HW7

312554054 林辰翰

Kernel Eigenfaces:

Simple_PCA & Kernel_PCA

```
def simple_PCA(num_image, images):
    image_transpose = images.T
    mean = np.mean(image_transpose, axis=1)
    mean = np.tile(mean.T, (num_image, 1)).T
    difference = image_transpose - mean
    covariance = difference.dot(difference.T) / num_image
    return covariance
```

```
def kernel_PCA(images, kernel_type, gamma):
    if kernel_type == 'linear':
        kernel = images.T.dot(images)
    elif kernel_type == 'rbf':
        kernel = np.exp(-gamma * cdist(images.T, images.T, 'sqeuclidean'))
    else:
        raise BaseException(f'Invalid kernel type. The kernel type should be linear or rbf')

    matrix_n = np.ones((29 * 24, 29 * 24), dtype=float) / (29 * 24)
    matrix = kernel - matrix_n.dot(kernel) - kernel.dot(matrix_n) + matrix_n.dot(kernel).dot(matrix_n)
    return matrix
```

The kernel_PCA function calculates the kernel matrix for images by combining spatial and color similarities. It first computes the squared Euclidean distance between all pairs of pixels based on their color values to determine color similarity. Simultaneously, it creates a grid of pixel coordinates and computes the squared Euclidean distance between pixel pairs based on their spatial coordinates to determine spatial similarity. The function then calculates the color similarity and spatial similarity using exponential functions scaled by factors gamma_c and gamma_s, respectively.

PCA

```python
def PCA(train_images, train_labels, test_images, test_labels, mode, k_neighbors, kernel_type, gamma):
    num_train = len(train_images)
    if mode == 'simple':
        matrix = simple_PCA(num_train, train_images)
    elif mode == 'kernel':
        matrix = kernel_PCA(train_images, kernel_type, gamma)
    else:
        raise BaseException(f'Invalid mode. The mode should be simple or kernel')

    eigenvec = find_eigenvector(matrix)
    eigenface(eigenvec, 0)
    reconstruction(num_train, train_images, eigenvec)
    classify(num_train, len(test_images), train_images, train_labels, test_images, test_labels, eigenvec, k_neighbors)
    plt.tight_layout()
    plt.show()
```

The PCA function performs Principal Component Analysis (PCA) on training images, projecting them onto principal components, reconstructing images, and classifying test images using k-nearest neighbors. It begins by determining the number of training images. Based on the mode ('simple' or 'kernel'), it computes either a covariance matrix for simple PCA with simple_PCA or a kernel matrix for Kernel PCA with kernel_PCA, using kernel_type and gamma. It then finds the eigenvectors of the matrix, visualizes them as eigenfaces with eigenface, and reconstructs the training images using reconstruction. The function classifies test images by projecting them onto the principal components and using k-nearest neighbors with classify.

Simple_LDA & Kernel_LDA

```python
def simple_LDA(num_of_each_class, images, labels):
    overall_mean = np.mean(images, axis=0)
    n_class = len(num_of_each_class)
    class_mean = np.zeros((n_class, 29 * 24))
    for label in range(n_class):
        class_mean[label, :] = np.mean(images[labels == label + 1], axis=0)

    scatter_b = np.zeros((29 * 24, 29 * 24), dtype=float)
    for idx, num in enumerate(num_of_each_class):
        difference = (class_mean[idx] - overall_mean).reshape((29 * 24, 1))
        scatter_b += num * difference.dot(difference.T)

    scatter_w = np.zeros((29 * 24, 29 * 24), dtype=float)
    for idx, mean in enumerate(class_mean):
        difference = images[labels == idx + 1] - mean
        scatter_w += difference.T.dot(difference)

    matrix = np.linalg.pinv(scatter_w).dot(scatter_b)
    return matrix
```

```python
def kernel_LDA(num_of_each_class, images, labels, kernel_type, gamma):
    n_class = len(num_of_each_class)
    n_image = len(images)

    if kernel_type == 'linear':
        kernel_of_each_class = np.zeros((n_class, 29 * 24, 29 * 24))
        for idx in range(n_class):
            image = images[labels == idx + 1]
            kernel_of_each_class[idx] = image.T.dot(image)
        kernel_of_all = images.T.dot(images)
    elif kernel_type == 'rbf':
        kernel_of_each_class = np.zeros((n_class, 29 * 24, 29 * 24))
        for idx in range(n_class):
            image = images[labels == idx + 1]
            kernel_of_each_class[idx] = np.exp(-gamma * cdist(image.T, image.T, 'sqeuclidean'))
        kernel_of_all = np.exp(-gamma * cdist(images.T, images.T, 'sqeuclidean'))
    else:
        raise BaseException(f'Invalid kernel type. The kernel type should be linear or rbf')

    matrix_n = np.zeros((29 * 24, 29 * 24))
    identity_matrix = np.eye(29 * 24)
    for idx, num in enumerate(num_of_each_class):
        matrix_n += kernel_of_each_class[idx].dot(identity_matrix - num * identity_matrix).dot(kernel_of_each_class[idx].T)

    matrix_m_i = np.zeros((n_class, 29 * 24))
    for idx, kernel in enumerate(kernel_of_each_class):
        for row_idx, row in enumerate(kernel):
            matrix_m_i[idx, row_idx] = np.sum(row) / num_of_each_class[idx]

    matrix_m_star = np.zeros(29 * 24)
    for idx, row in enumerate(kernel_of_all):
        matrix_m_star[idx] = np.sum(row) / n_image

    matrix_m = np.zeros((29 * 24, 29 * 24))
    for idx, num in enumerate(num_of_each_class):
        difference = (matrix_m_i[idx] - matrix_m_star).reshape((29 * 24, 1))
        matrix_m += num * difference.dot(difference.T)

    matrix = np.linalg.pinv(matrix_n).dot(matrix_m)
    return matrix
```

The simple_LDA function computes the Linear Discriminant Analysis (LDA) matrix by first calculating the overall mean of the images and the mean of each class. It then computes the between-class scatter matrix (scatter_b) and the within-class scatter matrix (scatter_w). The final LDA matrix is obtained by multiplying the pseudoinverse of scatter_w with scatter_b. In contrast, the kernel_LDA function calculates the LDA matrix using kernel methods for non-linear data. Depending on the kernel_type (either 'linear' or 'rbf'), it computes kernel matrices for each class and the overall dataset. It then constructs a normalization matrix and adjusts these kernel matrices. The function computes the mean kernel matrices for each class and the overall data, and finally calculates the between-class scatter matrix (matrix_m) and the within-class scatter matrix (matrix_n). The LDA matrix is derived by multiplying the pseudoinverse of matrix_n with matrix_m. Both functions ultimately produce a matrix that can be used for dimensionality reduction and classification.

LDA

```python
def LDA(train_images, train_labels, test_images, test_labels, mode, k_neighbors, kernel_type, gamma):
    n_train = len(train_images)
    _, num_of_each_class = np.unique(train_labels, return_counts=True)

    if mode == 'simple':
        matrix = simple_LDA(num_of_each_class, train_images, train_labels)
    elif mode == 'kernel':
        matrix = kernel_LDA(num_of_each_class, train_images, train_labels, kernel_type, gamma)
    else:
        raise BaseException(f'Invalid mode. The mode should be simple or kernel')

    target_eigenvectors = find_eigenvector(matrix)
    eigenface(target_eigenvectors, 1)
    reconstruction(n_train, train_images, target_eigenvectors)
    classify(n_train, len(test_images), train_images, train_labels, test_images, test_labels, target_eigenvectors, k_neighbors)
    plt.tight_layout()
    plt.show()
```

The LDA function performs Linear Discriminant Analysis on training images, visualizes the resulting discriminant vectors, reconstructs images, and classifies test images using the k-nearest neighbors algorithm. First, it determines the number of training images and computes the number of images in each class. Based on the specified mode ('simple' or 'kernel'), it calls either simple_LDA or kernel_LDA to compute the LDA matrix. simple_LDA handles linear data, while kernel_LDA is used for non-linear data and requires kernel_type and gamma parameters. The function then finds the eigenvectors of the computed LDA matrix using find_eigenvector, which represent the discriminant vectors. These vectors are visualized as "eigenfaces" using the eigenface function. The function reconstructs the original training images using these vectors through the reconstruction function.

Find eigenvectors

```python
def find_eigenvector(matrix):
    eigenvalues, eigenvectors = np.linalg.eig(matrix)
    idx = np.argsort(eigenvalues)[::-1][:25]
    eigenvec = eigenvectors[:, idx].real
    return eigenvec

def eigenface(eigenvectors, mode):
    faces = eigenvectors.T.reshape((25, 29, 24))
    fig = plt.figure(1)
    # fig.canvas.set_window_title(f'{"Eigenfaces" if mode == 0 else "Fisherfaces"}')
    fig.suptitle(f'{"Eigenfaces" if mode == 0 else "Fisherfaces"}')
    for idx in range(25):
        plt.subplot(5, 5, idx + 1)
        plt.axis('off')
        plt.imshow(faces[idx, :, :], cmap='gray')
```

The find_eigenvector function computes the eigenvalues and eigenvectors of a

given matrix, sorts the eigenvalues in descending order, selects the top 25 eigenvectors, and returns their real parts. The eigenface function visualizes these eigenvectors by reshaping them into 29x24 pixel images. It creates a figure titled "Eigenfaces" or "Fisherfaces" based on the mode and displays the top 25 eigenvectors as grayscale images in a 5x5 grid, turning off the axes for clarity. This visualizes the principal or discriminant components from PCA or LDA.

Face reconstruction

```python
def reconstruction(num_image, images, eigenvec):
    reconstruct_image = np.zeros((10, 29 * 24))
    choice = np.random.choice(num_image, 10)
    for idx in range(10):
        reconstruct_image[idx, :] = images[choice[idx], :].dot(eigenvec).dot(eigenvec.T)

    fig = plt.figure(2)
    # fig.canvas.set_window_title('Reconstructed faces')
    fig.suptitle('Reconstructed faces')
    for idx in range(10):
        plt.subplot(10, 2, idx * 2 + 1)
        plt.axis('off')
        plt.imshow(images[choice[idx], :].reshape(29, 24), cmap='gray')

        plt.subplot(10, 2, idx * 2 + 2)
        plt.axis('off')
        plt.imshow(reconstruct_image[idx, :].reshape(29, 24), cmap='gray')

def decorrelate(num_image, images, eigenvec):
    decorrlated_image = np.zeros((num_image, 25))
    for idx, image in enumerate(images):
        decorrlated_image[idx, :] = image.dot(eigenvec)
    return decorrlated_image

def classify(num_train, num_test, train_images, train_labels, test_images, test_labels, eigenvec, k_neighbors):
    decorrelate_train = decorrelate(num_train, train_images, eigenvec)
    decorrelate_test = decorrelate(num_test, test_images, eigenvec)

    error = 0
    distance = np.zeros(num_train)

    for test_idx, test_image in enumerate(decorrelate_test):
        for train_idx, train_image in enumerate(decorrelate_train):
            distance[train_idx] = np.linalg.norm(test_image - train_image)

        min_distance = np.argsort(distance)[:k_neighbors]
        predict = np.argmax(np.bincount(train_labels[min_distance]))
        if predict != test_labels[test_idx]:
            error += 1
    print(f'Error count: {error}\nError rate: {float(error) / num_test}')
```

The reconstruction function selects 10 random images and reconstructs them using the provided eigenvectors. It initializes a zero matrix for the reconstructed images, computes the reconstructed images by projecting them onto the eigenvectors, and then visualizes both the original and reconstructed images side by side in a figure titled "Reconstructed faces." The decorrelate function

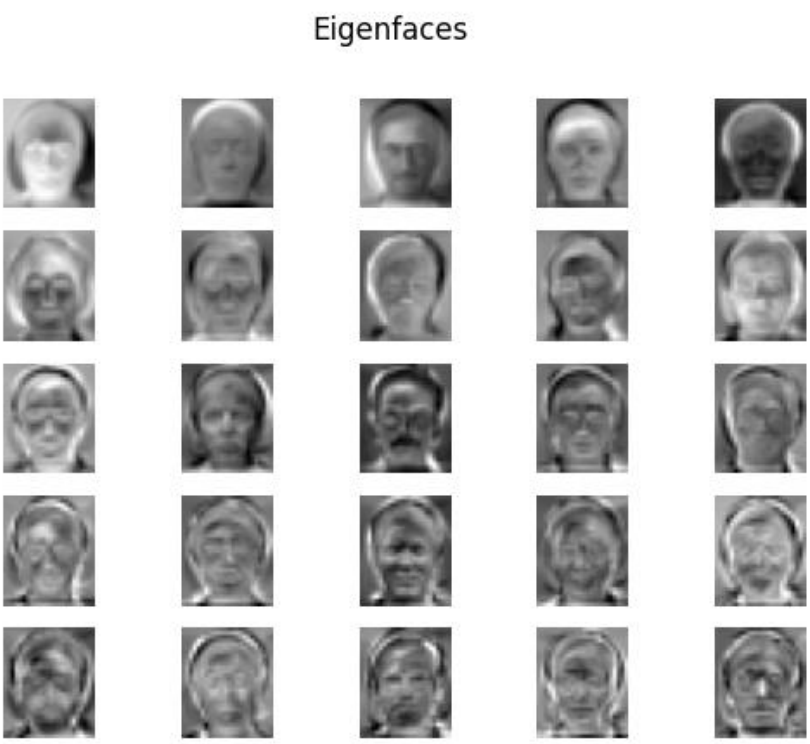
projects each image onto the eigenvector space, returning the decorrelated images. The classify function decorrelates both the training and test images, calculates the Euclidean distance between each test image and all training images, identifies the k-nearest neighbors, and predicts the label for each test image based on the majority label of its nearest neighbors. It counts the classification errors and prints the error count and error rate.

Load images and labels

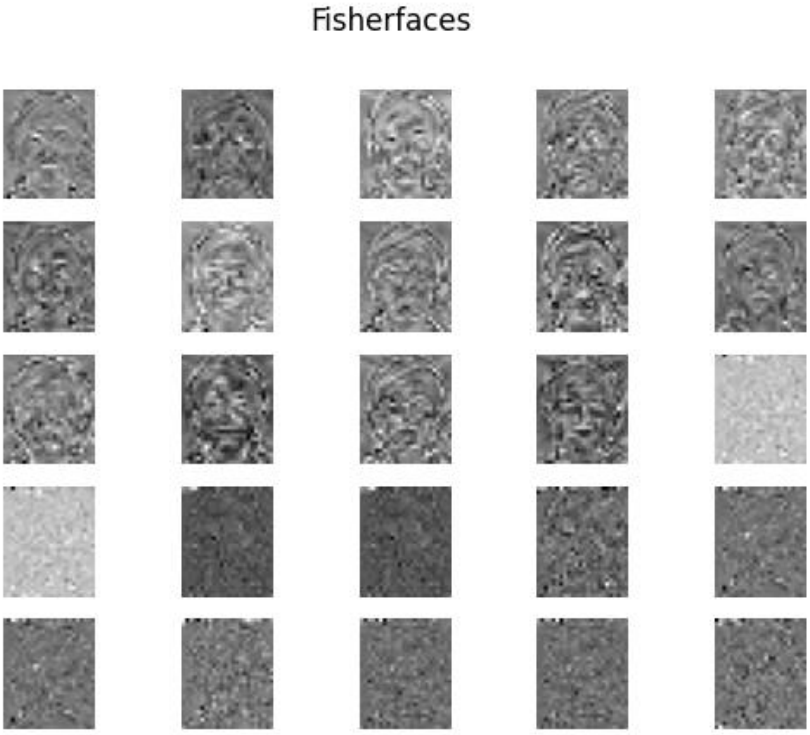
```python
def read_images_from_directory(directory_path, image_size=(24, 29)):
    """
    讀取指定目錄中的圖像和標籤。

    參數：
    directory_path (str): 圖像目錄的路徑。
    image_size (tuple): 圖像大小，預設為 (24, 29)。

    返回：
    images (numpy.ndarray): 讀取的圖像數據。
    labels (numpy.ndarray): 對應的標籤數據。
    """
    n_files = 0
    with os.scandir(directory_path) as directory:
        n_files = len([file for file in directory if file.is_file()])

    images = np.zeros((n_files, image_size[0] * image_size[1]))
    labels = np.zeros(n_files, dtype=int)

    with os.scandir(directory_path) as directory:
        for idx, file in enumerate(directory):
            if file.path.endswith('.pgm') and file.is_file():
                face = np.asarray(Image.open(file.path).resize(image_size)).reshape(1, -1)
                images[idx, :] = face
                labels[idx] = int(file.name[7:9])

    # images: 二維numpy數組，形狀為(n_files, images[0]*images[1])，每一列儲存flatten後的圖像數據
    # labels: 一為numpy數組，長度為n_files，對應的labels
    return images, labels
```

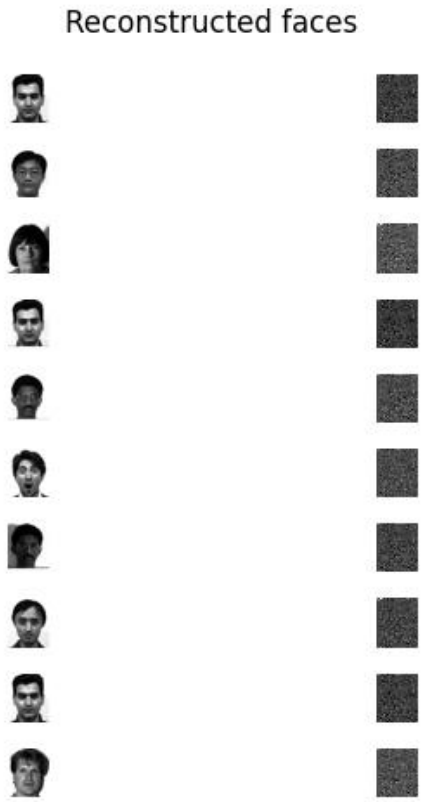
The read_images_from_directory function reads images and labels from a directory. It first counts the files to initialize the images and labels arrays. For each .pgm file in the directory, it opens, resizes, and flattens the image, storing it in the images array. The label is extracted from the file name and stored in the labels array. The function returns the images array, with each row representing a flattened image, and the labels array containing the corresponding labels.

Simple PCA: (Error count=6, Error rate=0.2)

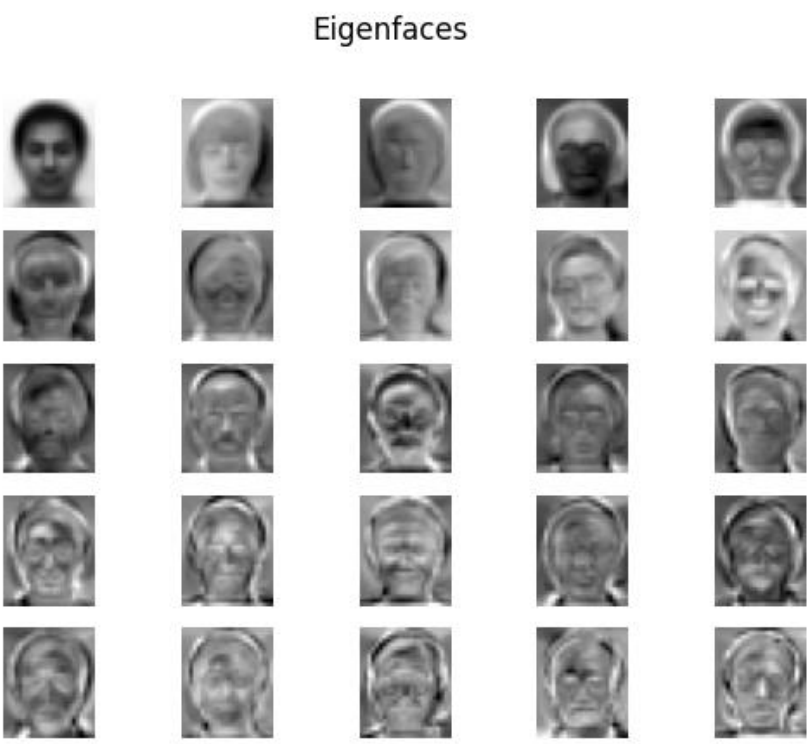


Simple LDA: (Error count=2, Error rate=0.067)

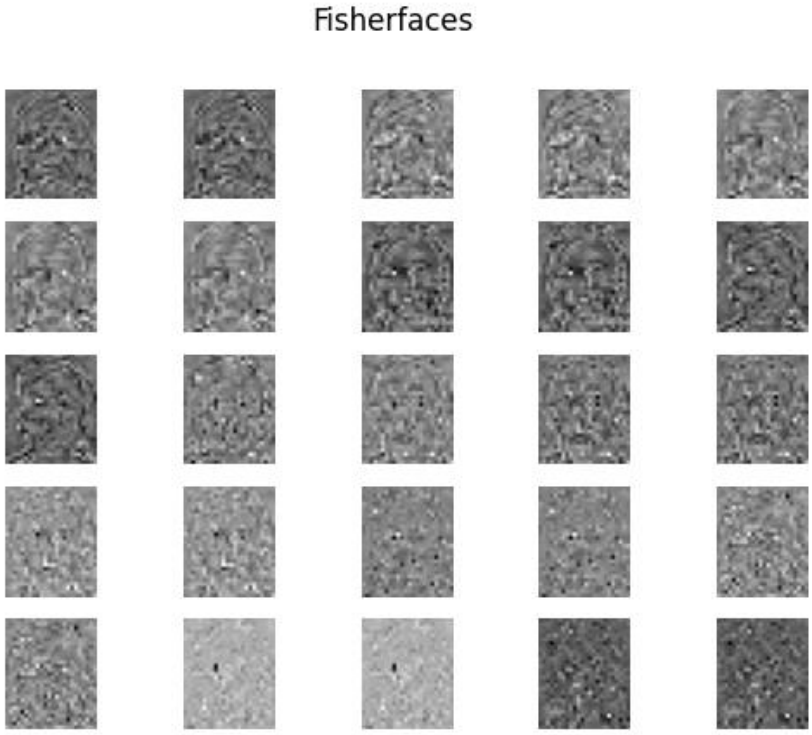


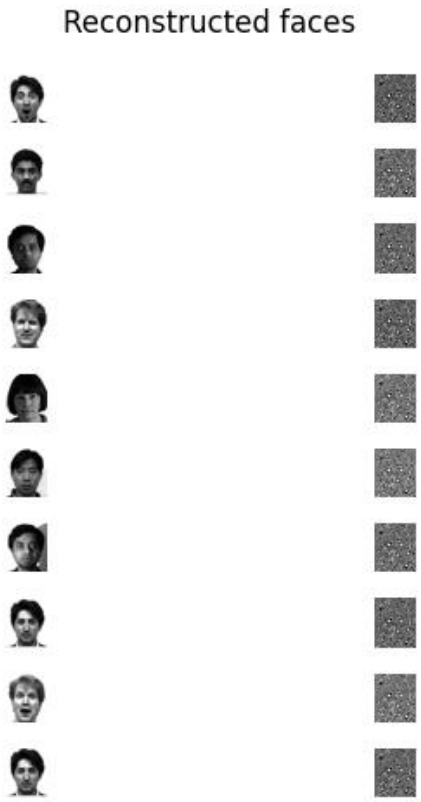


Kernel PCA (Linear): (Error count=2, Error rate=0.067)

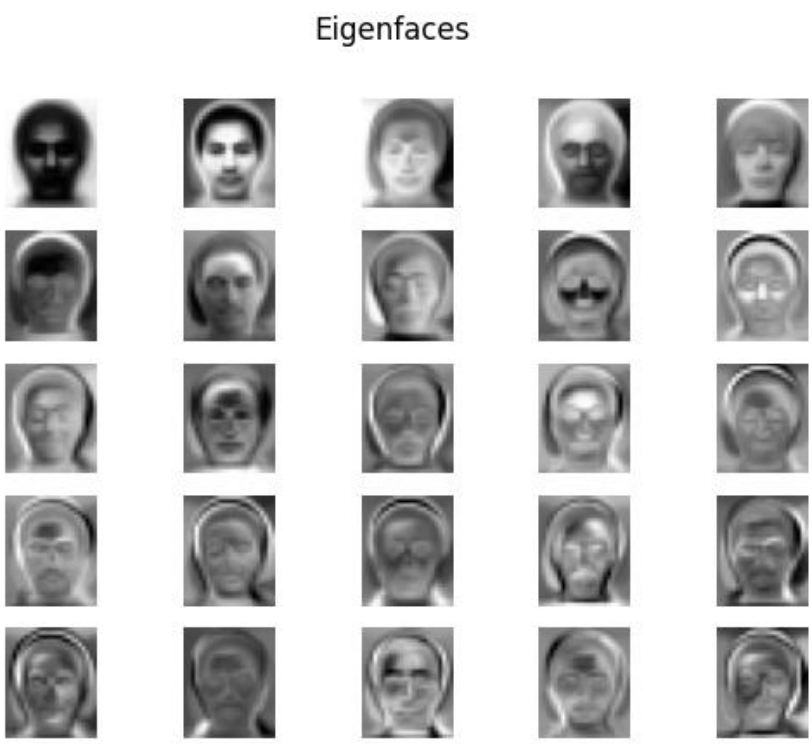


Kernel LDA (Linear): (Error count=27, Error rate=0.9)

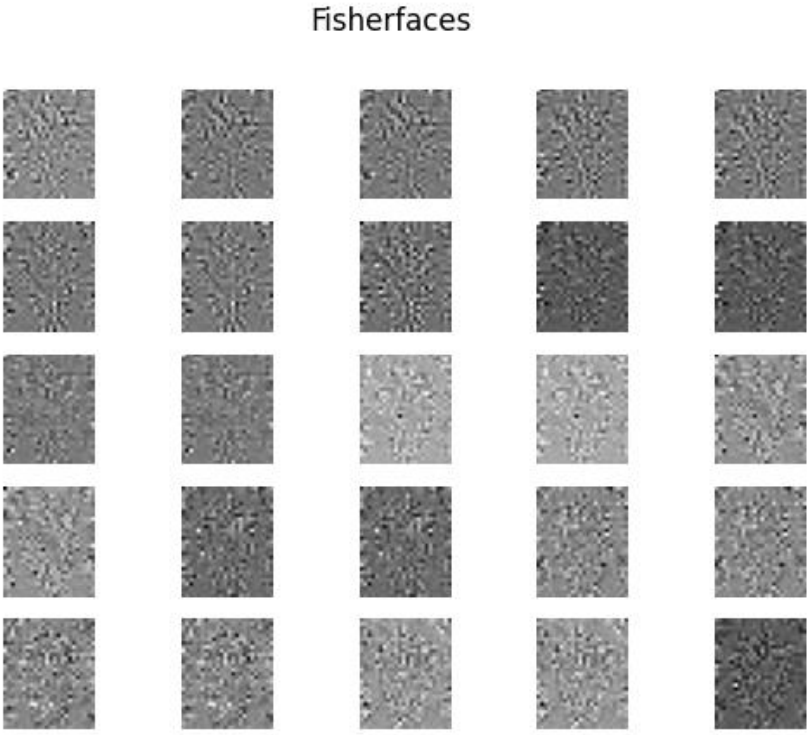


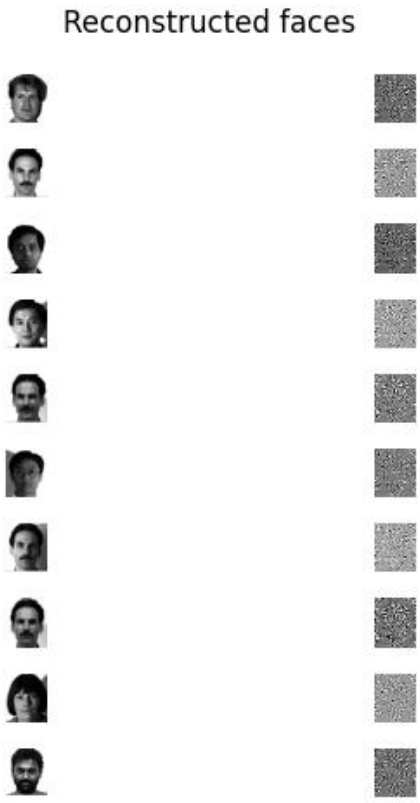


Kernel PCA (RBF): (Error count=2, Error rate=0.667)



Kernel LDA (RBF): (Error count=16, Error rate=0.5333)





Discussion:

Variance vs. Class Separation: PCA is more optimal for capturing the most significant variance in the data, making it effective for general feature extraction and noise reduction. In contrast, LDA is tailored for class separation, making it more suitable for supervised learning tasks where the goal is to distinguish between known categories.

Reconstruction Quality: PCA reconstructions tend to be more visually accurate but less class-specific, while LDA reconstructions may lose some general details but excel at preserving discriminative features.

Classification Performance: LDA generally provides better classification results than PCA for tasks involving labeled data due to its focus on maximizing between-class separability.

tsne:

Part 1

```
# Store images of clustering state
img = []

# Run iterations
for iter in range(max_iter):

    # Compute pairwise affinities
    sum_Y = np.sum(np.square(Y), 1)
    num = -2. * np.dot(Y, Y.T)

    if mode == 'tSNE':
        num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
    elif mode == 'sSNE':
        num = np.exp(-1 * np.add(np.add(num, sum_Y).T, sum_Y))
    else:
        raise BaseException(f'Invalid mode. The mode should be tSNE or sSNE(symmetricSNE)')

    num[range(n), range(n)] = 0.
    Q = num / np.sum(num)
    Q = np.maximum(Q, 1e-12)

    # Compute gradient
    PQ = P - Q
    # Choose tSNE or sSNE by adjusting mode
    if mode == 'tSNE':
        for i in range(n):
            dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
    elif mode == 'sSNE':
        for i in range(n):
            dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
    else:
        raise BaseException(f'Invalid mode. The mode should be tSNE or sSNE(symmetricSNE)')
```

The modified code implements both t-SNE and symmetric SNE (s-SNE) for dimensionality reduction and visualization. In t-SNE, the pairwise affinities are computed using a heavy-tailed Student's t-distribution, which helps prevent the crowding problem by allowing distant points to influence the embedding. The gradient is computed based on these affinities to optimize the low-dimensional representation. For s-SNE, the pairwise affinities are computed using a Gaussian distribution, leading to a symmetric similarity measure. The gradient is then computed using this Gaussian-based affinity, which may result in a more crowded embedding since distant points have less influence compared to t-SNE.

In the code, the sne function handles both t-SNE and s-SNE modes by switching the similarity measure and gradient computation based on the mode parameter. The x2p function calculates the pairwise affinities for high-dimensional data, while the curr_state function captures the current state of the embedding for visualization. The

observations show that while t-SNE effectively spreads out the data points, reducing the crowding problem, s-SNE might produce more crowded visualizations due to the symmetric Gaussian similarity measure. Despite this, s-SNE can be useful for certain datasets where Gaussian similarities are preferred for capturing the local structure.

Capture the current state

```
def curr_state(Y, labels, mode, perplexity):

    plt.clf()  # clear the current img
    plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
    plt.title(f'{"t-SNE" if mode == "tSNE" else "symmetric-SNE"}, perplexity = {perplexity}')
    plt.tight_layout()
    canvas = plt.get_current_fig_manager().canvas
    canvas.draw()

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

    # plt.clf()  # clear the current img
    # plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
    # plt.title(f'{"t-SNE" if mode == "tSNE" else "symmetric-SNE"}, perplexity = {perplexity}')
    # plt.tight_layout()
    # canvas = plt.get_current_fig_manager().canvas
    # canvas.draw()
    # width, height = canvas.get_width_height()
    # # img_data = canvas.buffer_rgba()
    # img_data = np.frombuffer(canvas.buffer_rgba(), dtype=np.uint8).reshape(height, width, 4)

    # return Image.frombuffer('RGBA', (width, height), img_data, 'raw', 'RGBA', 0, 1)
```

The curr_state function captures the current state of the embedding during the t-SNE or symmetric SNE (s-SNE) algorithm's iterations and converts it into an image. It clears the current plot, creates a scatter plot of the low-dimensional data points colored by their labels, sets the plot title based on the mode (t-SNE or s-SNE) and perplexity, and draws the canvas. The function then returns an image created from the canvas content, which can be used to visualize the embedding's evolution over time. This function helps in generating a visual record of how the data points are arranged in the reduced dimensional space throughout the algorithm's execution.

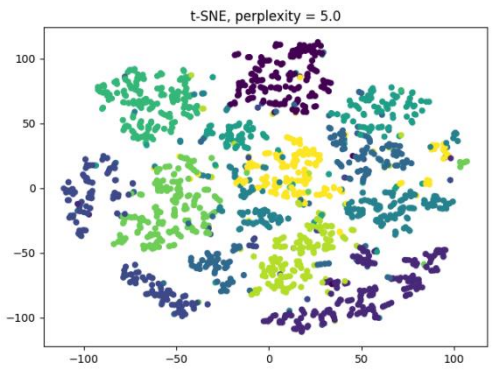
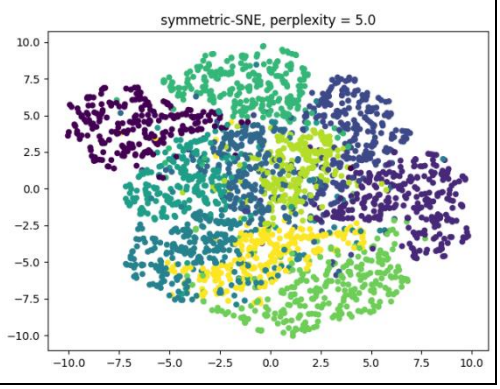
Similarity compute

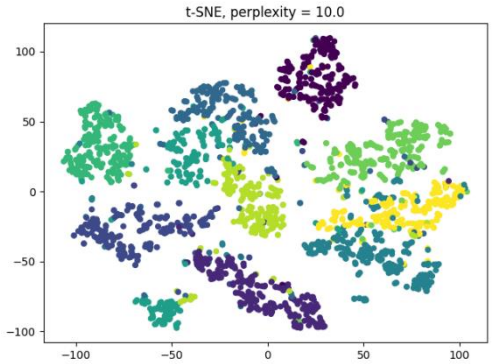
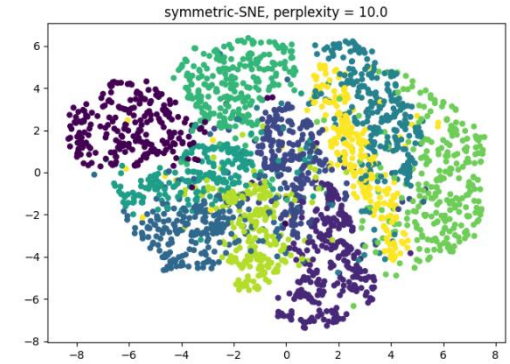
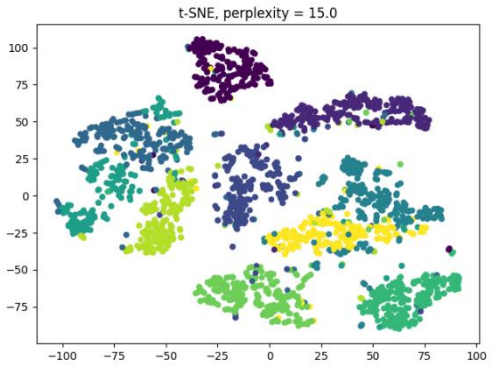
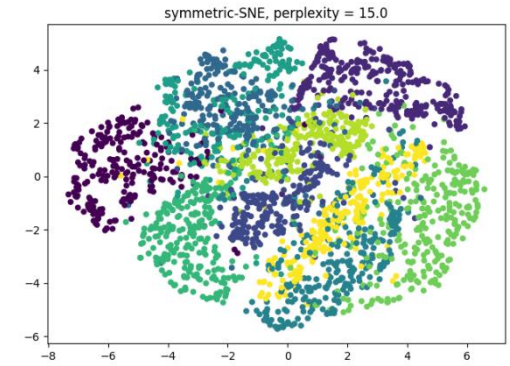
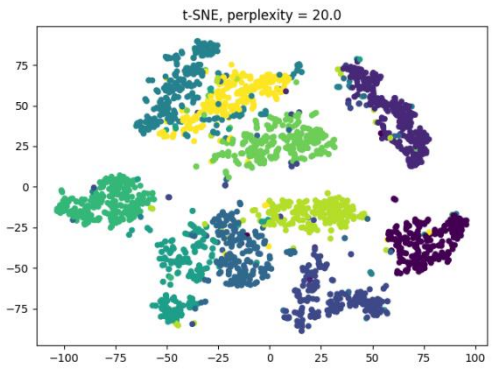
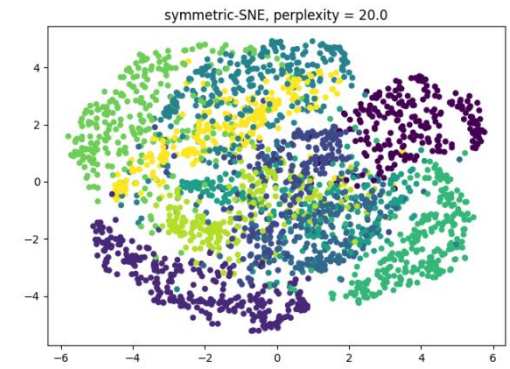
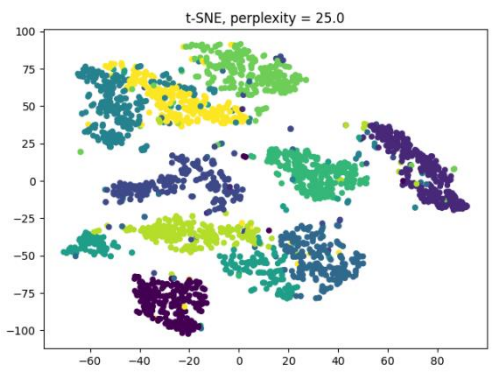
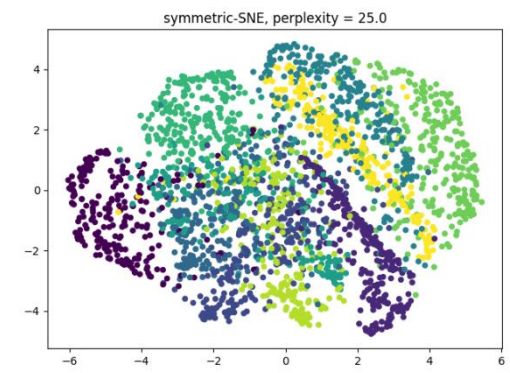
```python
def similarity(P, Q, labels, output_dir):
    index = np.argsort(labels)
    plt.clf()
    plt.figure(1)

    # plot P
    logP = np.log(P)
    sorted_P = logP[index][:, index]
    plt.subplot(121)
    img = plt.imshow(sorted_P, cmap='gray', vmin=np.min(logP), vmax=np.max(logP))
    plt.colorbar(img)
    plt.title(f'High dimensional space')

    # plot Q
    logQ = np.log(Q)
    sorted_Q = logQ[index][:, index]
    plt.subplot(122)
    img = plt.imshow(sorted_Q, cmap='gray', vmin=np.min(logQ), vmax=np.max(logQ))
    plt.colorbar(img)
    plt.title(f'Low dimensional space')

    plt.tight_layout()

    # Save the similarity plot
    filename = f'{output_dir}/similarity_plot.png'
    plt.savefig(filename)
    plt.close()

def parse_arguments():
    parser = ArgumentParser(description='SNE')
    parser.add_argument('--mode', default='sSNE', help='tSNE, sSNE(symmetricSNE)', type=str)
    parser.add_argument('--perplexity', default=5.0, type=float)

    return parser.parse_args()
```

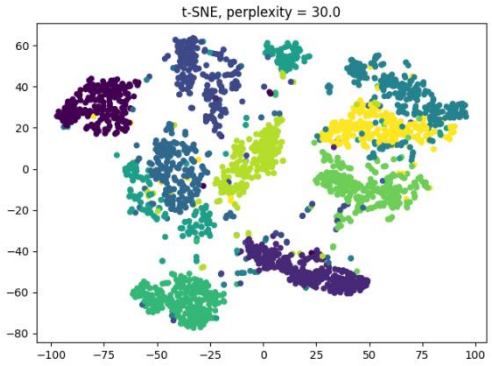
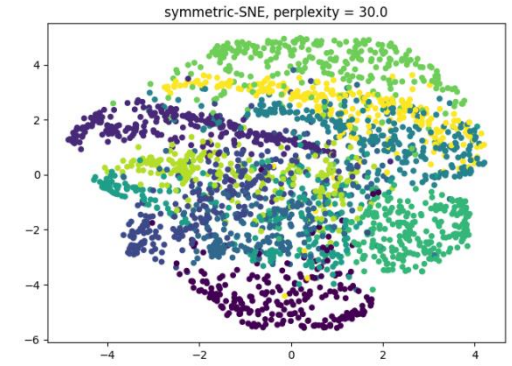
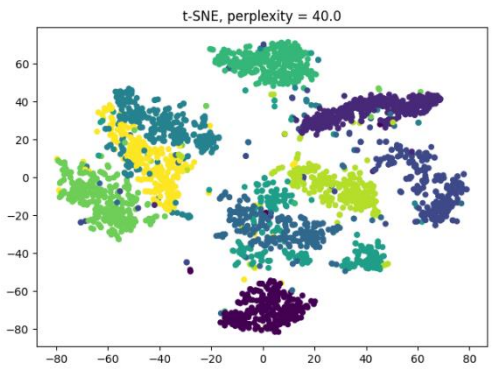
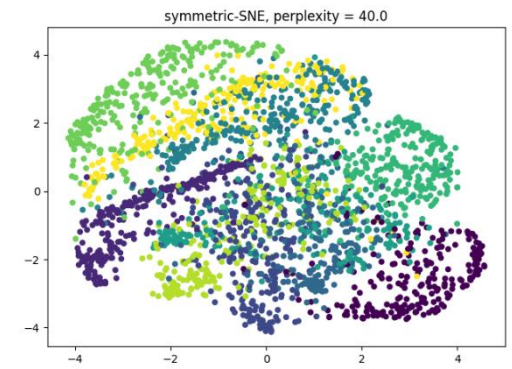
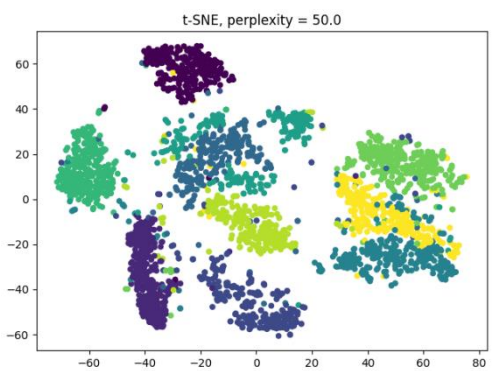
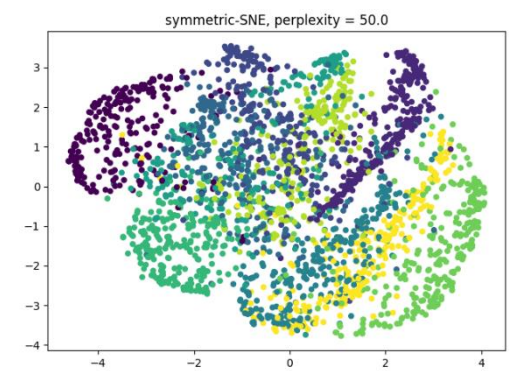
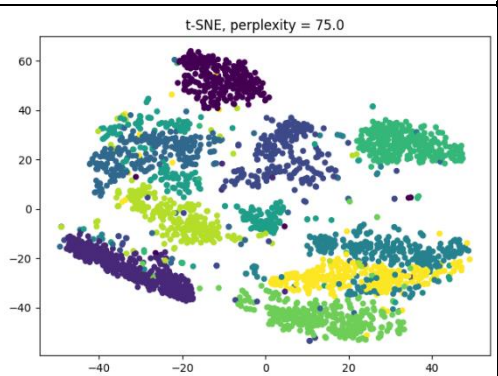
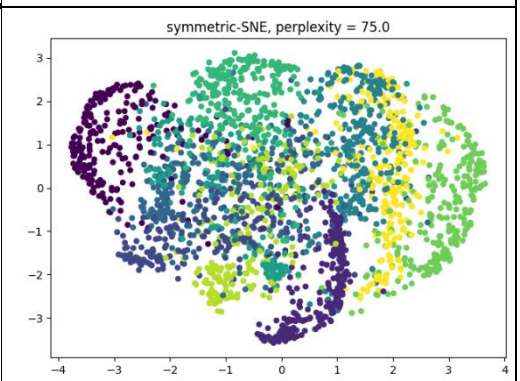
The similarity function visualizes the pairwise similarity matrices P (high-dimensional space) and Q (low-dimensional space) for t-SNE or symmetric SNE (s-SNE). It sorts the matrices based on the labels, applies a logarithmic transformation for better visualization, and creates side-by-side heatmaps of the sorted matrices. The function labels the plots accordingly and saves the combined similarity plot as an image in the specified output directory. This visualization helps to compare and understand how well the low-dimensional representation (Q) preserves the pairwise similarities from the high-dimensional data (P).

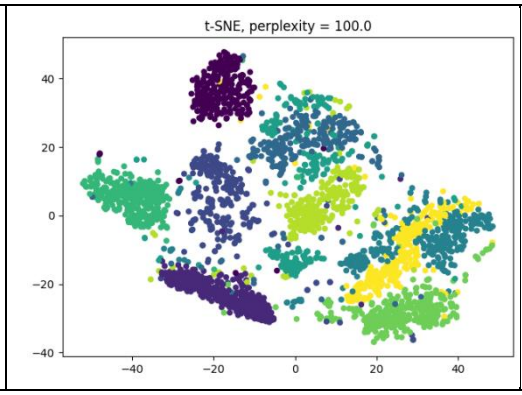
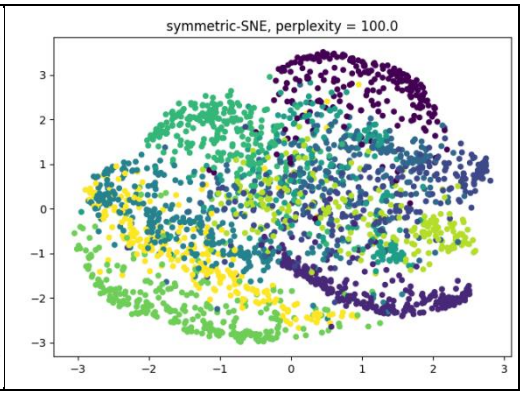
```python
if __name__ == "__main__":
    print("Run Y = tsne.tsne(X, no_dims, perplexity) to perform t-SNE on your dataset.")
    print("Running example on 2,500 MNIST digits...")

    args = parse_arguments()
    mode = args.mode
    perplexity = args.perplexity

    output_dir = f'./{"t-SNE" if mode == "tSNE" else "symmetric-SNE"}_{perplexity}'
    os.makedirs(output_dir, exist_ok=True)

    # load data
    X = np.loadtxt("./tsne_python/tsne_python/mnist2500_X.txt")
    labels = np.loadtxt("./tsne_python/tsne_python/mnist2500_labels.txt")

    # tSNE or sSNE
    Y = sne(X, labels, mode, 2, 50, perplexity)

    # plot
    plt.figure(2)
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    plt.title(f'{"t-SNE" if mode == "tSNE" else "symmetric-SNE"}, perplexity = {perplexity}')
    plt.tight_layout()
    filename = f'{output_dir}/{"t-SNE" if mode == "tSNE" else "symmetric-SNE"}_{perplexity}.png'
    plt.savefig(filename)
    pylab.show()
```

Part 2

	tSNE	sSNE
5		

10	 t-SNE, perplexity = 10.0	 symmetric-SNE, perplexity = 10.0
15	 t-SNE, perplexity = 15.0	 symmetric-SNE, perplexity = 15.0
20	 t-SNE, perplexity = 20.0	 symmetric-SNE, perplexity = 20.0
25	 t-SNE, perplexity = 25.0	 symmetric-SNE, perplexity = 25.0

30	 t-SNE, perplexity = 30.0	 symmetric-SNE, perplexity = 30.0
40	 t-SNE, perplexity = 40.0	 symmetric-SNE, perplexity = 40.0
50	 t-SNE, perplexity = 50.0	 symmetric-SNE, perplexity = 50.0
75	 t-SNE, perplexity = 75.0	 symmetric-SNE, perplexity = 75.0

100		

Discussion:

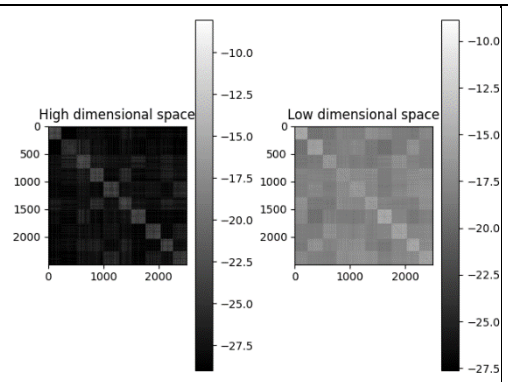
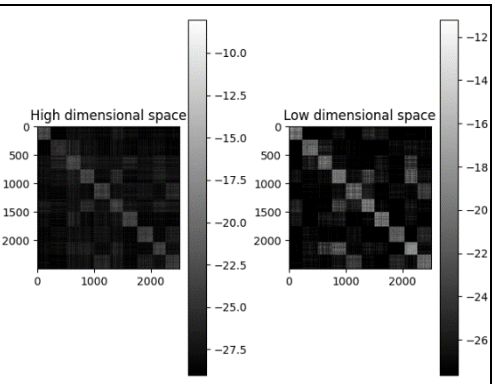
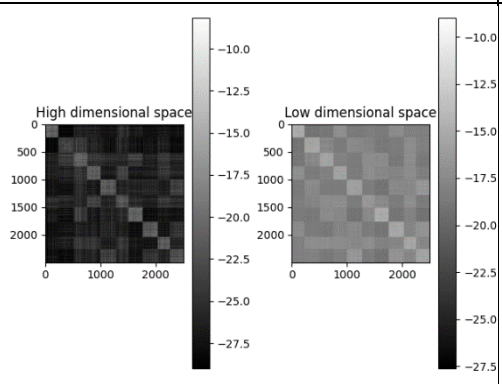
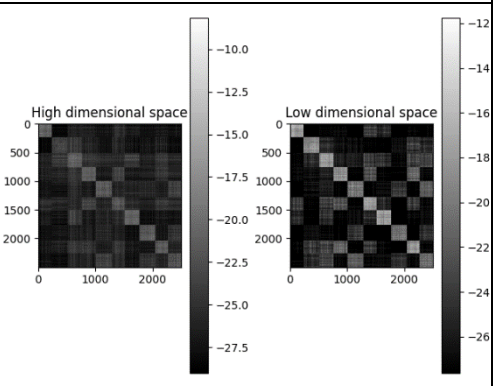
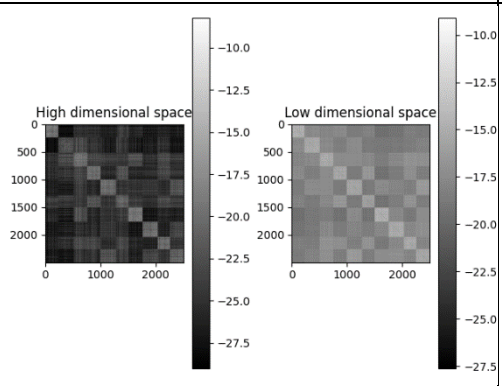
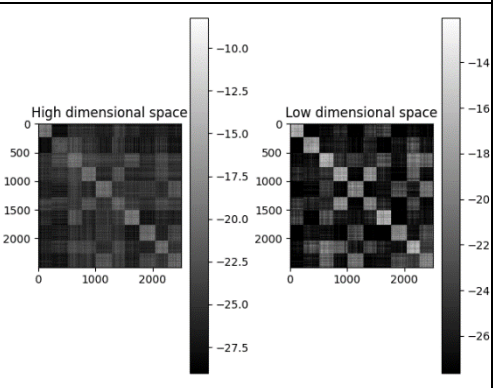
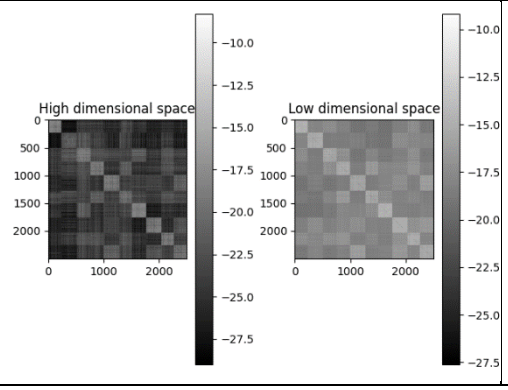
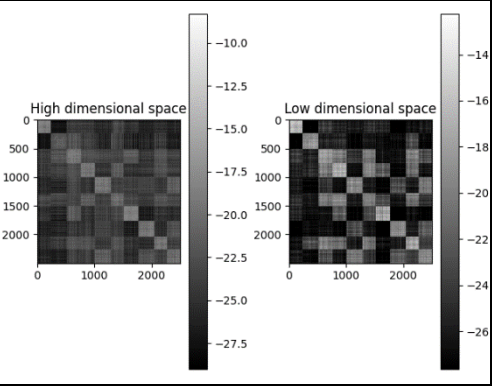
The process involves loading the dataset and preprocessing it using PCA to reduce the dimensionality to 50. Then, t-SNE and symmetric SNE (s-SNE) are applied to project the data onto a 2D space. During this projection, the embedding state is captured at each iteration and saved as a GIF to visualize the optimization process. Different colors are used to mark data points based on their labels, making it easy to distinguish between classes. Various perplexity values (e.g., 5, ..., 50, 75 and 100) are experimented with to observe changes in the visualization.

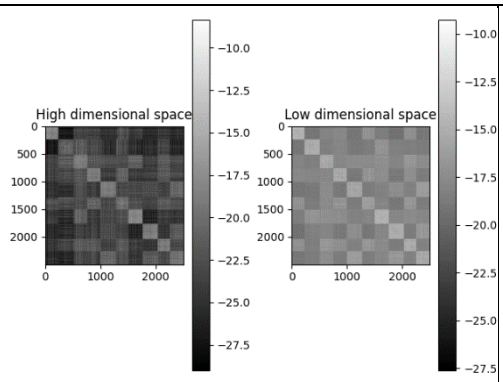
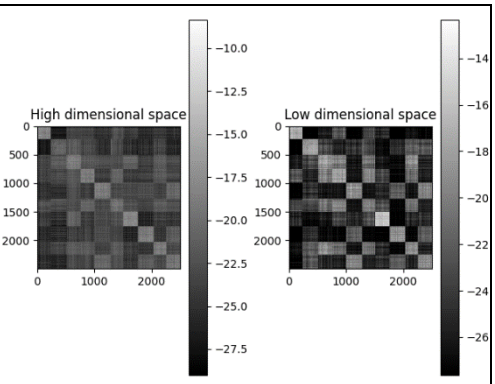
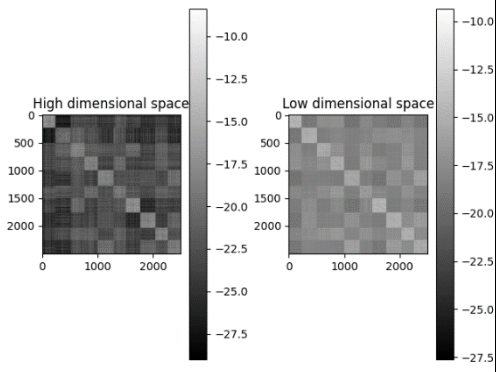
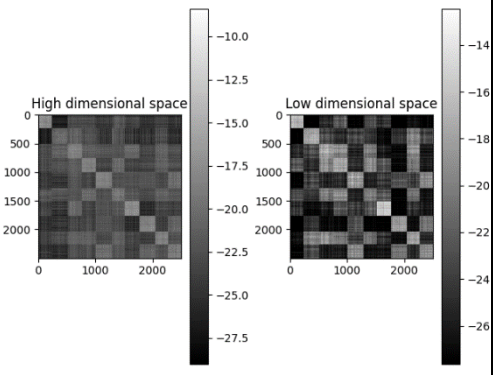
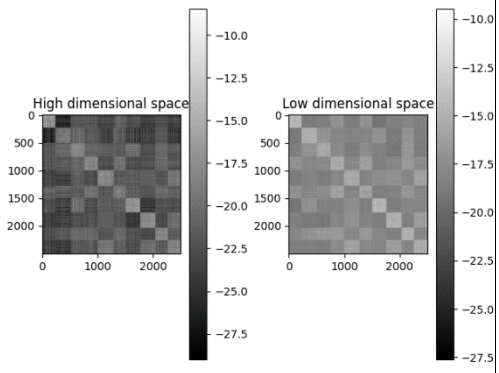
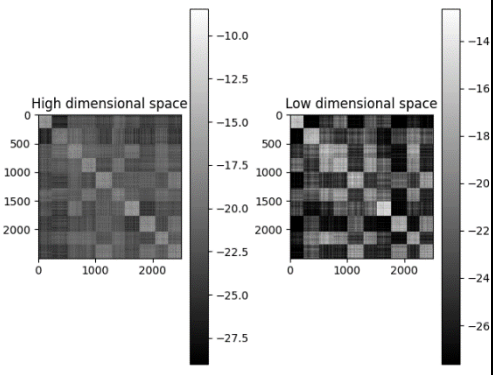
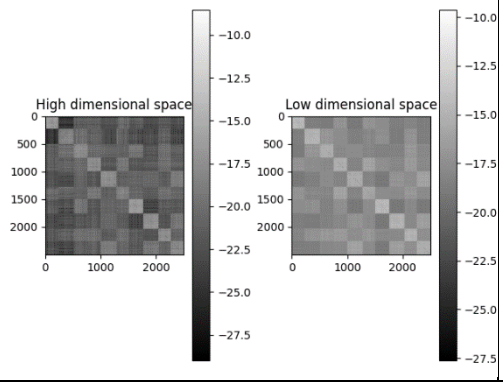
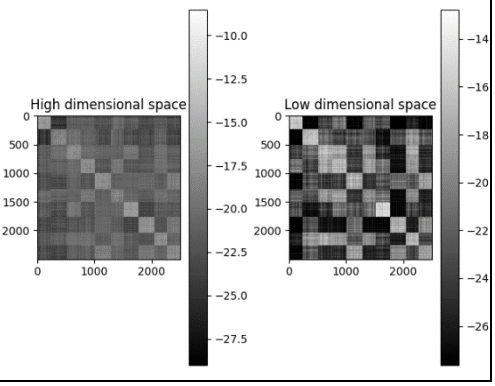
Part 3

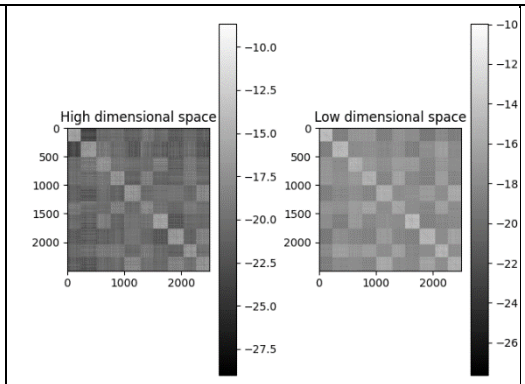
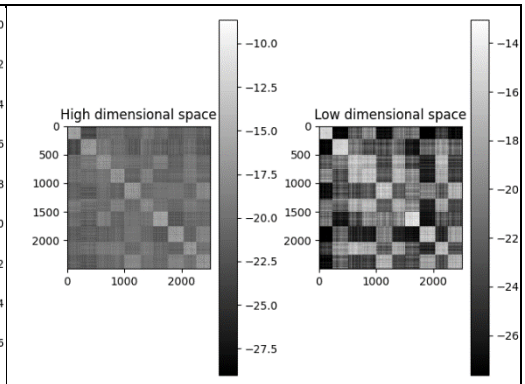
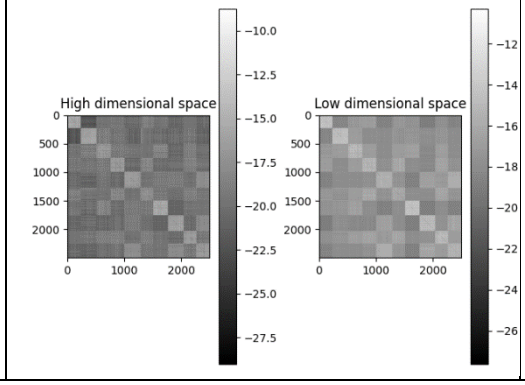
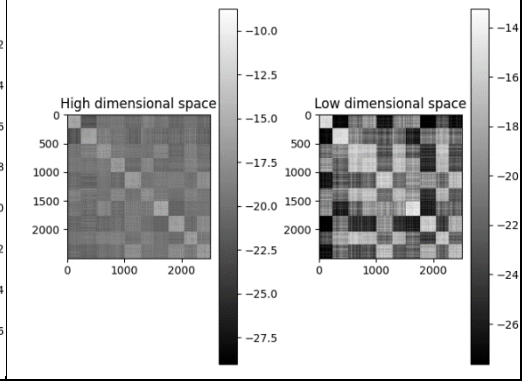
(.gif files were stored in the output directory)

Part 4

	tSNE	sSNE

5	 High dimensional space Low dimensional space	 High dimensional space Low dimensional space
10	 High dimensional space Low dimensional space	 High dimensional space Low dimensional space
15	 High dimensional space Low dimensional space	 High dimensional space Low dimensional space
20	 High dimensional space Low dimensional space	 High dimensional space Low dimensional space

25	 High dimensional space Low dimensional space	 High dimensional space Low dimensional space
30	 High dimensional space Low dimensional space	 High dimensional space Low dimensional space
40	 High dimensional space Low dimensional space	 High dimensional space Low dimensional space
50	 High dimensional space Low dimensional space	 High dimensional space Low dimensional space

75		
100		

Discussion:

t-SNE effectively spreads out data points, reducing the crowding problem by using a heavy-tailed Student's t-distribution for pairwise affinities, allowing distant points to influence the embedding more. This results in more distinct clusters in the 2D space.

On the other hand, symmetric SNE (s-SNE) can lead to more crowded visualizations as distant points have less influence, resulting in overlapping clusters.

Varying the perplexity affects the balance between local and global aspects of the data: lower perplexity values focus on local structures, while higher values capture broader relationships, influencing the overall spread and separation of clusters in the visualizations.