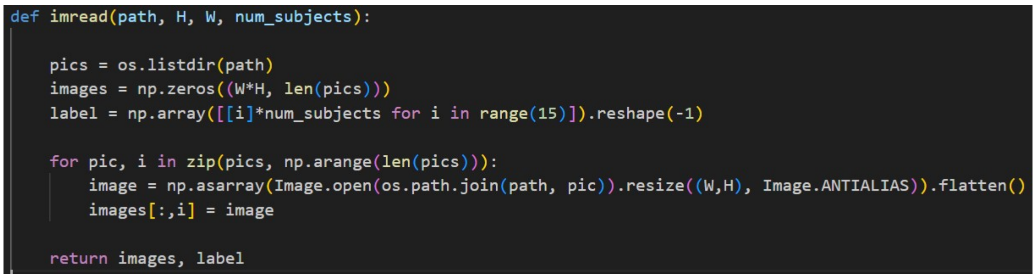
Machine Learning
HW7-Kernel Eigenfaces & t-SNE

Yu-Ju Tsai, 311652010

June 14, 2023

1 Kernel Eigenfaces

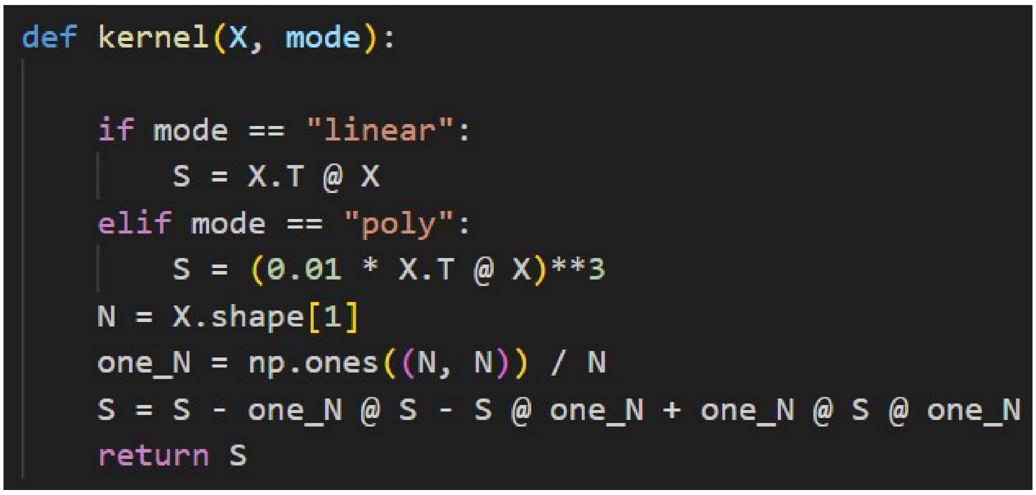
- Load data:
  Due to the presence of 135 files (15×9), the size of the training images is 45045x135, where each column represents an individual files, similarly, the size of the training labels is 135, corresponding to the labels for each training image, on the other hand, the test dataset consists of 30 files (15×2), resulting in the test images having a size of 45045×30, with each column representing an individual test files, likewise, the size of the test labels is 30.



```python
def imread(path, H, W, num_subjects):

    pics = os.listdir(path)
    images = np.zeros((W*H, len(pics)))
    label = np.array([[i]*num_subjects for i in range(15)]).reshape(-1)

    for pic, i in zip(pics, np.arange(len(pics))):
        image = np.asarray(Image.open(os.path.join(path, pic)).resize((W,H), Image.ANTIALIAS)).flatten()
        images[:,i] = image

    return images, label
```

Figure 1: Load Data

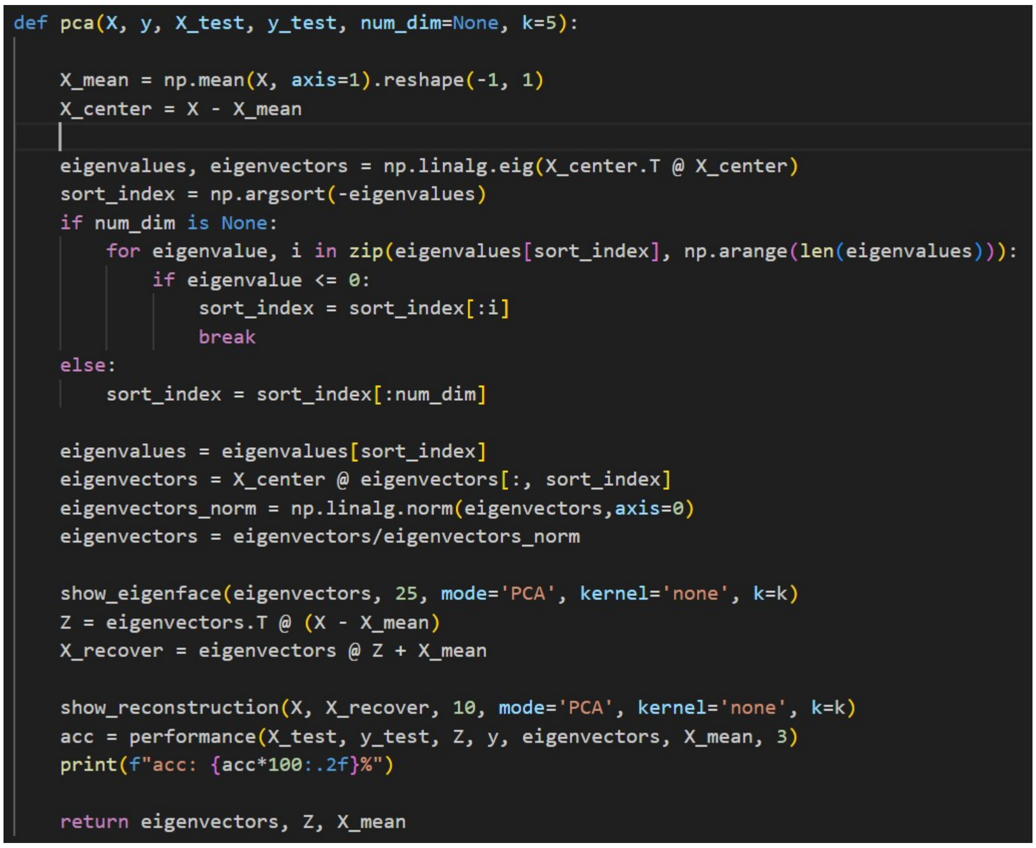
- Kernel:
  X is image, the kernel is calculated before computing the eigenvalues.



```python
def kernel(X, mode):

    if mode == "linear":
        S = X.T @ X
    elif mode == "poly":
        S = (0.01 * X.T @ X)**3
    N = X.shape[1]
    one_N = np.ones((N, N)) / N
    S = S - one_N @ S - S @ one_N + one_N @ S @ one_N
    return S
```

Figure 2: Kernel

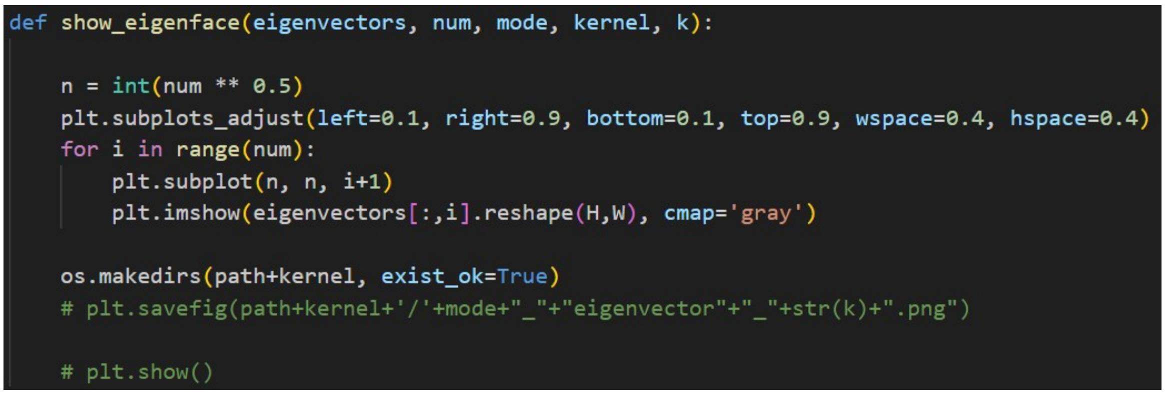
- PCA:
  Due to the large size of X*X.T, computing its eigen decomposition can be computationally expensive. Therefore, we calculate the eigenvalues and eigenvectors of X.T*X instead. Subsequently, we multiply the corresponding eigenvectors (excluding those associated with eigenvalues smaller than 0) with X to represent a subset of eigenvectors for X*X.T. By utilizing the columns of the eigenvectors, we can obtain eigenfaces. During the recovery process, we project X onto a lower dimension using the transose of eigenvector matrix, and subsequently project it back to the original dimension using the eigenvector.



```python
def pca(X, y, X_test, y_test, num_dim=None, k=5):

    X_mean = np.mean(X, axis=1).reshape(-1, 1)
    X_center = X - X_mean

    eigenvalues, eigenvectors = np.linalg.eig(X_center.T @ X_center)
    sort_index = np.argsort(-eigenvalues)
    if num_dim is None:
        for eigenvalue, i in zip(eigenvalues[sort_index], np.arange(len(eigenvalues))):
            if eigenvalue <= 0:
                sort_index = sort_index[:i]
                break
    else:
        sort_index = sort_index[:num_dim]

    eigenvalues = eigenvalues[sort_index]
    eigenvectors = X_center @ eigenvectors[:, sort_index]
    eigenvectors_norm = np.linalg.norm(eigenvectors,axis=0)
    eigenvectors = eigenvectors/eigenvectors_norm

    show_eigenface(eigenvectors, 25, mode='PCA', kernel='none', k=k)
    Z = eigenvectors.T @ (X - X_mean)
    X_recover = eigenvectors @ Z + X_mean

    show_reconstruction(X, X_recover, 10, mode='PCA', kernel='none', k=k)
    acc = performance(X_test, y_test, Z, y, eigenvectors, X_mean, 3)
    print(f"acc: {acc*100:.2f}%")

    return eigenvectors, Z, X_mean
```

Figure 3: PCA

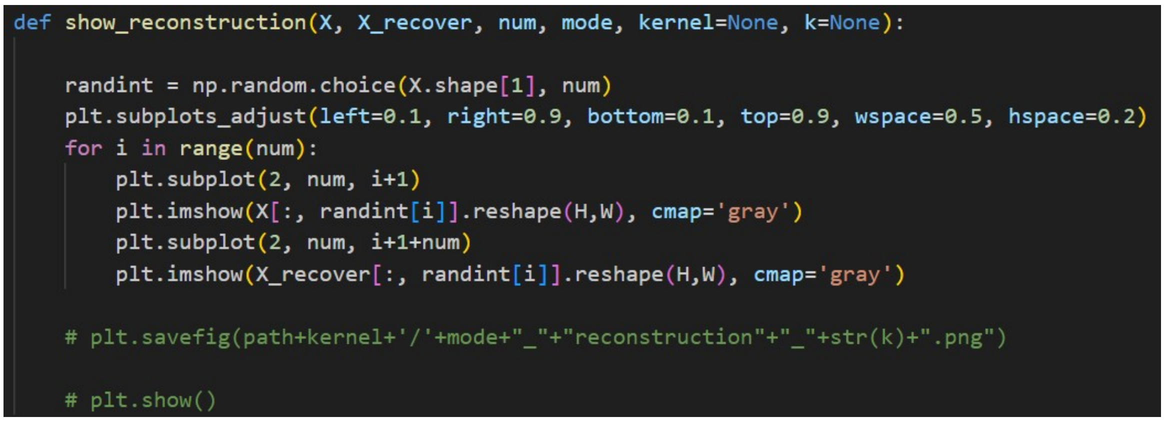
- Show eigenface:
  num=25, and mode='PCA, Kernel PCA, LDA, Kernel lda', each column of matrix X represents one image and the results will be saved in the specified path.



```python
def show_eigenface(eigenvectors, num, mode, kernel, k):

    n = int(num ** 0.5)
    plt.subplots_adjust(left=0.1, right=0.9, bottom=0.1, top=0.9, wspace=0.4, hspace=0.4)
    for i in range(num):
        plt.subplot(n, n, i+1)
        plt.imshow(eigenvectors[:,i].reshape(H,W), cmap='gray')

    os.makedirs(path+kernel, exist_ok=True)
    # plt.savefig(path+kernel+'/'+mode+"_"+"eigenvector"+"_"+str(k)+".png")

    # plt.show()
```

Figure 4: show eigenface

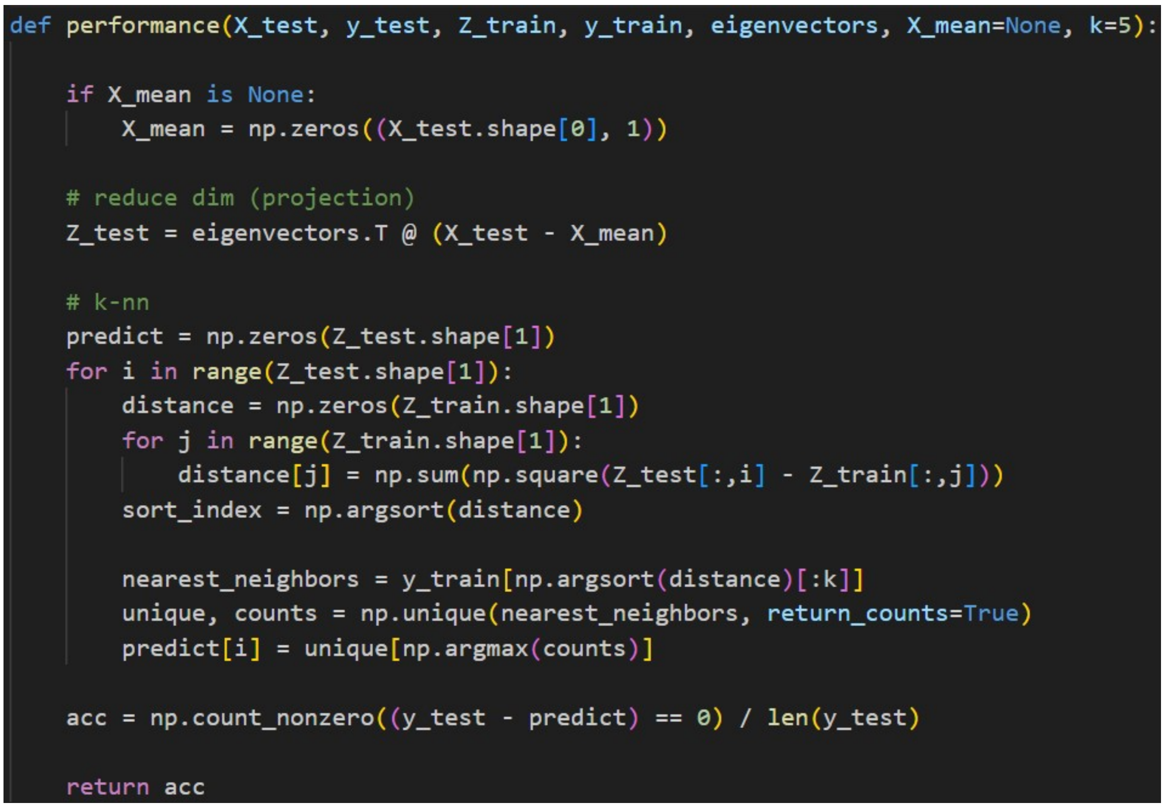
- Show reconstruction:
  X represents the original images, while X recover denotes the images that have undergone Principal Component Analysis (PCA) and have been subsequently remapped to the original image space. The remaining details remain unchanged from the aforementioned description.



```python
def show_reconstruction(X, X_recover, num, mode, kernel=None, k=None):

    randint = np.random.choice(X.shape[1], num)
    plt.subplots_adjust(left=0.1, right=0.9, bottom=0.1, top=0.9, wspace=0.5, hspace=0.2)
    for i in range(num):
        plt.subplot(2, num, i+1)
        plt.imshow(X[:, randint[i]].reshape(H,W), cmap='gray')
        plt.subplot(2, num, i+1+num)
        plt.imshow(X_recover[:, randint[i]].reshape(H,W), cmap='gray')

    # plt.savefig(path+kernel+'/'+mode+"_"+"reconstruction"+"_"+str(k)+".png")

    # plt.show()
```

Figure 5: show reconstruction

- Performance:
  First, the test data is mapped to a lower dimension using eigenvectors. Then, the distance between each column of the mapped test data and z train is calculated and stored in the ”distance” matrix. Next, the K-Nearest Neighbors (KNN) algorithm is utilized to determine the predictions. Finally, by comparing the predictions with the test labels, the accuracy can be obtained.

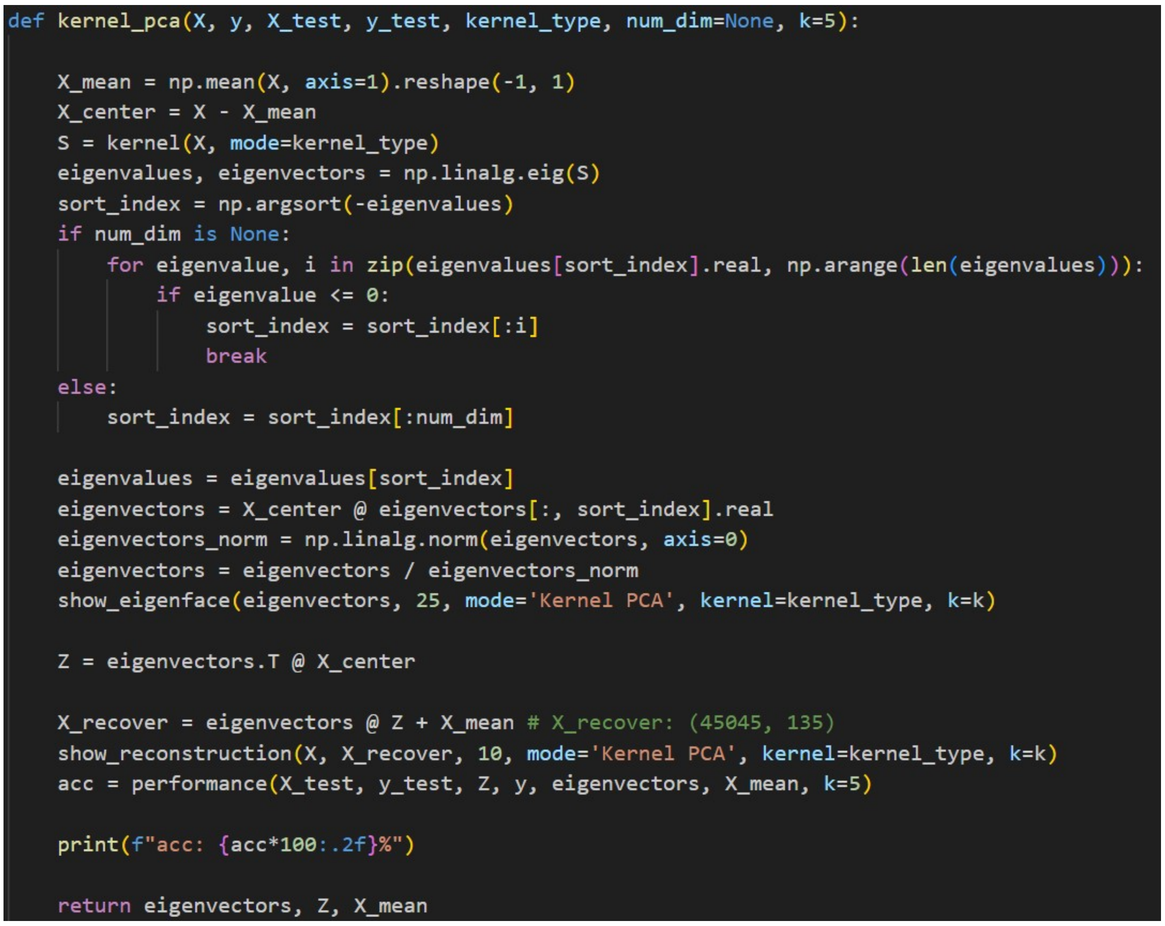


```python
def performance(X_test, y_test, Z_train, y_train, eigenvectors, X_mean=None, k=5):

    if X_mean is None:
        X_mean = np.zeros((X_test.shape[0], 1))

    # reduce dim (projection)
    Z_test = eigenvectors.T @ (X_test - X_mean)

    # k-nn
    predict = np.zeros(Z_test.shape[1])
    for i in range(Z_test.shape[1]):
        distance = np.zeros(Z_train.shape[1])
        for j in range(Z_train.shape[1]):
            distance[j] = np.sum(np.square(Z_test[:,i] - Z_train[:,j]))
        sort_index = np.argsort(distance)

        nearest_neighbors = y_train[np.argsort(distance)[:k]]
        unique, counts = np.unique(nearest_neighbors, return_counts=True)
        predict[i] = unique[np.argmax(counts)]

    acc = np.count_nonzero((y_test - predict) == 0) / len(y_test)

    return acc
```

Figure 6: performance

- Kernel pca:
  The main difference between Kernel PCA and PCA lies in the preprocessing step before calculating the eigenvectors and eigenvalues. In Kernel PCA, the data is first transformed

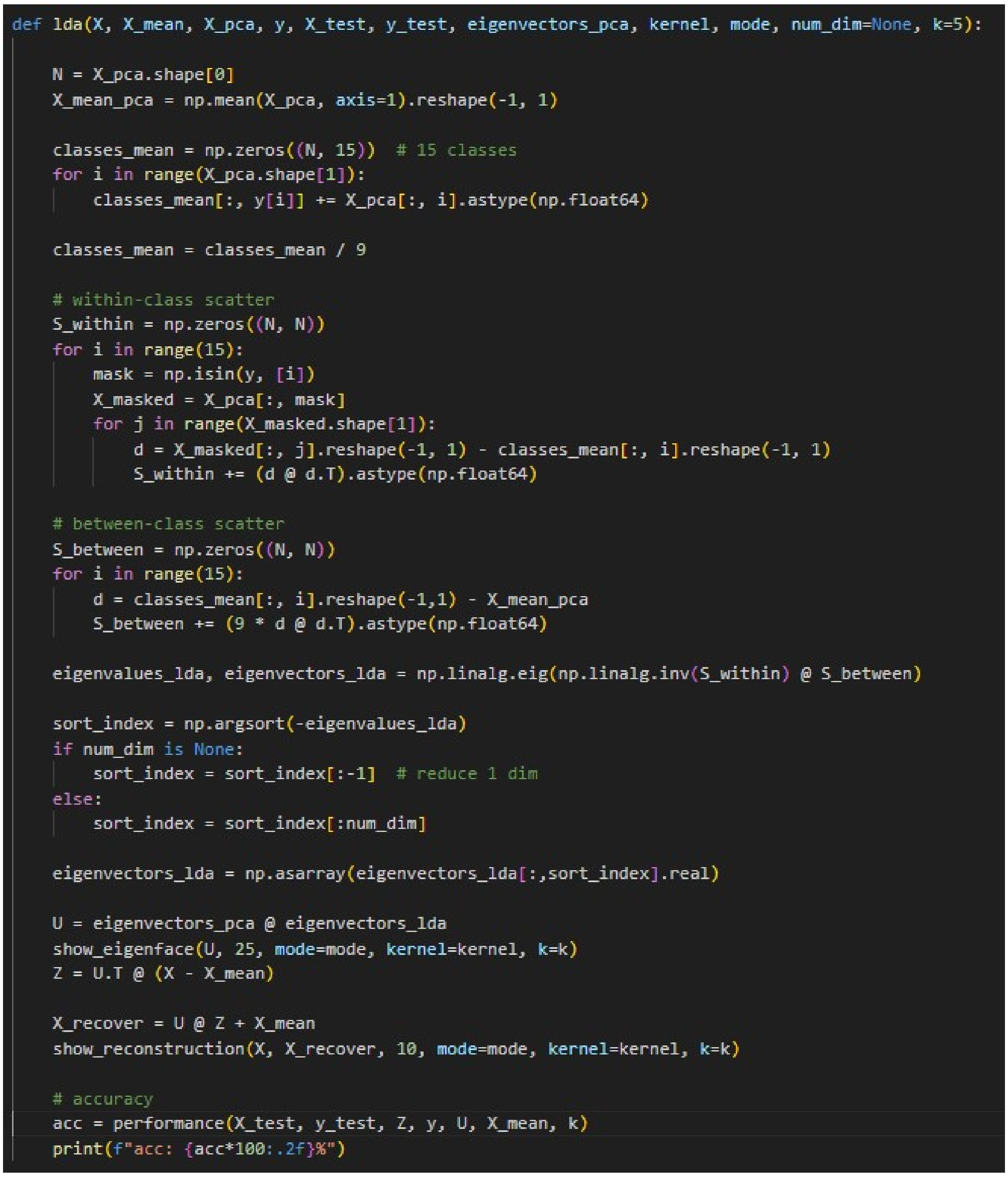
into a feature space using a kernel function, whereas in PCA, the data is directly used in its original input space, once the data is transformed into the feature space, the remaining steps in Kernel PCA are similar to PCA.



```python
def kernel_pca(X, y, X_test, y_test, kernel_type, num_dim=None, k=5):

    X_mean = np.mean(X, axis=1).reshape(-1, 1)
    X_center = X - X_mean
    S = kernel(X, mode=kernel_type)
    eigenvalues, eigenvectors = np.linalg.eig(S)
    sort_index = np.argsort(-eigenvalues)
    if num_dim is None:
        for eigenvalue, i in zip(eigenvalues[sort_index].real, np.arange(len(eigenvalues))):
            if eigenvalue <= 0:
                sort_index = sort_index[:i]
                break
    else:
        sort_index = sort_index[:num_dim]

    eigenvalues = eigenvalues[sort_index]
    eigenvectors = X_center @ eigenvectors[:, sort_index].real
    eigenvectors_norm = np.linalg.norm(eigenvectors, axis=0)
    eigenvectors = eigenvectors / eigenvectors_norm
    show_eigenface(eigenvectors, 25, mode='Kernel PCA', kernel=kernel_type, k=k)

    Z = eigenvectors.T @ X_center

    X_recover = eigenvectors @ Z + X_mean # X_recover: (45045, 135)
    show_reconstruction(X, X_recover, 10, mode='Kernel PCA', kernel=kernel_type, k=k)
    acc = performance(X_test, y_test, Z, y, eigenvectors, X_mean, k=5)

    print(f"acc: {acc*100:.2f}%")

    return eigenvectors, Z, X_mean
```

Figure 7: kernel pca

- LDA:
  The LDA function shares similarities with PCA, but they differ in the initial stage. In LDA, we aim to optimize $J(w) = \frac{w^T S_B w}{w^T S_W w}$. To achieve this, we calculate S between class and S within class first, and then define $S = S_w^{-1} S_B$. The remaining steps align with those in PCA.
  The testing phase is a common step in both PCA and LDA. It involves computing the z values for the test data and calculating the distance between the training z values and the testing z values. By finding the kth nearest neighborhood of the test data, we can determine its corresponding class. Ultimately, the accuracy of the classification is obtained.

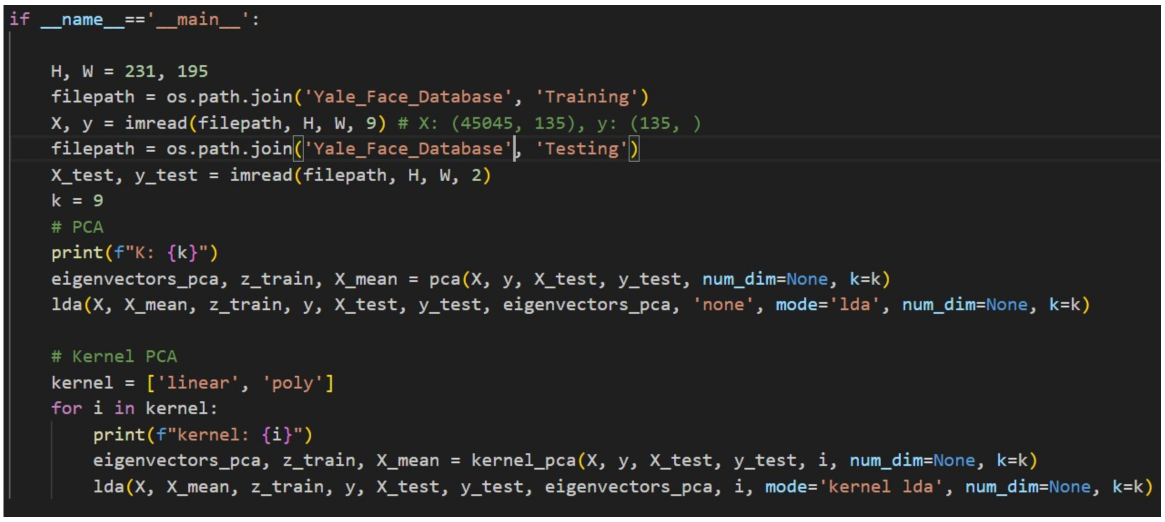


```python
def lda(X, X_mean, X_pca, y, X_test, y_test, eigenvectors_pca, kernel, mode, num_dim=None, k=5):

    N = X_pca.shape[0]
    X_mean_pca = np.mean(X_pca, axis=1).reshape(-1, 1)

    classes_mean = np.zeros((N, 15))  # 15 classes
    for i in range(X_pca.shape[1]):
        classes_mean[:, y[i]] += X_pca[:, i].astype(np.float64)

    classes_mean = classes_mean / 9

    # within-class scatter
    S_within = np.zeros((N, N))
    for i in range(15):
        mask = np.isin(y, [i])
        X_masked = X_pca[:, mask]
        for j in range(X_masked.shape[1]):
            d = X_masked[:, j].reshape(-1, 1) - classes_mean[:, i].reshape(-1, 1)
            S_within += (d @ d.T).astype(np.float64)

    # between-class scatter
    S_between = np.zeros((N, N))
    for i in range(15):
        d = classes_mean[:, i].reshape(-1,1) - X_mean_pca
        S_between += (9 * d @ d.T).astype(np.float64)

    eigenvalues_lda, eigenvectors_lda = np.linalg.eig(np.linalg.inv(S_within) @ S_between)

    sort_index = np.argsort(-eigenvalues_lda)
    if num_dim is None:
        sort_index = sort_index[:-1]  # reduce 1 dim
    else:
        sort_index = sort_index[:num_dim]

    eigenvectors_lda = np.asarray(eigenvectors_lda[:,sort_index].real)

    U = eigenvectors_pca @ eigenvectors_lda
    show_eigenface(U, 25, mode=mode, kernel=kernel, k=k)
    Z = U.T @ (X - X_mean)

    X_recover = U @ Z + X_mean
    show_reconstruction(X, X_recover, 10, mode=mode, kernel=kernel, k=k)

    # accuracy
    acc = performance(X_test, y_test, Z, y, U, X_mean, k)
    print(f"acc: {acc*100:.2f}%")
```

Figure 8: lda

- Main:
  The process involves dimensionality reduction and classification using PCA and LDA. Firstly, for PCA, the information is printed based on the value of the parameter ”k”, followed by calling the ”Pca” function to perform PCA on the training data and passing the corresponding test data. Then, for the LDA part, the results of PCA are used for LDA, including both the training and test data. Next, a loop is used to iterate through linear and polynomial kernels, calling the ”Kernel pca” function separately to perform kernel PCA on the training and test data. Finally, LDA is applied again on the results of each kernel function for classification. Throughout the process, different parameters are used to determine the retained dimensions, evaluating the effectiveness of dimensionality

reduction and classification.



```
if __name__=='__main__':

    H, W = 231, 195
    filepath = os.path.join('Yale_Face_Database', 'Training')
    X, y = imread(filepath, H, W, 9) # X: (45045, 135), y: (135, )
    filepath = os.path.join('Yale_Face_Database', 'Testing')
    X_test, y_test = imread(filepath, H, W, 2)
    k = 9
    # PCA
    print(f"K: {k}")
    eigenvectors_pca, z_train, X_mean = pca(X, y, X_test, y_test, num_dim=None, k=k)
    lda(X, X_mean, z_train, y, X_test, y_test, eigenvectors_pca, 'none', mode='lda', num_dim=None, k=k)

    # Kernel PCA
    kernel = ['linear', 'poly']
    for i in kernel:
        print(f"kernel: {i}")
        eigenvectors_pca, z_train, X_mean = kernel_pca(X, y, X_test, y_test, i, num_dim=None, k=k)
        lda(X, X_mean, z_train, y, X_test, y_test, eigenvectors_pca, i, mode='kernel lda', num_dim=None, k=k)
```

Figure 9: main

1.1 Result

- PCA:

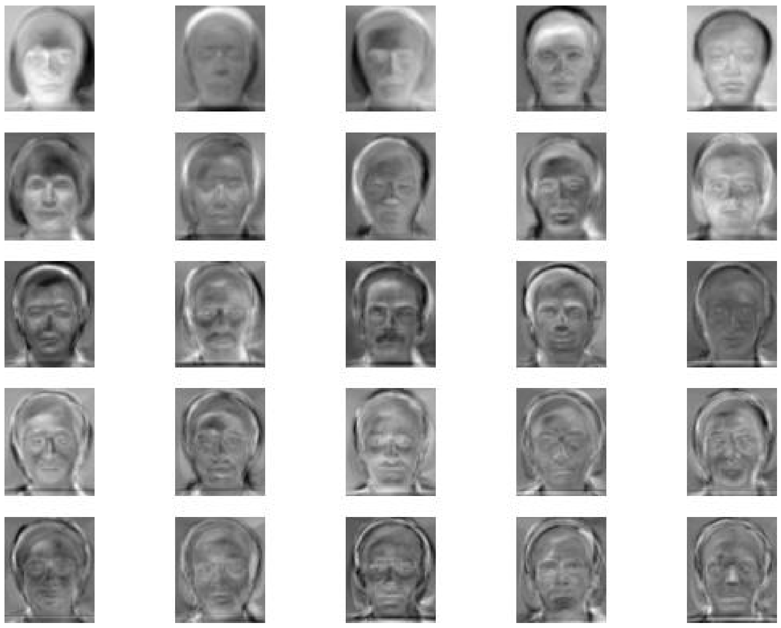


Figure 10: PCA eigenfaces

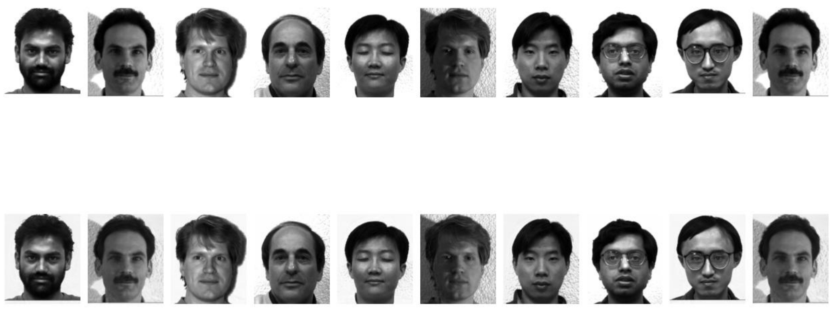


Figure 11: PCA reconstruction

- LDA:

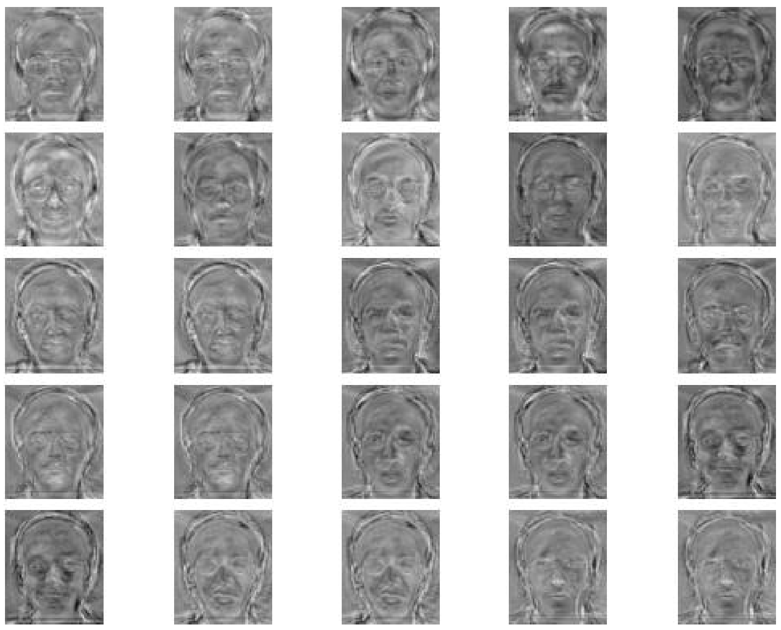


Figure 12: LDA eigenfaces

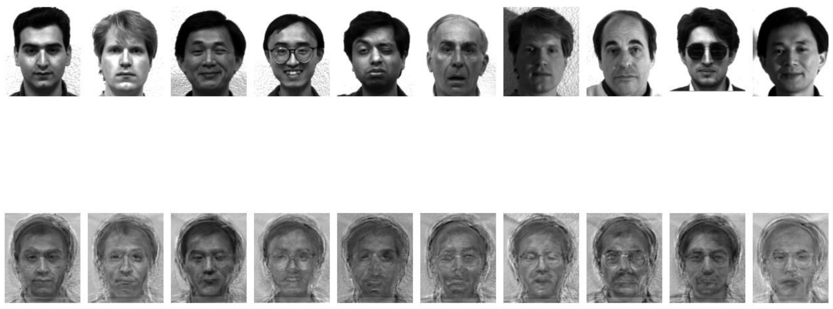


Figure 13: LDA reconstruction

- Kernel:

– Linear:

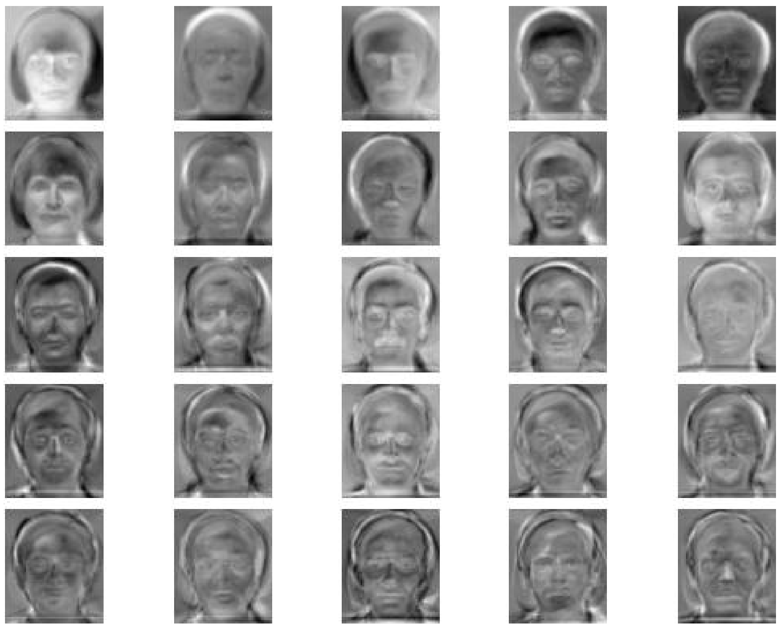


Figure 14: kernel(linear) PCA eigenfaces

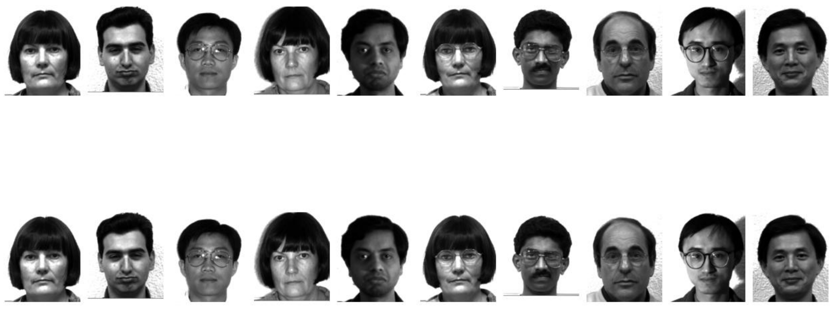


Figure 15: kernel(linear) PCA reconstruction

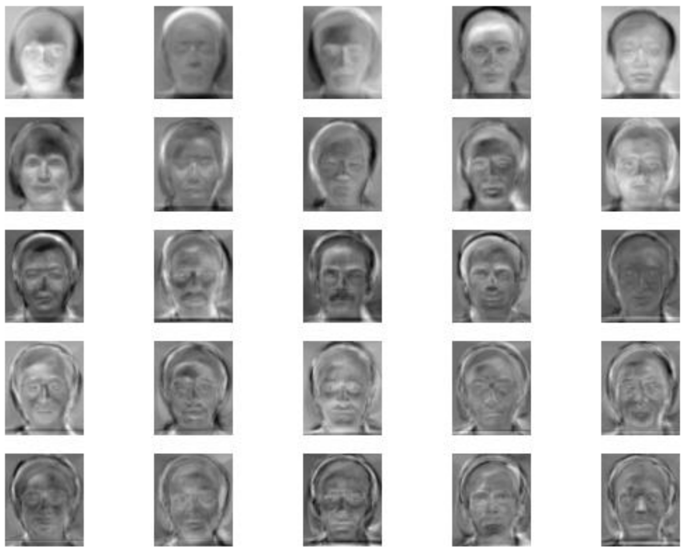


Figure 16: kernel(linear) lda eigenfaces

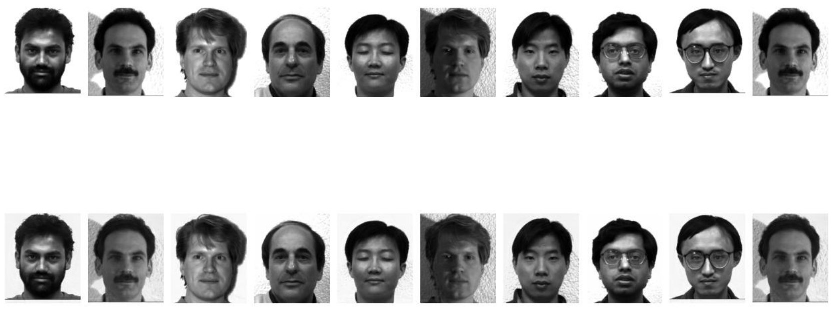


Figure 17: kernel(linear) lda reconstruction

– Polynomial:

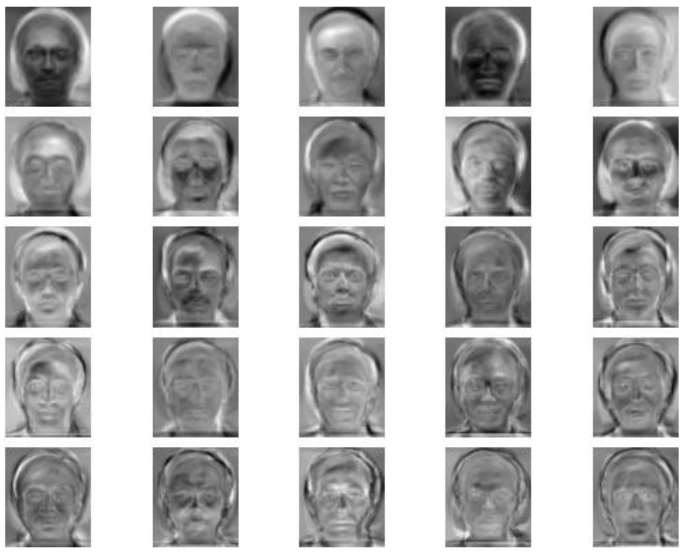


Figure 18: kernel(polynomial) PCA eigenfaces

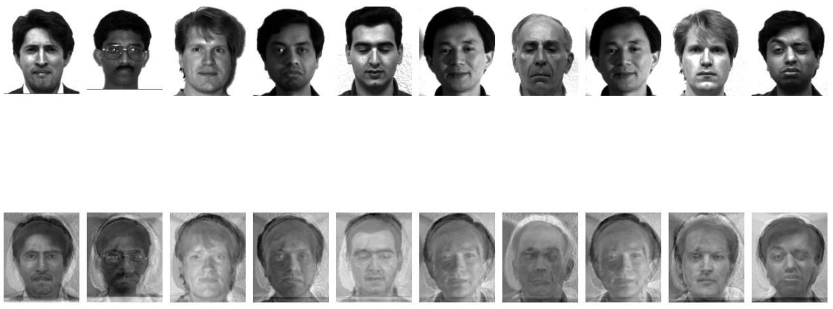


Figure 19: kernel(polynomial) PCA reconstruction

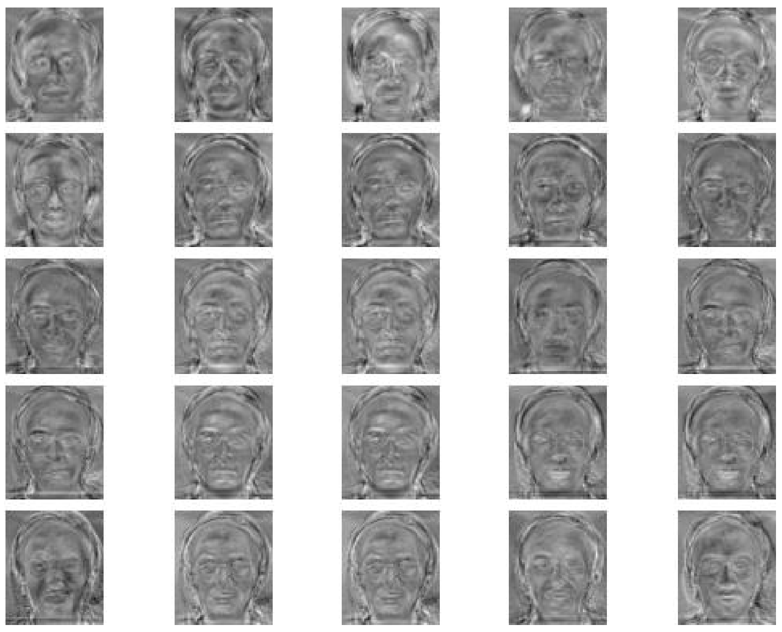


Figure 20: kernel(polynomial) lda eigenfaces

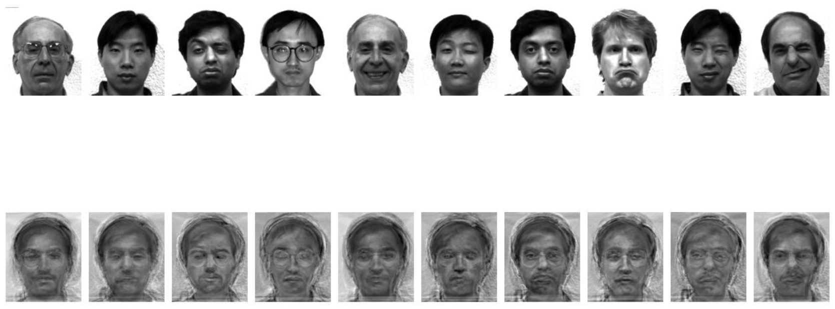


Figure 21: kernel(polynomial) lda reconstruction

- Accuracy:

Method \k	3	5	7	9
pca accuracy	83.33%	83.33%	83.33%	83.33%
lda accuracy	100.00%	96.67%	96.67%	96.67%
kernel: linear				
pca accuracy	86.67%	86.67%	86.67%	86.67%
lda accuracy	93.33%	93.33%	96.67%	100%
kernel: poly				
pca accuracy	83.33%	83.33%	83.33%	83.33%
lda accuracy	73.33%	66.67%	56.67%	53.33%

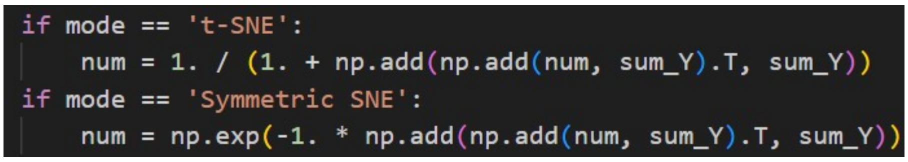
Table 1: accuracy

1.2 Observation

1. The reconstructed faces appear rather fuzzy in appearance.
2. Increasing the value of K does not necessarily lead to higher accuracy. The accuracy of PCA is more stable compared to LDA.
3. The reconstruction performance of PCA is excellent.
4. Regardless of the kernel used, the accuracy of PCA is consistently satisfactory. However, when it comes to LDA, the accuracy is subpar in the case of the polynomial kernel, while it performs well with the linear kernel.

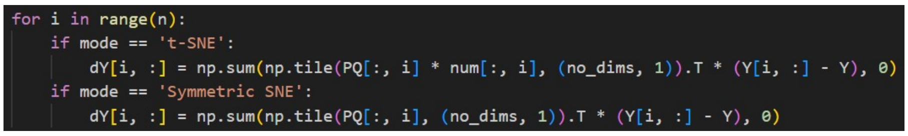
2 t-SNE

The code for Symmetric SNE and t-SNE differs in how to calculate q and the gradient. The main difference lies in addressing the crowding problem, where not all features can be well separated when mapping high-dimensional data to low-dimensional space. In the case of Symmetric SNE, it uses union probability and optimizes the function to obtain the gradient. However, it does not directly tackle the crowding problem. On the other hand, t-SNE addresses the crowding problem by utilizing t-distribution in the low-dimensional space. By incorporating t-distribution, t-SNE mitigates the issue of overlapping features in the low-dimensional representation.



```python
if mode == 't-SNE':
    num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
if mode == 'Symmetric SNE':
    num = np.exp(-1. * np.add(np.add(num, sum_Y).T, sum_Y))
```

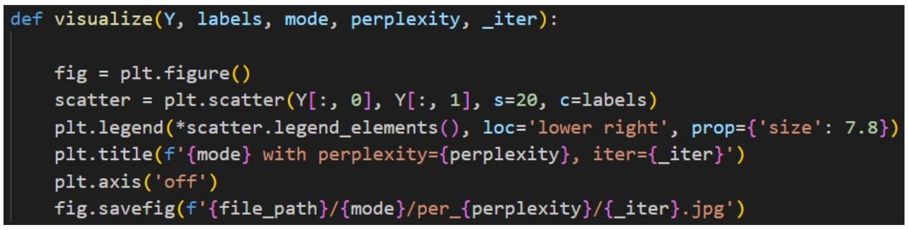
Figure 22: calculate q



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

Figure 23: gradient

- Visualize:
  Visualize the dimension-reduced results Y and save them as an image file. Use a scatter plot to represent the dimension-reduced results Y, where the position of each point is determined by the first two columns of Y, and the points are colored according to the labels. The legend is placed in the lower right corner of the plot, and a title is set, which includes information about the mode, perplexity, and iteration count. Finally, the image is saved at the specified file path.

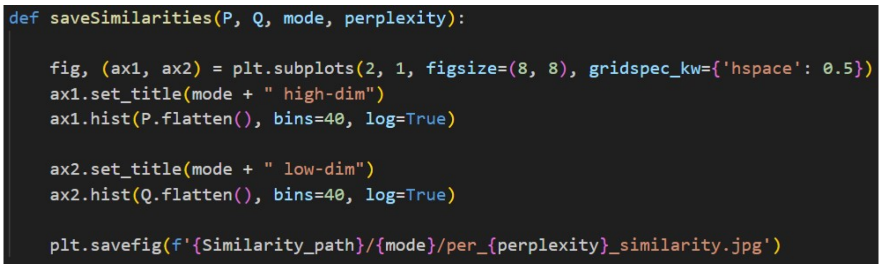


```python
def visualize(Y, labels, mode, perplexity, _iter):

    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'{mode} with perplexity={perplexity}, iter={_iter}')
    plt.axis('off')
    fig.savefig(f'{file_path}/{mode}/per_{perplexity}/{_iter}.jpg')
```

Figure 24: visualize

- SaveSimilarities:
  This code preserves the similarity between the high-dimensional and low-dimensional representations. The first subplot visualizes the flattened histogram of similarity values (P) in the high-dimensional representation, while the second subplot visualizes the flattened histogram of similarity values (Q) in the low-dimensional representation. Each subplot is

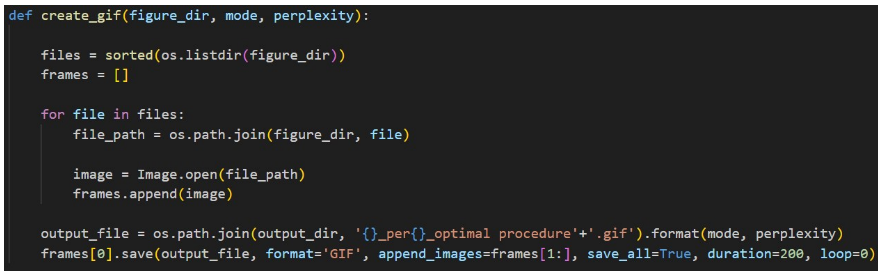
labeled with the mode (”high-dim” or ”low-dim”) and the corresponding similarity value. Afterward, the image is saved at the specified file path.



```python
def saveSimilarities(P, Q, mode, perplexity):

    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8), gridspec_kw={'hspace': 0.5})
    ax1.set_title(mode + " high-dim")
    ax1.hist(P.flatten(), bins=40, log=True)

    ax2.set_title(mode + " low-dim")
    ax2.hist(Q.flatten(), bins=40, log=True)

    plt.savefig(f'{Similarity_path}/{mode}/per_{perplexity}_similarity.jpg')
```

Figure 25: save Similarities

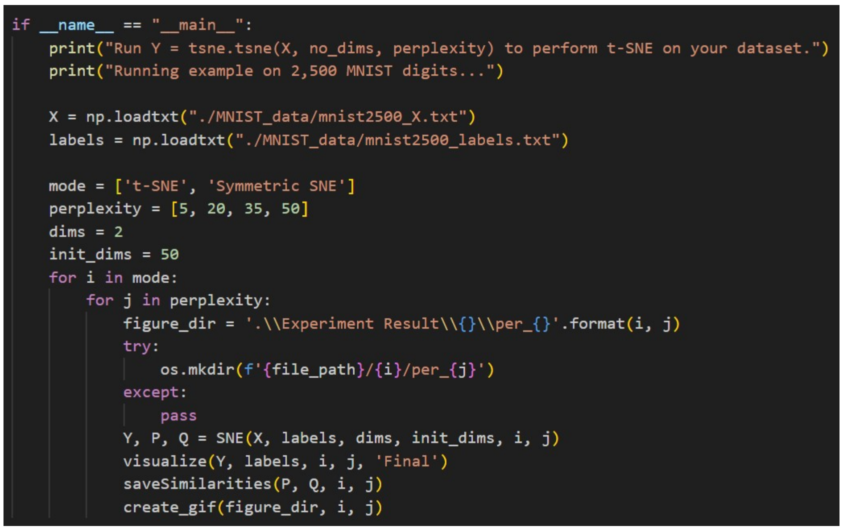
- Create gif:
  The create gif function creates a GIF animation by sequentially opening and appending image files from a specified directory containing the photos. The resulting animation is then saved as a GIF file in the specified output file. The function takes the figure directory as the input for the photo path and the mode parameter can be set to either ’t-SNE’ or ’Symmetric SNE’.



```python
def create_gif(figure_dir, mode, perplexity):

    files = sorted(os.listdir(figure_dir))
    frames = []

    for file in files:
        file_path = os.path.join(figure_dir, file)

        image = Image.open(file_path)
        frames.append(image)

    output_file = os.path.join(output_dir, '{}_per{}_optimal procedure'+'.gif').format(mode, perplexity)
    frames[0].save(output_file, format='GIF', append_images=frames[1:], save_all=True, duration=200, loop=0)
```

Figure 26: create gif

- Main:
  The available options for the variable ”mode” are ’t-SNE’ and ’Symmetric SNE’. The options for the variable ”perplexity” are 5, 20, 35, and 50.



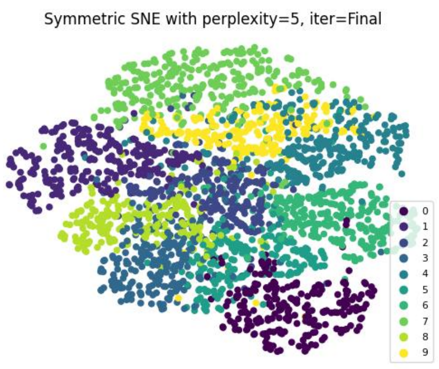
```python
if __name__ == "__main__":
    print("Run Y = tsne.tsne(X, no_dims, perplexity) to perform t-SNE on your dataset.")
    print("Running example on 2,500 MNIST digits...")

    X = np.loadtxt("./MNIST_data/mnist2500_X.txt")
    labels = np.loadtxt("./MNIST_data/mnist2500_labels.txt")

    mode = ['t-SNE', 'Symmetric SNE']
    perplexity = [5, 20, 35, 50]
    dims = 2
    init_dims = 50
    for i in mode:
        for j in perplexity:
            figure_dir = '.\\Experiment Result\\{}\\per_{}'.format(i, j)
            try:
                os.mkdir(f'{file_path}/{i}/per_{j}')
            except:
                pass
            Y, P, Q = SNE(X, labels, dims, init_dims, i, j)
            visualize(Y, labels, i, j, 'Final')
            saveSimilarities(P, Q, i, j)
            create_gif(figure_dir, i, j)
```

Figure 27: main

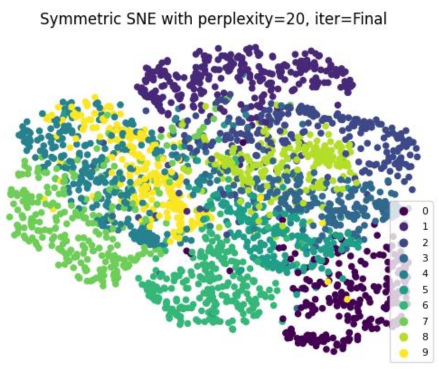
2.1 Result

Result after 1000 iterations with different perplexity:

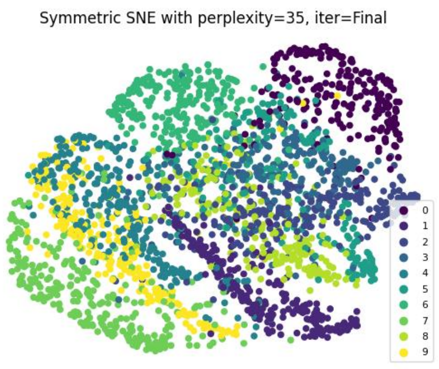
- Symmetric SNE:



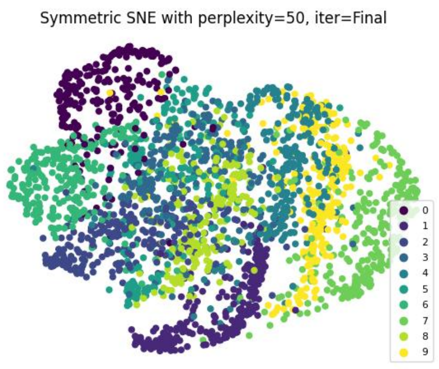
(a) perplexity=5



(b) perplexity=20

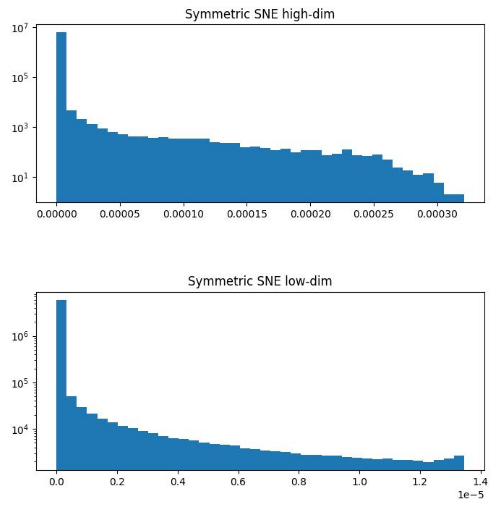


(c) perplexity=35

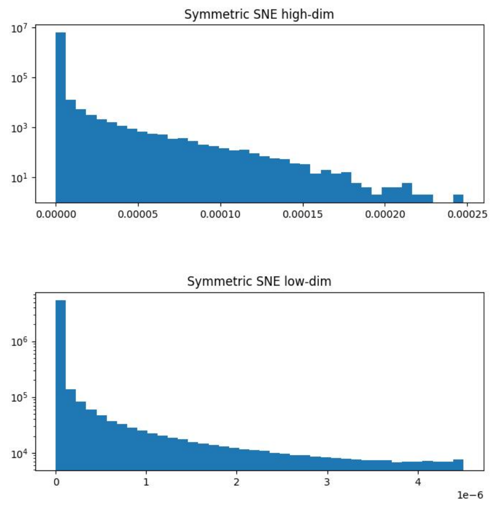


(d) perplexity=50

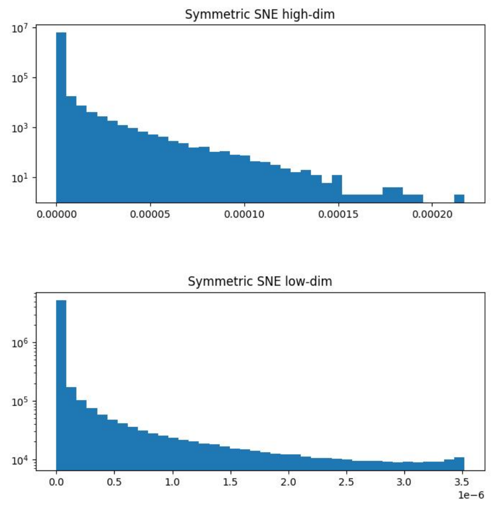
Figure 28: Symmetric SNE



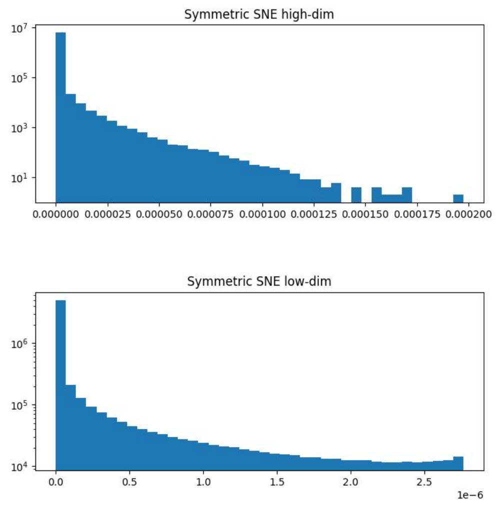
(a) perplexity=5



(b) perplexity=20



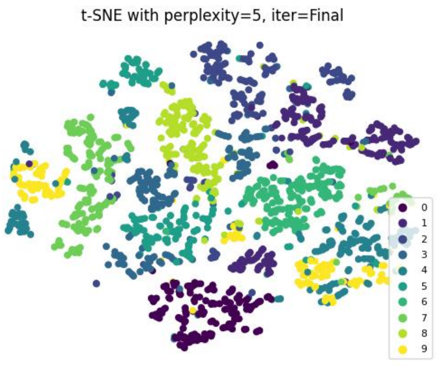
(c) perplexity=35



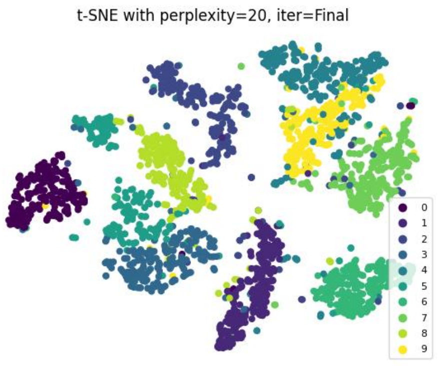
(d) perplexity=50

Figure 29: similarity

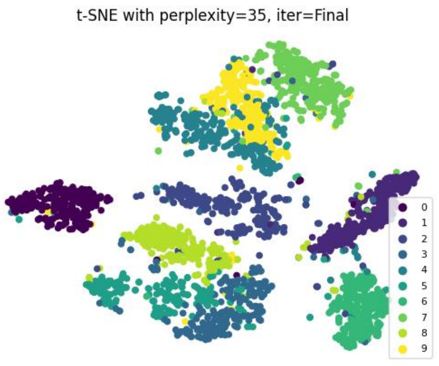
- t-SNE:



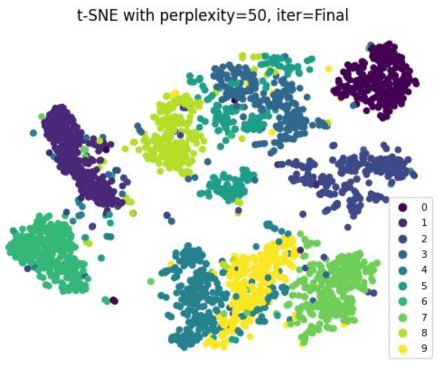
(a) perplexity=5



(b) perplexity=20

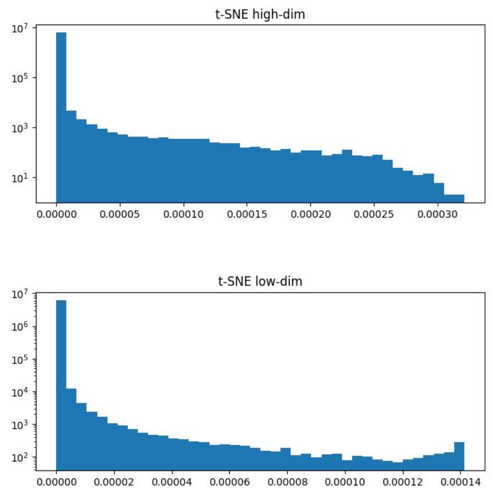


(c) perplexity=35

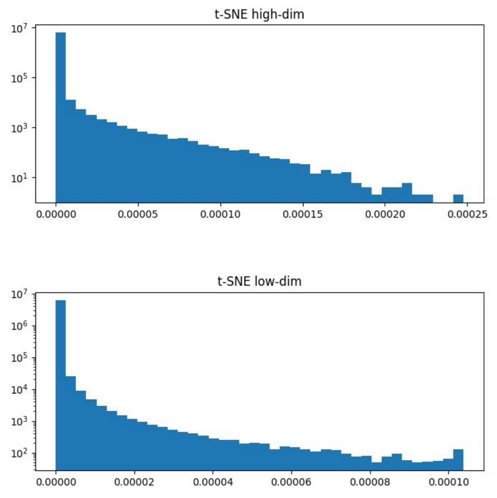


(d) perplexity=50

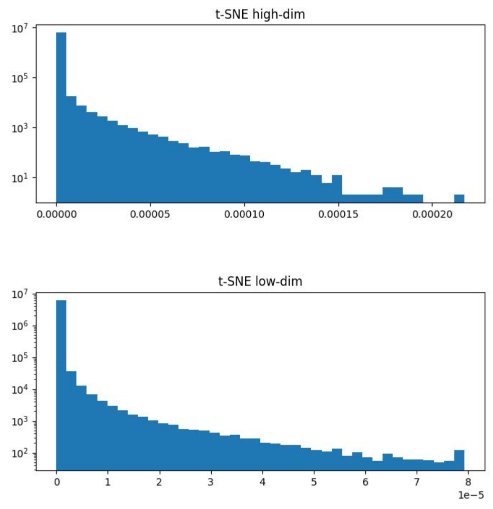
Figure 30: t-sne



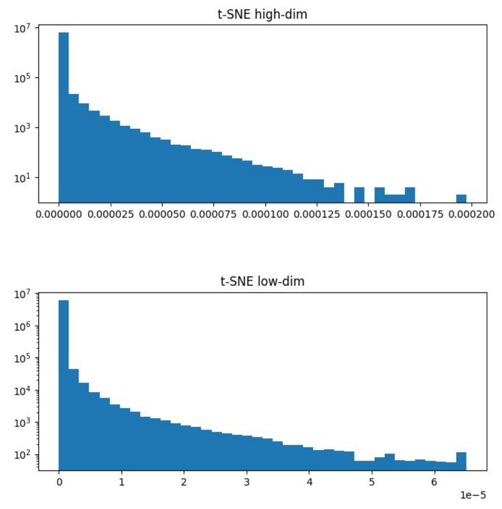
(a) perplexity=5



(b) perplexity=20



(c) perplexity=35



(d) perplexity=50

Figure 31: similarity

2.2 Observation

For the results:

1. the Symmetric SNE results exhibit a higher degree of crowding compared to t-SNE, indicating that t-SNE demonstrates superior data classification capabilities.

2. The results of both Symmetric SNE and t-SNE exhibit heightened levels of crowding as the perplexity value increases, as opposed to when it is set at a lower value.

3. Perplexity has a stronger impact on t-SNE compared to symmetric SNE, primarily because symmetric SNE tends to experience a crowding problem.

4. Via a gif, we can observe the grouping of data during the training process. In the case of t-SNE, different classes are rapidly separated and placed in distinct regions within the 2D space. Subsequently, the distribution is adjusted to achieve a more generalized representation, on the other hand, symmetric SNE only makes subtle movements of the data within the low-dimensional space compared to SNE. It then makes slight modifications to refine the distribution and improve its overall quality.

For pairwise similarity:

1. The shape of the perplexity parameter does not affect the pairwise similarity in either Symmetric SNE or t-SNE.

2. The range of results obtained from Symmetric SNE is greater than that of t-SNE, contributing to a more prominent crowding issue in Symmetric SNE. In contrast, t-SNE exhibits a comparatively lesser degree of crowding.