
K=9: Accuracy:0.7333 (22/30)
K=11: Accuracy:0.7333 (22/30)
K=13: Accuracy:0.6667 (20/30)
 70%|
K=15: Accuracy:0.7000 (21/30)
K=17: Accuracy:0.6667 (20/30)
K=19: Accuracy:0.7333 (22/30)
100%|
Average accuracy: 0.7267
```

Part 3. Face recognition (with kernel)

kernelPCA

1. Compute the kernel matrix *K* using the specified kernel function.

$$K_c = K - 1_N K - K 1_N + 1_N K 1_N$$

2. Compute the eigenvalues and eigenvectors of the kernel matrix *K*.
3. Normalize each eigenvector by dividing it by its Euclidean norm.
4. Sort the eigenvectors based on their corresponding eigenvalues in descending order and select the top *k* eigenvectors.
5. Return the matrix of principal components *W* and the kernel matrix *K*.

```
def kernelPCA(data, kernel_type, k=25):
    K = computeKernel(data, data, kernel_type)

    eigenvalues, eigenvectors = np.linalg.eig(K)

    eigenvectors = eigenvectors / np.linalg.norm(eigenvectors, axis=0)

    idx = np.argsort(-eigenvalues)
    W = eigenvectors[:, idx][:, :k].real

    return W, K
```

kernelLDA

1. Compute the kernel matrix *K* using the specified kernel function.
2. Create matrix *Z*, which is used in the between-class scatter matrix calculation.
3. Compute *Sw* and *Sb* in the kernel space.

$$S_w = KK^T \quad S_b = KZK^T$$

4. Compute the eigenvalues and eigenvectors of the matrix *Sw^−1 * Sb* using the pseudoinverse of *Sw*.

$$S_w^{-1} S_b v = \lambda v$$

5. Normalize each eigenvector by dividing it by its Euclidean norm.
6. Sort the eigenvectors based on their corresponding eigenvalues in descending order and select the top *k* eigenvectors.

```python
def kernelLDA(data, kernel_type, k=25):
    K = computeKernel(data, data, kernel_type)

    Z = np.full((len(data), len(data)), 1 / train_num)

    Sw = K @ K.T
    Sb = K @ Z @ K.T

    eigenvalues, eigenvectors = np.linalg.eig(np.linalg.pinv(Sw) @ Sb)
    eigenvectors = eigenvectors / np.linalg.norm(eigenvectors, axis=0)

    idx = np.argsort(-eigenvalues)
    W = eigenvectors[:, idx][:, :k].real

    return W, K
```

kernelFaceRecongnition

1.For each test sample, compute the distance to all training samples.

2.Identify the K nearest neighbors based on the computed distances.

3.Perform majority voting to predict the class of the test sample.

4.Compare the predicted class with the true class and count errors.

```python
def kernelFaceRecongnition(W, train_data, test_data, kernel_type, kernel, K):
    # Project training and testing data into the lower-dimensional space
    low_train = kernel @ W
    K_test = computeKernel(test_data, train_data, kernel_type)
    low_test = K_test @ W

    # KNN
    err = 0
    for i in range(len(low_test)):
        vote = np.zeros(subject_num, dtype=int)
        dist = distance(low_test[i], low_train)
        nearest = np.argsort(dist)[:K]
        for n in nearest:
            vote[n // train_num] += 1
        predict = np.argmax(vote) + 1
        if predict != i // 2 + 1:
            err += 1
    accuracy = 1 - err / len(low_test)
    print(f"K={K}: Accuracy:{accuracy:.4f} ({len(low_test) - err}/{len(low_test)})")
    return accuracy
```

Result

We noticed that both the linear and polynomial kernels performed better than the RBF kernel in both kernel PCA and kernel LDA. Interestingly, traditional PCA outperformed all three types of kernel methods. Specifically, for kernel LDA, the polynomial and RBF kernels did better than standard LDA. Additionally, the average accuracy of kernel PCA was higher than that of kernel LDA, similar to what we saw in Part 2 on face recognition.

Kernel PCA (linear) and KNN

```
K=1: Accuracy:0.8000 (24/30)
K=3: Accuracy:0.8333 (25/30)
K=5: Accuracy:0.8333 (25/30)
K=7: Accuracy:0.8000 (24/30)
K=9: Accuracy:0.8333 (25/30)
K=11: Accuracy:0.8333 (25/30)
K=13: Accuracy:0.8000 (24/30)
K=15: Accuracy:0.8000 (24/30)
K=17: Accuracy:0.7333 (22/30)
K=19: Accuracy:0.7333 (22/30)
Average accuracy: 0.8000
```

Kernel LDA (linear) and KNN

```
K=1: Accuracy:0.7000 (21/30)
K=3: Accuracy:0.6667 (20/30)
K=5: Accuracy:0.6667 (20/30)
K=7: Accuracy:0.6333 (19/30)
K=9: Accuracy:0.6333 (19/30)
K=11: Accuracy:0.6000 (18/30)
K=13: Accuracy:0.5333 (16/30)
K=15: Accuracy:0.7000 (21/30)
K=17: Accuracy:0.6667 (20/30)
K=19: Accuracy:0.6000 (18/30)
Average accuracy: 0.6400
```

Kernel PCA (polynomial) and KNN

```
K=1: Accuracy:0.8333 (25/30)
K=3: Accuracy:0.8333 (25/30)
K=5: Accuracy:0.8667 (26/30)
K=7: Accuracy:0.8333 (25/30)
K=9: Accuracy:0.7000 (21/30)
K=11: Accuracy:0.7000 (21/30)
K=13: Accuracy:0.7667 (23/30)
K=15: Accuracy:0.7000 (21/30)
K=17: Accuracy:0.7000 (21/30)
K=19: Accuracy:0.6333 (19/30)
Average accuracy: 0.7567
```

Kernel LDA (polynomial) and KNN

```
K=1: Accuracy:0.6667 (20/30)
K=3: Accuracy:0.6000 (18/30)
K=5: Accuracy:0.6000 (18/30)
K=7: Accuracy:0.6667 (20/30)
K=9: Accuracy:0.6667 (20/30)
K=11: Accuracy:0.6667 (20/30)
K=13: Accuracy:0.6333 (19/30)
K=15: Accuracy:0.6667 (20/30)
K=17: Accuracy:0.7000 (21/30)
K=19: Accuracy:0.7333 (22/30)
Average accuracy: 0.6600
```

Kernel PCA (RBF) and KNN

```
K=1: Accuracy:0.8000 (24/30)
K=3: Accuracy:0.7667 (23/30)
K=5: Accuracy:0.7667 (23/30)
K=7: Accuracy:0.8000 (24/30)
K=9: Accuracy:0.7667 (23/30)
K=11: Accuracy:0.8000 (24/30)
K=13: Accuracy:0.8333 (25/30)
K=15: Accuracy:0.8000 (24/30)
K=17: Accuracy:0.7000 (21/30)
K=19: Accuracy:0.7667 (23/30)
Average accuracy: 0.7800
```

Kernel LDA (RBF) and KNN

```
K=1: Accuracy:0.6000 (18/30)
K=3: Accuracy:0.6333 (19/30)
K=5: Accuracy:0.6333 (19/30)
K=7: Accuracy:0.5667 (17/30)
K=9: Accuracy:0.6333 (19/30)
K=11: Accuracy:0.6333 (19/30)
K=13: Accuracy:0.5667 (17/30)
K=15: Accuracy:0.6333 (19/30)
K=17: Accuracy:0.6333 (19/30)
K=19: Accuracy:0.6333 (19/30)
Average accuracy: 0.6167
```

II. t-SNE

Part 1. t-SNE and Symmetric SNE

t-SNE: Uses a heavy-tailed Student's t-distribution with one degree of freedom to compute the pairwise affinities in the low-dimensional space. This helps to alleviate the crowding problem by allowing moderate distances between points in the high-dimensional space to also be represented by moderate distances in the low-dimensional space.
Symmetric SNE: Uses a Gaussian distribution to compute the pairwise affinities in both the high-dimensional and low-dimensional spaces. This can lead to the crowding problem, where many points are crowded together in the center of the low-dimensional map.

Low-Dimensional Affinities

t-SNE

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

Symmetric SNE

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

```
# Compute pairwise affinities >> qij
sum_Y = np.sum(np.square(Y), 1)
num = -2. * np.dot(Y, Y.T)
if _type == 't-SNE':
    num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
if _type == 'Symmetric SNE':
    num = np.exp(-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)
```

Gradient

t-SNE

$$\frac{\partial C}{\partial y_i} = 4\sum_j (p_{ij} - q_{ij})(y_i - y_j)q_{ij}$$

Symmetric SNE

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

```
PQ = P - Q
for i in range(n):
    if _type == 't-SNE':
        dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
    if _type == 'Symmetric SNE':
        dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

Observation

The cost function in **t-SNE** converged more smoothly and quickly, indicating that the optimization process was more efficient.
In **symmetric SNE**, the cost function showed more instability and slower convergence. This might be due to the crowding problem, which makes it harder for the algorithm to find an optimal embedding.

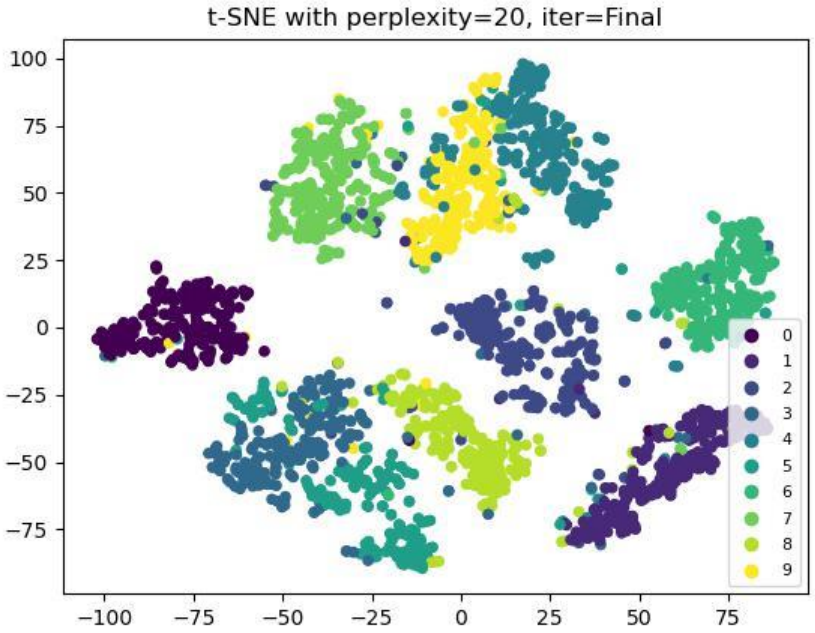
Part 2. Visualization of embedding

visualize

This function generates a scatter plot of the 2D representation of the data produced by t-SNE or symmetric SNE, colors the points according to their labels, adds a legend and title, displays the plot, and saves it as an image file. This helps in visually assessing the quality of the low-dimensional embedding and how well the clusters are separated.

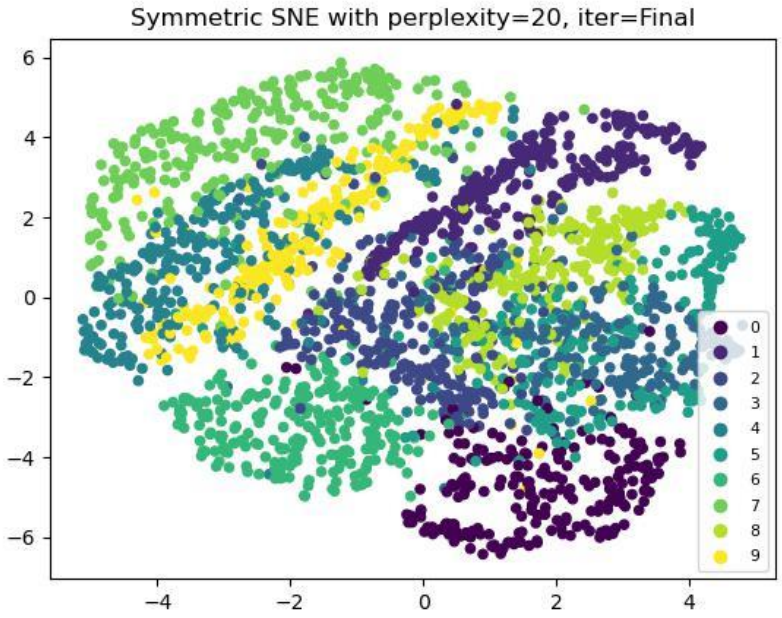
```
def visualize(Y, labels, no_dims, perplexity, _type, _iter):
    fig = plt.figure()
    scatter = plt.scatter(Y[:, 0], Y[:, 1], s=20, c=labels)
    plt.legend(*scatter.legend_elements(), loc='lower right', prop={'size': 7.8})
    plt.title(f'{_type} with perplexity={perplexity}, iter={_iter}')
    plt.show()
    fig.savefig(f'{file_path}/{_type}/per_{perplexity}/{_iter}.jpg')
```

t-SNE with perplexity=20



1. The points are more spread out with clear gaps between clusters.
2. The clusters are distinct and well-separated, effectively reflecting the local and global structure of the high-dimensional data.
3. Utilizes Student's t-distribution (with one degree of freedom), which helps to reduce the crowding problem, resulting in a more even distribution of points in the low-dimensional space.

symmetric SNE with perplexity=20



1. The points appear more crowded, especially in the center area, with less clear boundaries between clusters.
2. Many points overlap, making it difficult to distinguish between different clusters, leading to a loss of detailed data structure.
3. Uses Gaussian distribution, which in the low-dimensional space cannot effectively spread out the points, leading to the crowding problem.

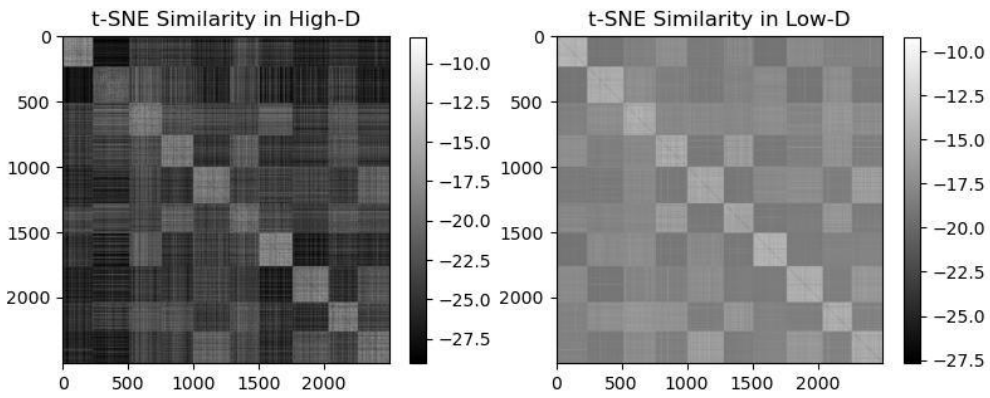
In summary, t-SNE provides a clearer representation of the data's cluster structure compared to symmetric SNE. This advantage is mainly due to t-SNE's use of the heavy-tailed Student's t-distribution, which alleviates the crowding problem in the low-dimensional space.

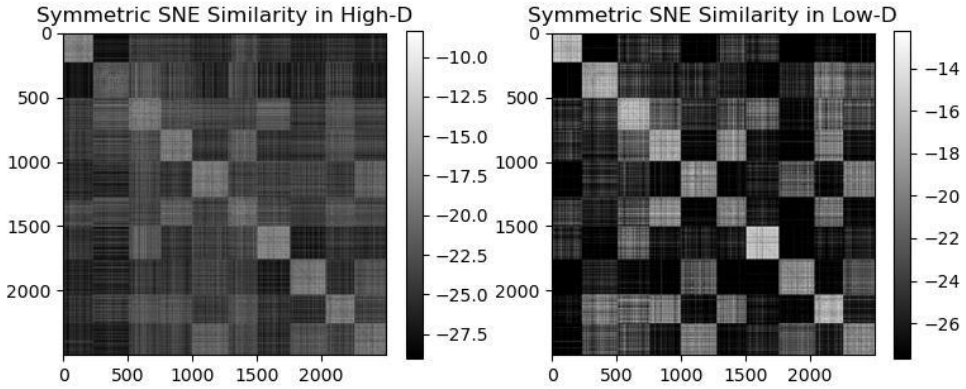
Part 3. Visualization of pairwise similarity

plotSimilarity

This function generates a visual comparison of the similarity matrices in high-dimensional and low-dimensional spaces. By sorting the matrices based on the labels and plotting them side by side.

```python
def plotSimilarity(P, Q, labels, _type, perplexity):
    fig, ax = plt.subplots(1, 2, figsize=(8, 4))

    idx = labels.argsort()
    sortP = P[:, idx][idx]
    sortQ = Q[:, idx][idx]

    ax[0].set_title(f'{_type} Similarity in High-D')
    img = ax[0].imshow(np.log(sortP), cmap='gray')
    fig.colorbar(img, ax=ax[0], shrink=0.7)

    ax[1].set_title(f'{_type} Similarity in Low-D')
    img = ax[1].imshow(np.log(sortQ), cmap='gray')
    fig.colorbar(img, ax=ax[1], shrink=0.7)

    fig.tight_layout()
    fig.savefig(f'{file_path}/{_type}/per_{perplexity}/_similarity.jpg')
```

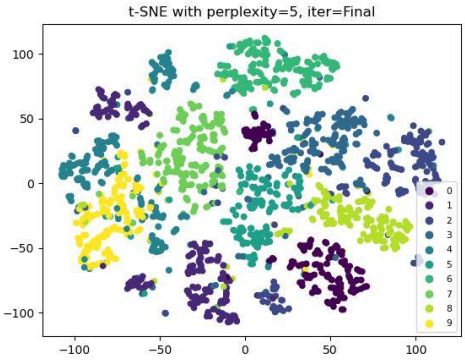


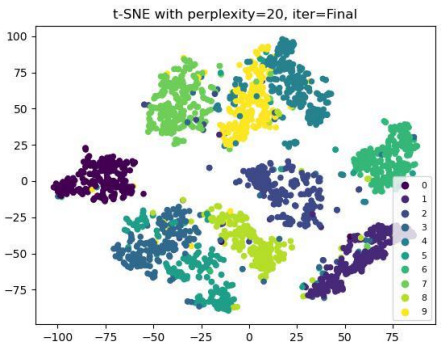


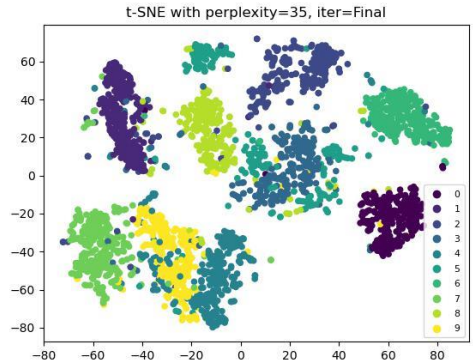
Part 4. Play with different perplexity

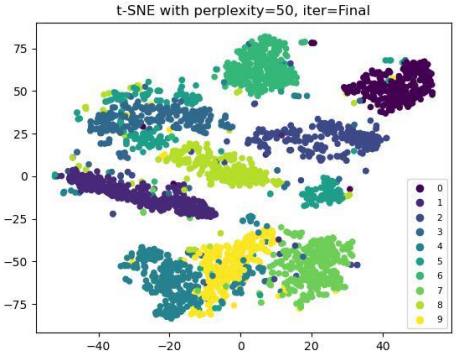
Result

t-SNE



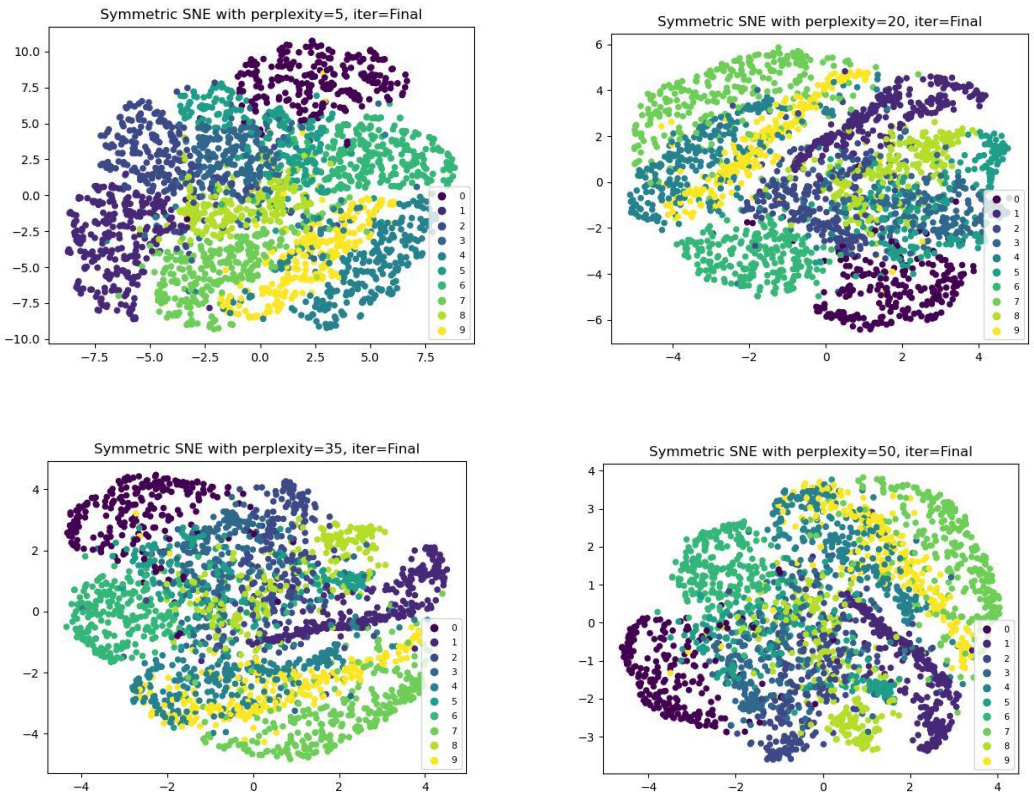






From these images, we can observe that as the perplexity value increases, the point distribution in the t-SNE visualization becomes increasingly tight, with more points within each cluster and smaller gaps between clusters. At lower perplexity values, the visualization focuses more on preserving local structures, while at higher perplexity values, the visualization shows better global structures. This indicates that the perplexity value plays a crucial role in balancing local and global structures in the t-SNE visualization.

symmetric SNE



From these images, we can observe that as the perplexity value increases, the point distribution in the symmetric SNE visualization becomes tighter, with points within clusters closer together and slightly better-defined cluster boundaries. However, the crowding problem is still present, especially compared to t-SNE visualizations. This indicates that while higher perplexity values help to reduce some of the crowding, symmetric SNE still struggles to separate clusters effectively, leading to a less clear overall structure.