# 1. Code with detailed explanations

- **Kernel Eigenfaces - PCA**

In part 1, we have to show 25 eigenfaces and randomly pick 10 images to show their reconstruction. First, we need to load the data:

```
def read_pgm(path, resized_factor):
    image_files = sorted(os.listdir(path))
    images = []
    labels = []
    file_names = []

    for image_file in image_files:
        with Image.open(os.path.join(path, image_file)) as image:
            width, height = image.size
            width_resized, height_resized = width // resized_factor,
                                            height // resized_factor
            image_resized = image.resize((width_resized, height_resized))
            image = np.array(image_resized).flatten()

        images.append(image)
        labels.append(int(image_file.split('.')[0][7:9]) - 1)
        file_name_prefix, _ = os.path.splitext(image_file)
        file_names.append(file_name_prefix)

    return np.array(images), np.array(labels), file_names, width_resized,
           height_resized
```

↑ For ease of computation, the original image dimensions were each divided by 3, reshaping them into (77, 65).

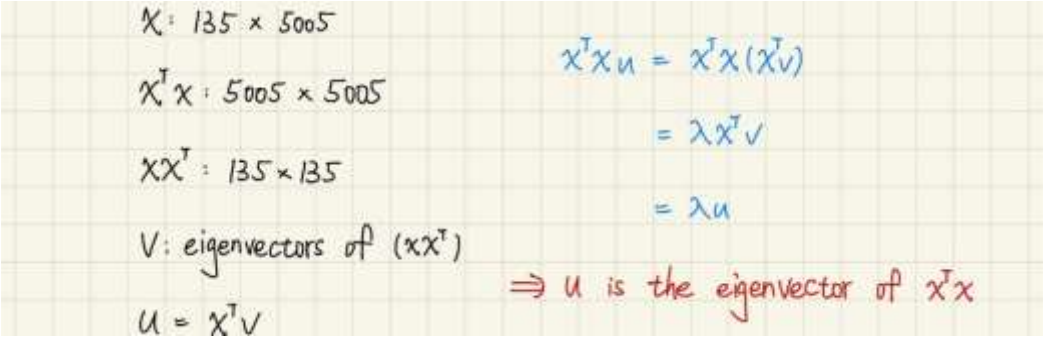
Next, we can perform PCA. The steps include:

1. Centering the data.
2. Computing the covariance matrix.
3. Extracting the eigenvectors of the covariance matrix and selecting the eigenvectors corresponding to the top **n_components** eigenvalues.

4. Projecting the data onto the principal component space. ↓

```
def pca(X, n_components):
    X_mean = np.mean(X, axis=0, keepdims=True)  # shape: (1, 5005)
    X_centered = X - X_mean  # shape: (135, 5005)

    S = np.cov(X_centered, rowvar=True, bias=False)  # covariance matrix,
shape: (135, 135)
    eigenvalues, eigenvectors = np.linalg.eigh(S)  # eigenvalues are in
ascending order
    sort_index = np.argsort(-eigenvalues)  # sort eigenvalues in descending
order
    eigenvectors = eigenvectors[:, sort_index]  # shape: (135, 135)

    # calculate eigenvectors of X.T @ X to get the projection matrix W
    projection_matrix = X_centered.T @ eigenvectors  # shape: (5005, 135)
    projection_matrix = projection_matrix[:, :n_components]  # shape:
(5005, n_components)
    projection_matrix = projection_matrix /
                        np.linalg.norm(projection_matrix, axis=0)  #
normalization

    return projection_matrix, X_mean  # X_mean would be used to reconstruct
the image
```

The eigenvectors of the covariance matrix **S** are the desired eigenfaces. Note that since the shape of **S** is (5005, 5005), directly computing its eigenvectors would be computationally expensive. We use the technique shown in the following image to reduce the computational load and obtain the final orthogonal projection matrix **W** (the collection of all u). ↓



The projected data **Z** and the reconstructed images can be easily obtained using the following formulas, while show_eigenface() and show_reconstruction() are two visualization functions, the details of which are omitted here. ↓

```
    n_components = 25
    train_W, train_X_mean = pca(train_images, n_components)  # shape:
(5005, n_components), (1, 5005)
    show_eigenface(train_W, height, width)

    train_Z = (train_images - train_X_mean) @ train_W  # Z = X_centered @
W, shape: (135, n_components)
    train_images_reconstructed = train_Z @ train_W.T + train_X_mean
    show_reconstruction(train_images, train_images_reconstructed, height,
                        width, train_file_names)
```

In part 2, we have to do face recognition and use k nearest neighbor to classify which subject the testing image belongs to. We can implement KNN classification by calculating the distance between the projected data of the images in the training and test sets after being projected through the PCA projection matrix. ↓

```
def performance(train_Z, train_labels, test_Z, test_labels, k=5):
    predictions = np.zeros(test_Z.shape[0])

    for i in range(test_Z.shape[0]):
        distance = np.sum(np.square(test_Z[i] - train_Z), axis=1)  # shape:
(135,)
        sort_index = np.argsort(distance)
        nearest_neighbors = train_labels[sort_index[:k]]  # k nearest
neighbors
        labels, counts = np.unique(nearest_neighbors, return_counts=True)
        predictions[i] = labels[np.argmax(counts)]

    accuracy = np.mean(test_labels == predictions)

    return accuracy
```

In part 3, we need to implement kernel PCA and do face recognition. In the code implementation, the main difference from the previous approach is that the covariance

matrix S is not calculated. Instead, the kernel K is computed, and care must be taken to center the kernel when projecting data points. ↓

```
def compute_kernel(X, X_new, kernel_type):
    if kernel_type == 'linear':
        K = np.dot(X_new, X.T)

    elif kernel_type == 'polynomial':
        gamma = 0.005
        coefficient = 1
        degree = 2
        K = (gamma * np.dot(X_new, X.T) + coefficient) ** degree

    elif kernel_type == 'rbf':
        gamma = 0.002
        distances = cdist(X_new, X, 'sqeuclidean')  # shape:
(n_samples_test, n_samples_train)
        K = np.exp(-gamma * distances)

    return K


def kernel_pca(X, n_components, kernel_type):
    n_samples = X.shape[0]
    one_N = np.ones((n_samples, n_samples)) / n_samples  # one_N is a N x N
matrix with all elements equal to 1/N
    K = compute_kernel(X, X, kernel_type)  # shape: (n_samples, n_samples)
    K_centered = K - np.dot(one_N, K) - np.dot(K, one_N)
                   + np.dot(np.dot(one_N, K), one_N)

    eigenvalues, eigenvectors = np.linalg.eigh(K_centered)
    sort_index = np.argsort(-eigenvalues)  # sort eigenvalues in descending
order
    eigenvectors = eigenvectors[:, sort_index]

    principal_components = eigenvectors[:, :n_components]
    principal_components = principal_components /
                           np.linalg.norm(principal_components, axis=0)  #
normalization
```

```
    return principal_components, K_centered


def project_test(X_train, X_test, principal_components, kernel_type):
    K_test = compute_kernel(X_train, X_test, kernel_type)
    K_train = compute_kernel(X_train, X_train, kernel_type)

    n_samples_train = X_train.shape[0]
    n_samples_test = X_test.shape[0]
    one_n_test_train = np.ones((n_samples_test, n_samples_train)) /
                               n_samples_train
    one_n_train_train = np.ones((n_samples_train, n_samples_train)) /
                               n_samples_train

    K_test_centered = (K_test
                       - np.dot(one_n_test_train, K_train)
                       - np.dot(K_test, one_n_train_train)
                       + np.dot(one_n_test_train, np.dot(K_train,
                                one_n_train_train)))

    # principal_components: orthogonal projection W
    X_test_projected = np.dot(K_test_centered, principal_components)  #
shape: (30, n_components)

    return X_test_projected
```

The implementation process of the above code can be found in Unsupervised_Learning.pdf, pages 116 to 128.

- **Kernel Eigenfaces - LDA**

Some utility functions unrelated to the algorithm itself (e.g., **read_pgm()**, **show_fisherface()**, **show_reconstruction()**, **performance()**) are similar to their implementations in PCA. Therefore, they are not elaborated upon here. For parts 1 and 2, the derivation of the formulas for the within-class scatter matrix and the between-

class scatter matrix can be found on page 179 in Unsupervised_Learning.pdf. On the other hand, the derivation of the projection matrix **W** can be found on page 177 in Unsupervised_Learning.pdf. ↓

if now we are dealing with multi-class cases $(\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_k)$:

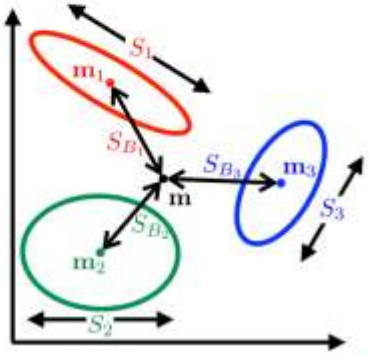
*within-class scatter*: $S_W = \sum_{j=1}^{k} S_j$, where $S_j = \sum_{i \in \mathcal{C}_j} (x_i - \mathbf{m}_j)(x_i - \mathbf{m}_j)^\top$

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

*between-class scatter*:

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

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




[nice article from Prof. Chi-Cheng Jou]

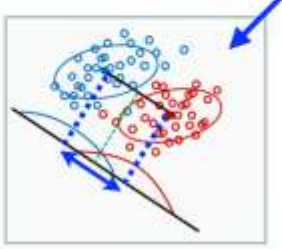
we assume $S_W$ is invertible (yes if $n > D$ which hosts most of time)

$$\Rightarrow S_W^{-1} S_B \mathrm{w} = \lambda \mathrm{w}$$

since $S_B$ is the outer product of two vectors, its rank is at most 1

$$\text{as } S_B = (\mathbf{m}_2 - \mathbf{m}_1)(\mathbf{m}_2 - \mathbf{m}_1)^\top$$

$$\text{then } S_B \mathrm{w} = (\mathbf{m}_2 - \mathbf{m}_1)\underbrace{(\mathbf{m}_2 - \mathbf{m}_1)^\top \mathrm{w}}_{\text{scalar}} = (\mathbf{m}_2 - \mathbf{m}_1) * \lambda_{\mathrm{w}}$$



$$\Rightarrow S_{\mathrm{w}}^{-1} S_B \mathrm{w} = S_{\mathrm{w}}^{-1} (\mathbf{m}_2 - \mathbf{m}_1) * \lambda_{\mathrm{w}} = \lambda \mathrm{w}$$

since it won't matter if w is scaled, then:

$$\Rightarrow \mathrm{w} = S_{\mathrm{w}}^{-1} (\mathbf{m}_2 - \mathbf{m}_1)$$

[nice article from Prof. Chi-Cheng Jou]

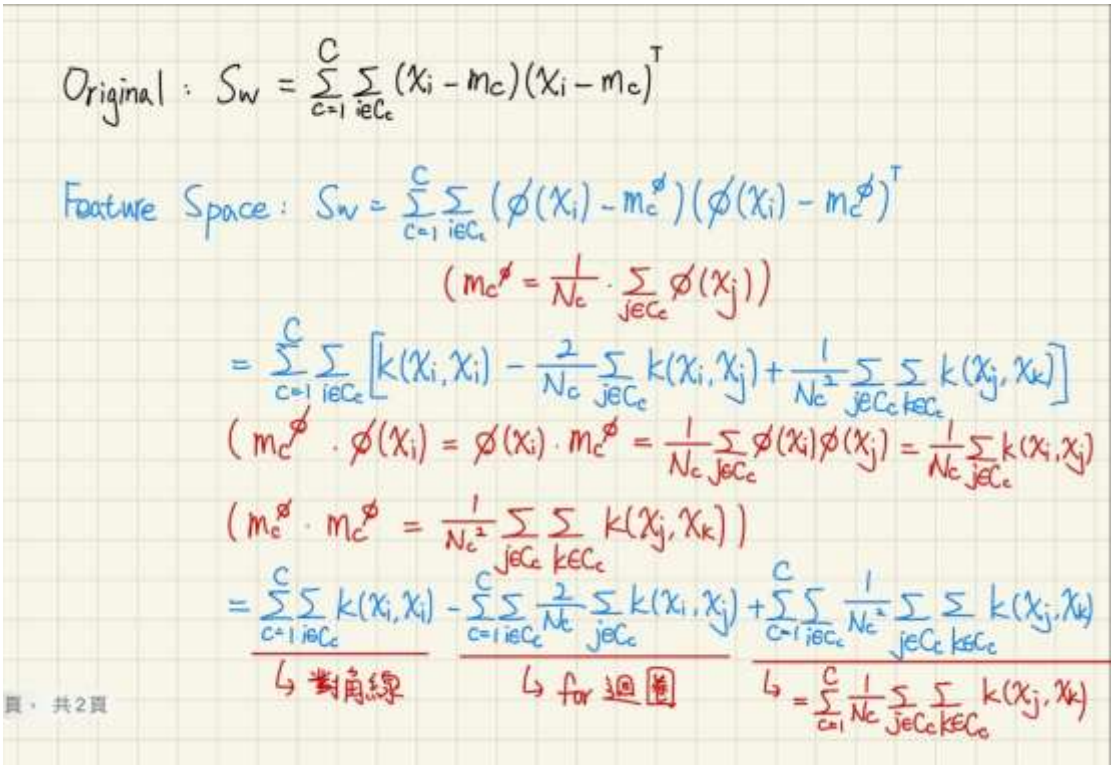
```
def lda(X, y, n_components):
    n_classes = len(np.unique(y))  # 15 classes
    # max_components = n_classes - 1


    # if n_components > max_components:
    #     print(f"Warning: n_components reduced from {n_components} to
{max_components}")
    #     n_components = max_components
```

```
    n_samples, n_features = X.shape
    overall_mean = np.mean(X, axis=0, keepdims=True)  # shape: (1, 5005)
    class_means = np.zeros((n_classes, n_features))

    S_W = np.zeros((n_features, n_features))  # shape: (5005, 5005)
    S_B = np.zeros((n_features, n_features))  # shape: (5005, 5005)

    for i in range(n_classes):
        x_i = X[y == i]  # samples of class i, shape: (9, 5005)
        class_means[i] = np.mean(x_i, axis=0, keepdims=True)
        within_class_diff = x_i - class_means[i]  # shape: (9, 5005)
        S_W += within_class_diff.T @ within_class_diff

        between_class_diff = (class_means[i] - overall_mean)  # shape: (1,
5005)
        S_B += len(x_i) * (between_class_diff.T @ between_class_diff)

    S_W += np.eye(n_features) * 1e-6  # pseudo inverse

    eigenvalues, eigenvectors = np.linalg.eigh(np.linalg.inv(S_W) @ S_B)
    sort_index = np.argsort(-eigenvalues)
    projection_matrix = eigenvectors[:, sort_index]
    projection_matrix = projection_matrix[:, :n_components]
    projection_matrix = projection_matrix /
                        np.linalg.norm(projection_matrix, axis=0)

    return projection_matrix, overall_mean
```

Similarly, fisherfaces are the eigenvectors of (S_W)^-1(S_B). The subsequent functions, show_fisherfaces() and show_reconstruction(), are also visualization functions. Their process is similar to PCA's, and details are not elaborated here.

In part 3, we have to implement kernel LDA. By replacing the formula from page 179 of Unsupervised_Learning.pdf with feature representation and expanding each term, we can apply the kernel trick and obtain the following expressions (for the between-class scatter matrix, I skip the derivation):

$$S_B^{\phi} = \sum_{c=1}^{C} N_c \left( \frac{1}{N_c^2} \sum_{i \in C_c} \sum_{j \in C_c} k(\mathbf{x}_i, \mathbf{x}_j) - 2 \frac{1}{N_c N} \sum_{i \in C_c} \sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{x}_j) + \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{x}_j) \right)$$




```python
def kernel_lda(X, y, n_components, kernel_type):
    N = X.shape[0]  # number of samples
    one_N = np.ones((N, N)) / N
    K = compute_kernel(X, X, kernel_type)  # shape: (N, N), where N is the number of samples
    K_centered = K - one_N @ K - K @ one_N + one_N @ K @ one_N
    unique_classes = np.unique(y)

    # compute SWφ (within-class scatter matrix)
    S_W_phi = np.zeros((N, N))

    for c in unique_classes:
        indices_c = np.where(y == c)[0]
        N_c = len(indices_c)

        # term1:
        term1 = np.zeros((N, N))
        for i in indices_c:
            term1[i, i] = K[i, i]
```

```
        # term2:
        term2 = np.zeros((N, N))
        for i in indices_c:
            for j in indices_c:
                term2[i, j] += K[i, j] / N_c

        # term3
        term3 = np.zeros((N, N))
        for j in indices_c:
            for k in indices_c:
                term3[j, k] += K[j, k] / N_c

        S_W_phi += (term1 - 2 * term2 + term3)
    # print("S_W_phi", S_W_phi)

    # compute SBφ (between-class scatter matrix)
    S_B_phi = np.zeros((N, N))

    for c in unique_classes:
        indices_c = np.where(y == c)[0]
        N_c = len(indices_c)

        # term1
        term1 = np.zeros((N, N))
        for i in indices_c:
            for j in indices_c:
                term1[i, j] = K[i, j] / (N_c ** 2)

        # term2
        term2 = np.zeros((N, N))
        for i in indices_c:
            for j in range(N):
                term2[i, j] = K[i, j] / (N_c * N)

        # term3
        term3 = np.zeros((N, N))
        for i in range(N):
```

```
            for j in range(N):
                term3[i, j] = K[i, j] / (N * N)

        S_B_phi += N_c * (term1 - 2 * term2 + term3)
    # print("S_B_phi", S_B_phi)

    S_W_phi += np.eye(N) * 1e-6  # pseudo inverse

    eigenvalues, eigenvectors = np.linalg.eigh(np.linalg.inv(S_W_phi) @
                                S_B_phi)
    sort_index = np.argsort(-eigenvalues)
    eigenvectors = eigenvectors[:, sort_index]

    principal_components = eigenvectors[:, :n_components]
    principal_components = principal_components /
                          np.linalg.norm(principal_components, axis=0)

    return principal_components, K_centered
```

In summary, there are a few points to note:

1. LDA uses class labels when calculating the projection matrix.
2. Since the feature representation is not explicitly written, the **overall_mean** and **class_mean** in the LDA formula should be expanded before applying the kernel trick.

- **t-SNE**

In part 1, we must modify the code to return the reference code to symmetric SNE. Two parts of the code need to be modified: 1. The formula for the low-dimensional pairwise similarity **Q** needs to be modified. 2. The calculation of the gradient needs to be adjusted. The formulas for computing the pairwise similarity **Q** and the gradient for t-SNE and symmetric SNE can be found on p.172 and p.168 of the Unsupervised_Learning.pdf, respectively:

## t-SNE

- In high-D, Gaussian disb. to turn distances into probabilities; in low-D, Student t-disb. is used to alleviate crowding problem

$$p_{ij} = \frac{\exp(-\,||\,x_i - x_j\,||^2\,/(2\sigma^2))}{\sum_{k\neq l}\exp(-\,||\,x_l - x_k\,||^2\,/(2\sigma^2))}$$

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

$$C = KL(P||Q) = \sum_i \sum_{j\neq i} p_{ij}\log\frac{p_{ij}}{q_{ij}}$$

large $p_{ij}$ modeled by small $q_{ij}$: large penalty
small $p_{ij}$ modeled by large $q_{ij}$: small penalty → t-SNE mainly preserves *local* similarity structure of data

- gradient:

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

[nice ref: https://wiki.math.uwaterloo.ca/statwiki/index.php?title=visualizing_Data_using_t-SNE] 17

## SNE (Stochastic Neighbor Embedding)

- **Symmetric** SNE
  - such that $p_{ji}=p_{ij}$, $q_{ji}=q_{ij}$, simplifying gradient

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

  - However, in practice we symmetrize (or average) the conditionals

joint probability of picking the pair $i$, $j$

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

original computation of $p_{ij}$ will have very small value if $x_i$ is outlier (where $||\,x_i - x_j\,||^2$ is large), thus $y_i$ has little contribution to cost
*we want to have all data contribute!*

$$p_{ij} = \frac{\exp(-\,||\,x_i - x_j\,||^2\,/(2\sigma^2))}{\sum_{k\neq l}\exp(-\,||\,x_l - x_k\,||^2\,/(2\sigma^2))}$$

same $\sigma$ for all; compare with all pairwise distance

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

[nice ref: https://wiki.math.uwaterloo.ca/statwiki/index.php?title=visualizing_Data_using_t-SNE] 16

```
        if symmetric_sne:
            num = np.exp(-np.add(np.add(num, sum_Y).T, sum_Y))  # modified
from tsne to symmetric sne
        else:
            num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))

        num[range(n), range(n)] = 0.
```

```
        Q = num / np.sum(num)
        Q = np.maximum(Q, 1e-12)

        # Compute gradient
        PQ = P - Q
        for i in range(n):
            if symmetric_sne:
                dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T *
                                  (Y[i, :] - Y), 0)  # modified from tsne to
symmetric sne
            else:
                dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i],
                                  (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

In parts 2, 3, and 4, we need to add some visualization functions to compare the results. Since these visualization functions do not involve complex algorithms, I will only present the code in this report and not go into the details. A detailed comparison of the results will be provided in the following sections. ↓

```
def create_animation(Y_frames, labels, filename, symmetric_sne):
    # Create the directory if it doesn't exist
    output_dir = "./tsne_visualization"
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    frames = []
    fig, ax = plt.subplots(figsize=(8, 8), dpi=100)

    unique_labels = np.unique(labels)
    distinct_colors = ['#e6194B', '#3cb44b', '#4363d8', '#f58231',
                       '#911eb4', '#42d4f4', '#f032e6', '#bfef45',
                       '#fabed4', '#469990']
    color_map = dict(zip(unique_labels, distinct_colors))

    for i, Y in enumerate(Y_frames):
        ax.clear()
        for label in unique_labels:
            mask = labels == label
```

```
            ax.scatter(Y[mask, 0], Y[mask, 1], c=color_map[label],
                       label=f'Class {int(label)}', alpha=0.6, s=20)

        ax.set_title(f'Iteration {(i + 1) * 10}')

        if symmetric_sne:
            ax.set_xlim(-8, 8)
            ax.set_ylim(-8, 8)

        else:
            ax.set_xlim(-80, 80)
            ax.set_ylim(-80, 80)

        fig.canvas.draw()

        frame = np.array(fig.canvas.renderer.buffer_rgba())
        frames.append(frame[:,:,:3])

    gif_path = os.path.join(output_dir, filename)
    imageio.v2.mimsave(gif_path, frames, loop=1, fps=5)
    print("Optimization procedure shows in", gif_path)

    # Save the final .png image
    png_path = gif_path.replace('.gif', '.png')
    plt.savefig(png_path, dpi=100, bbox_inches='tight')
    plt.close()
    print("The final result shows in", png_path)


def plot_pairwise_similarity_distribution(P, Q, method, perplexity):
    output_dir = "./tsne_visualization"
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    plt.figure(figsize=(12, 6))

    # Plot P (high-dimensional space)
    plt.subplot(1, 2, 1)
    plt.hist(P.flatten(), bins=35, log=True, density=True)
```

```
    plt.title(f"{method} High-dimensional space (P) with perplexity ",
              fontsize=12)
    plt.xlabel('Pairwise Similarity')
    plt.ylabel('Frequency (log scale in proportion)')
    ax = plt.gca()
    ax.xaxis.set_major_locator(MaxNLocator(nbins=6))
    ax.xaxis.set_major_formatter(ScalarFormatter(useMathText=True))

    # Plot Q (low-dimensional space)
    plt.subplot(1, 2, 2)
    plt.hist(Q.flatten(), bins=35, log=True, density=True)
    plt.title(f"{method} Low-dimensional space (Q)", fontsize=12)
    plt.xlabel('Pairwise Similarity')
    plt.ylabel('Frequency (log scale in proportion)')
    ax = plt.gca()
    # ax.xaxis.set_major_locator(MaxNLocator(nbins=6))
    ax.xaxis.set_major_formatter(ScalarFormatter(useMathText=True))

    plt.tight_layout()
    plt.savefig(f"{output_dir}/similarity_distribution_{method}_perplexity_
                {int(perplexity)}.png")
    # plt.show()
```

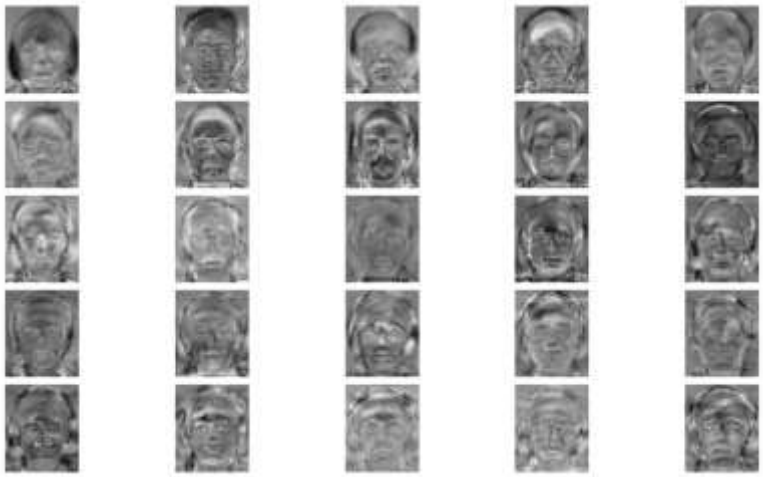
# 2. Experiments and Discussion

- **Kernel Eigenfaces**

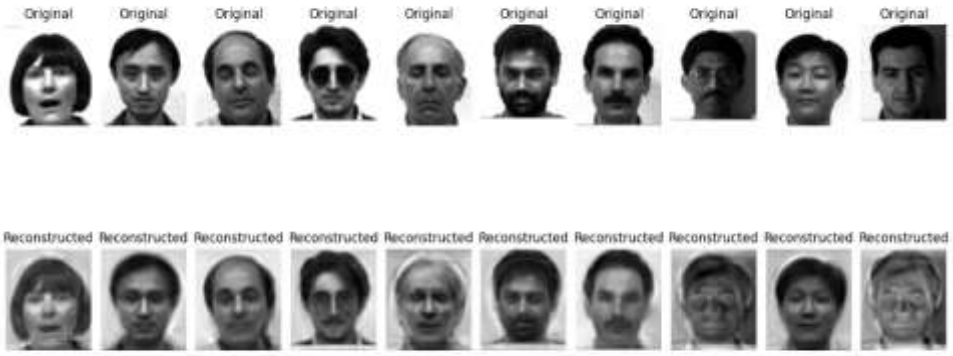
➢ Part 1:

First 25 eigenfaces:

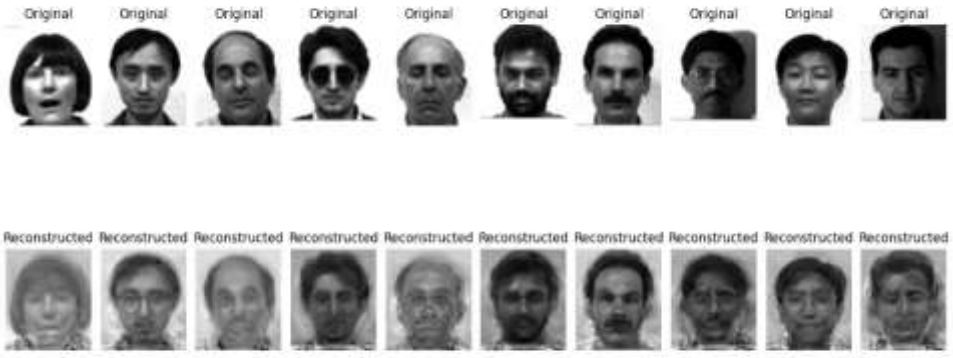
First 25 fisherfaces:



10 reconstructed images from PCA:




10 reconstructed images from LDA:




Since I set a random seed, the randomly selected images will be the same. Based

on the photos of Eigenfaces and Fisherfaces I generated, the style differences between the two are not significant. Fisherfaces retain slightly more details, while the principal components they focus on are different. The reconstructed images clearly show the differences between PCA and LDA. PCA performs better in restoring "expressions," such as surprised or winked expressions. On the other hand, LDA is better at reconstructing the facial contours. For example, in the first and third images from right to left, comparing the reconstructed results of PCA and LDA, we can see that the two pictures reconstructed by PCA only differ in skin tone. In contrast, the two images reconstructed by LDA show distinctly different facial contours, representing two other individuals. This could be because LDA performs dimensionality reduction based on class labels, which allows it to distinguish the facial contours of the 15 individuals more accurately. In contrast, PCA focuses on maximizing the variance of the features, so its principal components might include a combination of different facial features, leading to less precise differentiation between individuals and a stronger emphasis on overall variation.

- Part 2:

The hyperparameter k represents the number of nearest neighbors to be selected.

1. PCA:

| | k=1 | k=3 | k=5 | k=7 | k=9 |
|---|---|---|---|---|---|
| Accuracy | 83.33% | 83.33% | 86.67% | 86.67% | 83.33% |

2. LDA:

| | k=1 | k=3 | k=5 | k=7 | k=9 |
|---|---|---|---|---|---|
| Accuracy | 86.67% | 93.33% | 96.67% | 93.33% | 93.33% |

↑ We can observe that using LDA for face recognition yields better results than using PCA, which is quite intuitive. After all, LDA takes label information into account

when computing the projection matrix, whereas PCA does not.

- Part 3:

  The hyperparameter k represents the number of nearest neighbors to be selected.

  The hyperparameters of the polynomial kernel and rbf kernel for kernel PCA are:

  1. Polynomial kernel: gamma = 0.005, coefficient = 1, degree = 2
  2. RBF kernel: gamma = 0.002

  The hyperparameters of the polynomial kernel and rbf kernel for kernel LDA are:

  1. Polynomial kernel: gamma = 0.005, coefficient = 1, degree = 2
  2. RBF kernel: gamma = 0.001

  The kernel hyperparameter is set arbitrarily.

  1. PCA:

| Linear kernel | k=1 | k=3 | k=5 | k=7 | k=9 |
|---|---|---|---|---|---|
| Accuracy | 80.00% | 83.33% | 80.00% | 80.00% | 83.33% |
| Polynomial kernel | k=1 | k=3 | k=5 | k=7 | k=9 |
| Accuracy | 80.00% | 83.33% | 80.00% | 86.67% | 83.33% |
| RBF kernel | k=1 | k=3 | k=5 | k=7 | k=9 |
| Accuracy | 83.33% | 86.67% | 83.33% | 80.00% | 86.67% |

  2. LDA:

| Linear kernel | k=1 | k=3 | k=5 | k=7 | k=9 |
|---|---|---|---|---|---|
| Accuracy | 80.00% | 83.33% | 83.33% | 80.00% | 80.00% |
| Polynomial kernel | k=1 | k=3 | k=5 | k=7 | k=9 |
| Accuracy | 80.00% | 86.67% | 86.67% | 80.00% | 83.33% |
| RBF kernel | k=1 | k=3 | k=5 | k=7 | k=9 |
| Accuracy | 83.33% | 86.67% | 90.00% | 83.33% | 80.00% |

  ↑ We can see that the results of kernel LDA are slightly better and more stable than

those of kernel PCA. Compared to the original PCA and LDA, kernel PCA generally maintains similar performance, while kernel LDA shows a noticeable decline. The reasons for this, aside from potential issues in my code implementation, can be summarized as follows:

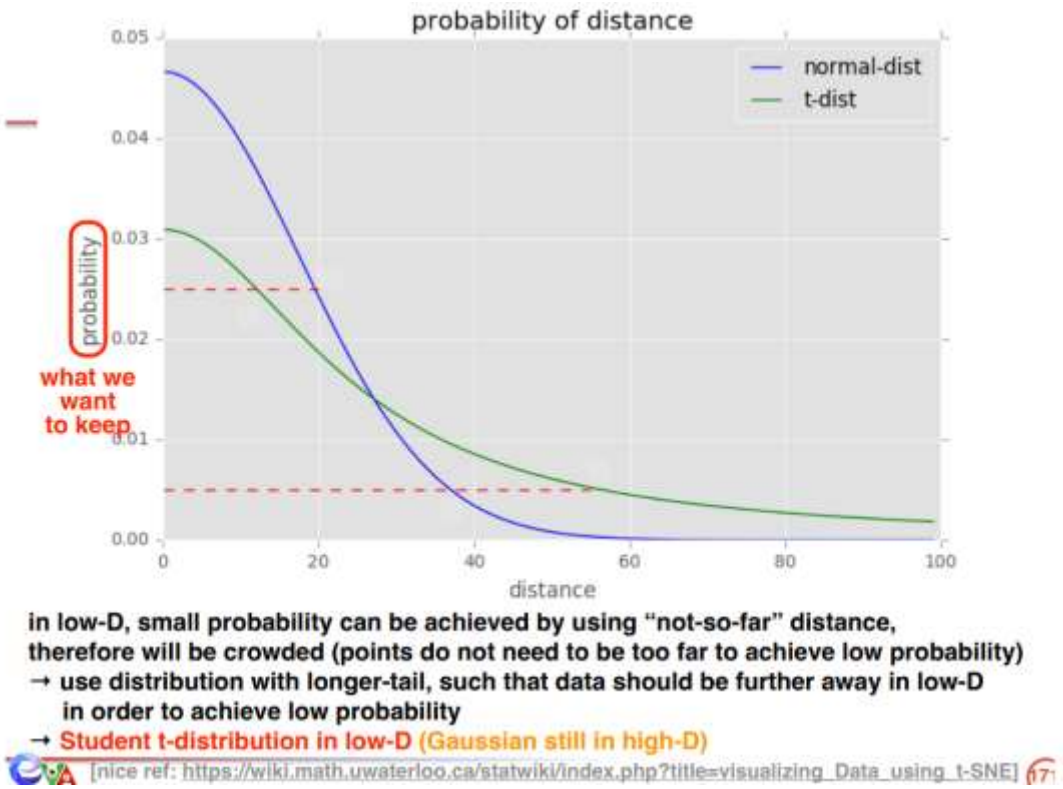
1. Choosing the appropriate kernel hyperparameters is crucial. Although I did not include the results of multiple hyperparameter tests for conciseness, it is evident that the performance of kernel PCA and kernel LDA can vary significantly under different hyperparameter settings. I have not used grid search to select the optimal hyperparameters, so the model may not perform at its best.
2. In the original LDA, the model already performs well, which suggests that the data is likely "linearly separable." In this case, introducing kernel methods for nonlinear mapping may have the opposite effect.
3. The size of the test set is too small, with only 30 images, which may lead to biased test results.

Finally, whether it's kernel PCA or kernel LDA, in most cases, we find that the performance of the RBF kernel is better than the other two kernels. Additionally, since the Linear kernel is just a special case of the Polynomial kernel, the overall performance is RBF kernel > Polynomial kernel > Linear kernel.

- **t-SNE**

➢ Part 1:

As described in the previous section, in the code, we only modify the calculation methods of the low-dimensional pairwise similarity Q and the gradient to implement symmetric SNE. The difference in results caused by these two formulas can be explained by the figure on p.171 of the Unsupervised_Learning.pdf:



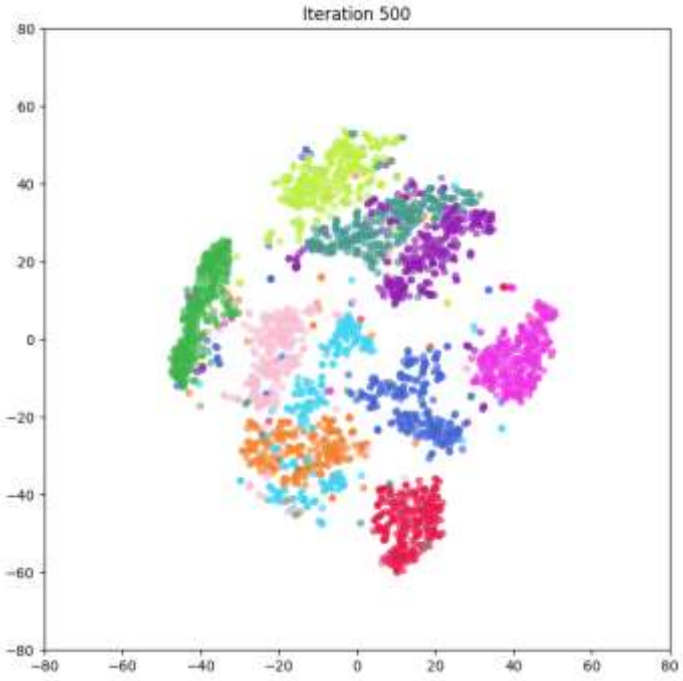

We can see that, under the same similarity, the distance between two points in the low-dimensional space is farther in t-SNE than symmetric SNE. This phenomenon becomes more pronounced as the similarity decreases. Generally, we prefer that the distance between two unrelated points (low similarity) be more significant. However, in the projection of symmetric SNE, there might be an issue where data points are too close to each other (the so-called "crowded problem"). Using t-SNE instead increases the distance between these data points, which helps alleviate the crowded problem.

In the reference code implementation, "Early Exaggeration" is a commonly used technique in t-SNE. During the early stages of the algorithm, it intentionally amplifies the differences in similarities to quickly separate the data points and prevent the local structure from becoming too crowded. Generally, this technique is more commonly used in t-SNE than in symmetric SNE. In implementing symmetric SNE for this assignment, I retained Early Exaggeration to keep the code simple. It is only briefly mentioned here for explanation purposes.

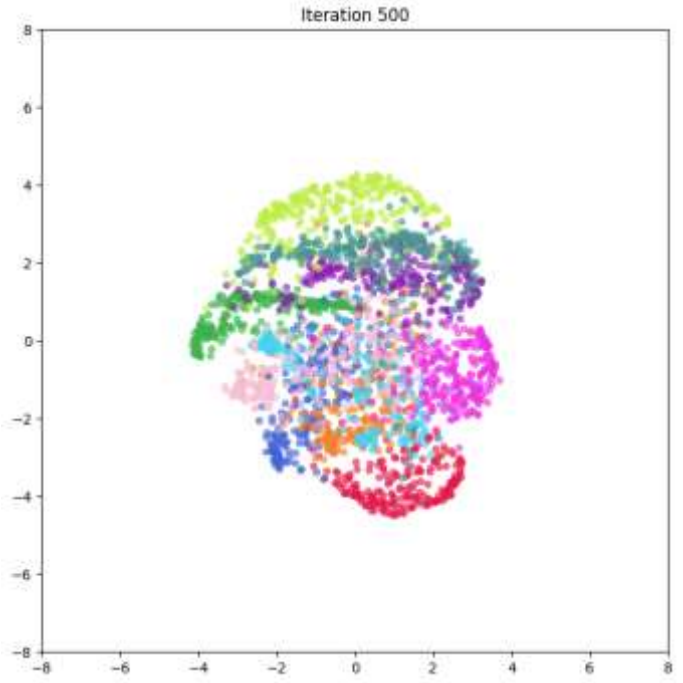
➢ Part 2:

The following are the projection results of t-SNE and symmetric SNE methods after 500 iterations. The path where the .gif file is saved is provided at the end of this report.

t-SNE: no_dims = 2, initial_dims = 50, perplexity = 50 ↓




Symmetric SNE: no_dims = 2, initial_dims = 50, perplexity = 50 ↓

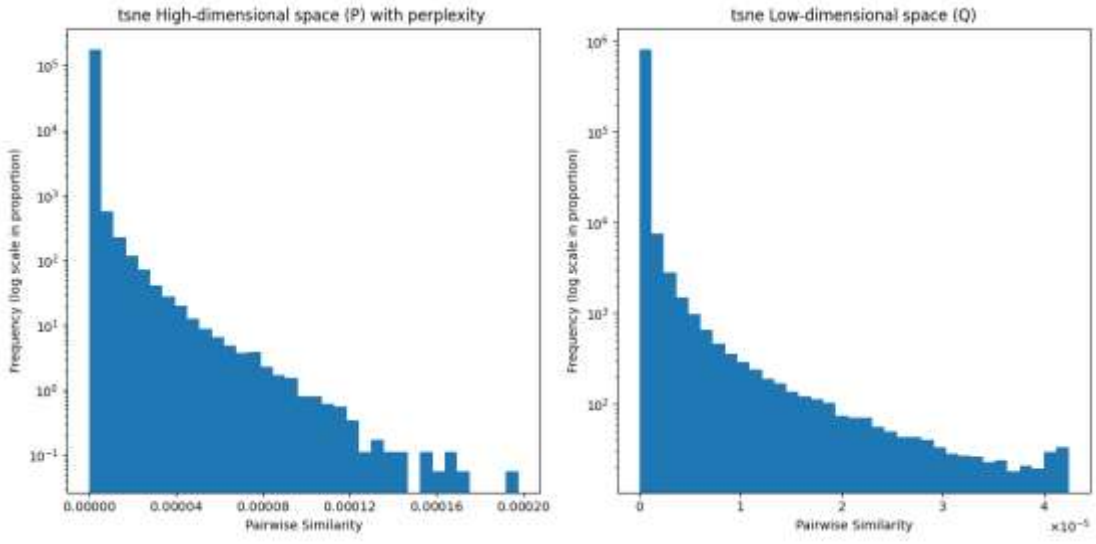


↑ From the above results, we can verify that t-SNE helps alleviate the crowded problem that symmetric SNE encounters.

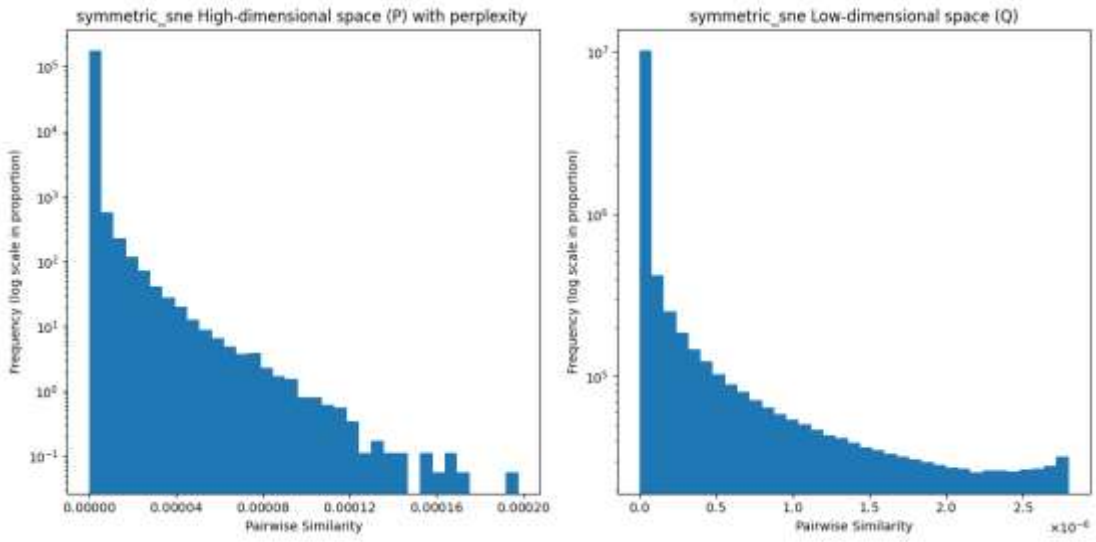
➢ Part 3:

The following figures are the distribution of pairwise similarities in both high- and low-dimensional space, based on both t-SNE and symmetric SNE of perplexity = 50. Since t-SNE and symmetric SNE use the same formula for calculating pairwise similarities in high-dimensional space, we can observe that the distribution of pairwise similarities plotted in the figure (left half) is identical. In the low-dimensional space, the tail of the distribution of pairwise similarities in t-SNE is longer. This can be observed by comparing the x-axis scales in t-SNE and symmetric SNE results. Due to this characteristic, the distribution of similarities between data points in the low-dimensional space becomes more spread out. This provides another perspective on how t-SNE can help alleviate the crowded problem.

t-SNE: no_dims = 2, initial_dims = 50, perplexity = 50 ↓



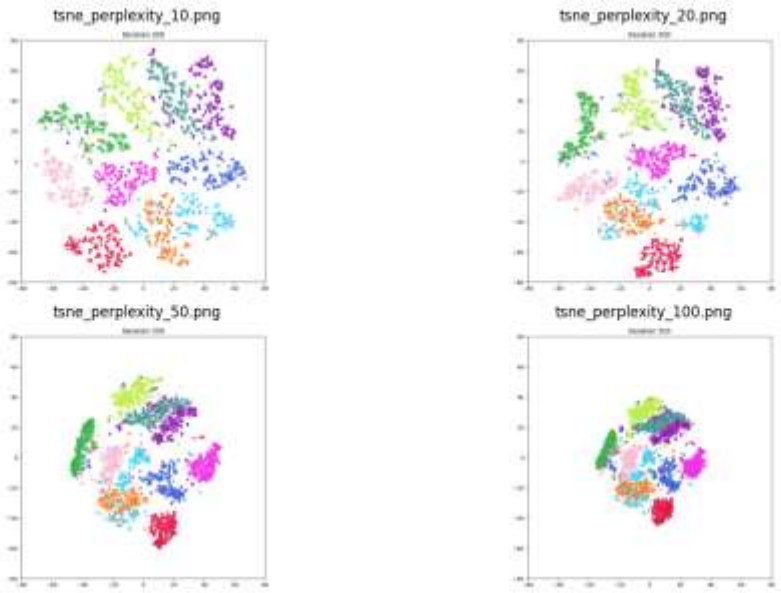

Symmetric SNE: no_dims = 2, initial_dims = 50, perplexity = 50 ↓



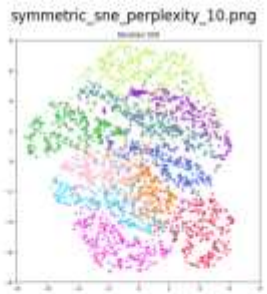

➢ Part 4:

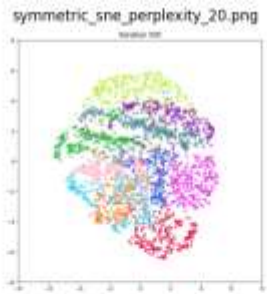
In this part, we must compare the effects of different perplexity values. The meaning of perplexity is a measure of the effective number of neighbors. The higher the perplexity value, the more neighbors each point has (i.e., the "effective neighborhood" becomes more extensive). I compared four different settings for perplexity: 10, 20, 50, and 100:

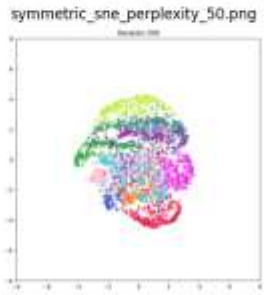
t-SNE: no_dims = 2, initial_dims = 50, perplexity = [10, 20, 50, 100] ↓

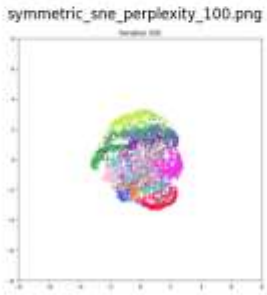



Symmetric SNE: no_dims = 2, initial_dims = 50, perplexity = [10, 20, 50, 100] ↓




↑ We can observe that the clustering effect becomes more pronounced as perplexity increases for both t-SNE and symmetric SNE. At the same time, the distance between different clusters decreases, making them more "crowded." On the other hand, when perplexity is too small, it may cause the points within the same cluster to be too dispersed, making the resulting low-dimensional representation challenging for further analysis. Choosing an appropriate perplexity is crucial. For example, in the figure above, t-SNE and symmetric SNE show promising results when perplexity = 50.

## 3. Observations and Discussion

1. Among PCA, LDA, kernel PCA, and kernel LDA, LDA has the slowest computation speed. The kernel method represents the similarity between samples using a matrix of shape (n_samples, n_samples). In the implementation of PCA, I used some techniques that allow the calculation of the eigenvectors of the covariance matrix by first computing a matrix of size (n_samples, n_samples). However, LDA requires calculating the inverse matrix and eigenvectors of a (n_features, n_features) matrix, which results in the most significant computational load.
2. In the current experimental setup, symmetric SNE requires around 400 epochs for

the projection results to no more extended change, while t-SNE has not yet converged even after 1000 epochs. This demonstrates that symmetric SNE converges more quickly than t-SNE.

3. Overall, in the first part of this assignment, we implemented PCA, LDA, kernel PCA, and kernel LDA methods and compared their performance in face recognition on the Yale Face Database. It was observed that the standard LDA performed the best without using a kernel, but it also required the most significant computational load (and was challenging to simplify). We can perform nonlinear projections using kernel methods and reduce the computational complexity (from calculating a **n_features** × **n_features** matrix to calculating a **n_samples** × **n_ samples** matrix). The downside, however, is that it requires setting appropriate hyperparameters, and it isn't easy to project eigenfaces or fisherfaces or perform reconstruction, making it challenging to verify whether the algorithm is implemented correctly. In the second part, we modified the t-SNE code to implement the symmetric SNE version and compared the results of both methods. This comparison validated how t-SNE addresses the "crowded problem" and allowed us to examine the effects of setting different perplexity values.

- **File Structure:**

pca.py

lda.py

kernel_pca.py

kernel_lda.py

tsne.py

eigenfaces (a folder)

- include the 25 eigenfaces

fisherfaces (a folder)

- include the 25 fisherfaces

pca_reconstructed_images (a folder)

- include the 10 reconstructed images

lda_reconstructed_images (a folder)

- include the 10 reconstructed images

Yale_Face_Database (a folder)

- given dataset

tsne_python (a folder)

- given dataset

tsne_visualization (a folder)

- include the visualization results in part 2 - part 4 for II. t-SNE