Homework 6 report

Machine Learning

Institute of Computer Science and Engineering

06/14/2024

Produced by :
PAULY Alexandre
alexandre.pauly@cy-tech.fr

Contents

1 Introduction

Context:

As part of the Machine Learning module, an assignment focusing on Kernel Eigenfaces using PCA and LDA and Stochastic Neighbor Embedding (SNE) has been initiated. This assignment aims to deepen understanding and practical implementation skills in these specific areas of machine learning. Both method are usefull for representing high-dimensional data in low-dimensional spaces while preserving underlying structure.

Objective:

In this report, we focus on understanding and implementing Kernel Eigenfaces from scratch and SNE using GitHub ressource. The main objective is to explain in detail each step of the implementation of these two techniques, including the underlying mathematical principles, computational procedures, and visualization techniques. Moreover, we will demonstrate the application of the Kernel Eigenfaces on a dataset of faces and SNE on a part of MNIST dataset.

2 Dataset

2.1 Kernel Eigenfaces

For Kernel eigenfaces, the Yale_Face_Database contains 165 images of 15 subjects (subject01, subject02, etc.). There are 11 images per subject, one for each of the following facial expressions or configurations: center-light, w/glasses, happy, left-light, w/no glasses, normal, right-light, sad, sleepy, surprised, and wink. These data are separated into training dataset (135 images) and testing dataset(30 images).

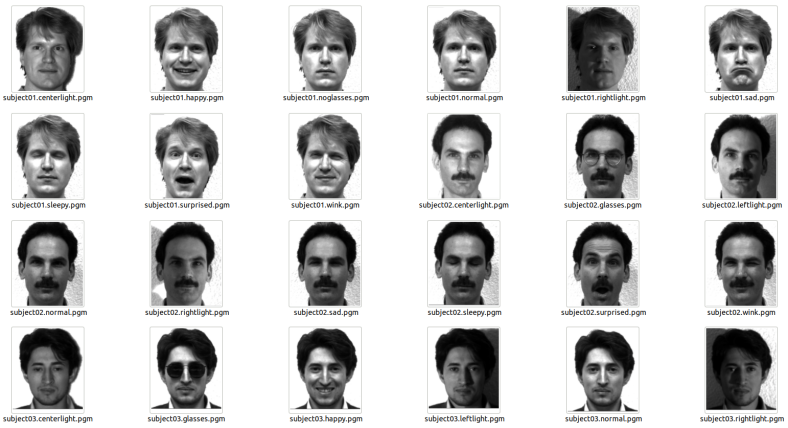


Figure 2: Yale face dataset

2.2 Stochastic Neighbor Embedding (SNE)

The data used in this study for t-SNE is derived from the MNIST dataset, which is extensively employed for handwritten character recognition and image classification tasks. This dataset comprises images of handwritten digits from zero to nine, each represented as a matrix of grayscale pixels. These images serve as digital representations of handwritten digits captured under various conditions, including diverse writing styles, different sizes, and varying levels of noise. This saple of mnist contains 2500 feature vectors with length 784, for describing 2500 mnist images.

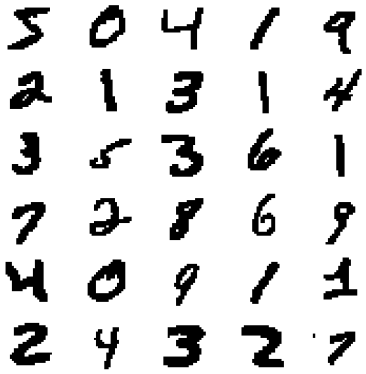


Figure 3: MNIST dataset

3 Implementation

3.1 Kernel Eigenfaces

PCA (Principal Component Analysis) and LDA (Linear Discriminant Analysis) are dimensionality reduction techniques used for visualizing and analyzing high-dimensional data in a lower-dimensional space while preserving data structure. PCA focuses on maximizing the variance between data points, thus capturing the most important directions of variation in the data. In contrast, LDA aims to maximize the separation between different classes by finding the axes that maximize the ratio of between-class variance to within-class variance. Kernel variants (kernel PCA and kernel LDA) extend these methods by capturing non-linear relationships in the data, offering increased flexibility to handle complex data structures.

Loading data

Before implementing this method, we start by loading data. To do this, we need to pay attention to the *pgm* format to extract correctly images, labels and filenames.

1st step : Pre-processing

To reduce the computational complexity during training and prediction time, we resize images from (231,195) to (77,65), a reduction in dimension by 3.

```
def pre_processing(data):
    # Initialization of variables
    num_imgs = data.shape[0]  # Number of images
    original_height, original_width = 231, 195  # Original
       dimensions
    compressed_height, compressed_width = 77, 65  # New
       dimensions
    img_compressed = np.zeros((num_imgs, compressed_height *
       compressed_width))  # List of compressed images

    # For each images
    for img in range(num_imgs):
        # For each rows
        for row in range(compressed_height):
            # For each columns
            for col in range(compressed_width):
                tmp = 0

                for i in range(3):
                    for j in range(3):
                        tmp += data[img][(row*3 + i) *
```

```
                                original_width + (col*3 + j)]
19
20                  img_compressed[img][row * compressed_width +
                       col] = tmp // 9
21
22      return img_compressed
```

Code 1 : *Pre-processing by resizing images*

3.1.1 Kernel-free approach

2nd step : Computing eigenvalues and eigenvectors

Without using kernel, both *PCA* and *LDA* functions start by computing eigenvalues and eigenvectors. But *PCA* need the covariance matrix on data :

$$\text{Covariance matrix:} \quad \Sigma = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T$$
$$\text{Eigenvalues and eigenvectors:} \quad \Sigma\mathbf{v} = \lambda\mathbf{v}$$

```
1  def PCA(data):
2      # Computing covariance matrix
3      covariance = np.cov(data.T)
4
5      # Computing eigenvalue and eigenvector using covariance
           matrix
6      eigenvalue, eigenvector = np.linalg.eigh(covariance)
7
8      W = eigenvector_processing(eigenvalue, eigenvector)
9
10     return W
```

Code 2 : *PCA method*

And *LDA* calculate diffusion matrix for intra and inter-classes as following :

$$\text{Diffusion matrix intra-class:} \quad S_W = \sum_{i=1}^{c}\sum_{\mathbf{x}\in X_i}(\mathbf{x} - \mathbf{m}_i)(\mathbf{x} - \mathbf{m}_i)^T$$
$$\text{Diffusion matrix inter-class:} \quad S_B = \sum_{i=1}^{c} N_i(\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^T$$
$$\text{Transformation matrix:} \quad S_W^{-1}S_B$$
$$\text{Eigenvectors:} \quad S_W^{-1}S_B\mathbf{w} = \lambda\mathbf{w}$$

```python
def LDA(data, label):
    # Initialization of variables
    dimension = data.shape[1]                # Dimension of data
    Sw = np.zeros((dimension, dimension))    # Dispersion matrix intra-class
    Sb = np.zeros((dimension, dimension))    # Dispersoion matrix inter-class
    mean = np.mean(data, axis=0)             # Mean vector
    nb_label = np.max(label)

    # For each label
    for i in range(nb_label):
        # Subject to extract
        id = np.where(label == i + 1)
        subject = data[id]

        # Mean vector for this subject
        mean_subject = np.mean(subject, axis=0)

        # Difference within class
        within_diff = subject - mean_subject
        Sw += within_diff.T @ within_diff

        # Difference without class
        between_diff = mean_subject - mean
        Sb += 9 * between_diff.T @ between_diff

    # Compute transformation matrix
    Sw_Sb = np.linalg.pinv(Sw) @ Sb
    eigenvalue, eigenvector = np.linalg.eigh(Sw_Sb)

    W = eigenvector_processing(eigenvalue, eigenvector)

    return W
```

Code 3 : *LDA method*

Both *PCA* and *LDA* keep first 25 largest eigenvectors and normalizes them. They are stored in the W variable for the following steps.

```python
def eigenvector_processing(eigenvalue, eigenvector):
    # Sorting eigenvalues
    index = np.argsort(-eigenvalue)
    eigenvector = eigenvector[:, index]

```

```
    # Keep 25 first eigenvector
    W = eigenvector[:, :25]

    # Eigen vectors normalization
    for i in range(W.shape[1]):
        W[:, i] = W[:, i] / np.linalg.norm(W[:, i])

    return W
```

Code 4 : *Eigenvectors selection*

3rd step : Plotting faces after reducing the dimension space

Before faces reconstruction, we plot faces after *PCA* or *LDA* to save them and also for the followings analysis.

```
def plot(matrix, method):
    fig = plt.figure(figsize=(5, 5))

    # For 25 first images
    for i in range(matrix.shape[1]):
        img = matrix[:, i].reshape(77, 65)
        ax = fig.add_subplot(5, 5, i + 1)
        ax.axis('off')
        ax.imshow(img, cmap='gray')

    # Saving plot
    fig.savefig(f'kernel_eigenfaces_results/{method}.jpg')

    plt.show()
```

Code 5 : *Plot faces after PCA or LDA*

4th step : Faces reconstruction

Now that *PCA* or *LDA* have been applied, we need to reconstruct 10 faces from the eigenvectors kept. To do this, we use xWW^{T} where W represent the 10 eigenvectors kept. This function show the true image and the reconstructed image to make a comparison.

```
def face_reconstruction(W, train_img, method):
    sample = np.random.choice(len(train_img), 10, replace=
       False)

    # Plot display
    fig = plt.figure(figsize=(8, 2))

    for i in range(10):
        img = train_img[sample[i]].reshape(77, 65)
```

```
        ax = fig.add_subplot(2, 10, i + 1)
        ax.axis('off')
        ax.imshow(img, cmap='gray')

        x = img.reshape(1, -1)

        # Face reconstruction
        reconstruct_img = x @ W @ W.T
        reconstruct_img = reconstruct_img.reshape(77, 65)
        ax = fig.add_subplot(2, 10, i + 11)
        ax.axis('off')
        ax.imshow(reconstruct_img, cmap='gray')

    # Saving reconstruction
    fig.savefig(f'kernel_eigenfaces_results/{method}
       _reconstruction.jpg')

    plt.show()
```

Code 6 : *Faces reconstruction*

5th step : Predictions

To make predictions we use KNN to calculte the nearest neighbor after projecting the data onto the eigenvectors calculated by *PCA* or *LDA*. For each test image, the euclidean distance is used between this image and eigenvectors projected to calculate the k neighbors. After identifying the nearest neighbours, the prediction is made by assigning the majority label among these neighbours to the current test image. If the prediction is incorrect, an error is recorded.

In more, this function make some iterations with different value of clusters (from 1 to 30) to generate a graph with the error value for a given number of cluster.

```
def prediction(train_img, train_label, test_img, test_label,
   W, output_name, kernel_bool=False, train_kernel=None,
   kernel_type=None):
    # Initialization of variables
    errors = []

    for k in range(1, 30):
        error = 0   # Error

        xW_train = train_img @ W
        xW_test = test_img @ W

        for i in range(len(test_img)):
```

```
            # Distance initialization
            distance = np.zeros(len(train_img))

            for j in range(len(train_img)):
                distance[j] = np.sum((xW_test[i] - xW_train[j
                    ]) ** 2)

            # Prediction by using the nearest neighbor
            neighbors = np.argsort(distance)[:k]
            prediction = np.argmax(np.bincount(train_label[
                neighbors]))

            if test_label[i] != prediction:
                error += 1

        errors.append(error / 30 * 100)

    # Plotting errors
    plot_errors(errors, output_name)
```

Code 7 : *Predictions and computing error*

3.1.2 Kernel approach

2nd step : Centering data

With a kernel, the process is very similar, but we need to center data in first.

```
# Centering data
mean = np.mean(train_img, axis=0)
centered_train = train_img - mean
centered_test = test_img - mean
```

Code 8 : *Centering data*

3rd step : Computing kernel

Now, this method requires the use of a kernel, we tested with 3 different kernels, all three previously used in **homework 5**. The first is a linear kernel, the second an RBF kernel and the last an addition of the linear kernel and RBF kernel.

```
def linear_kernel(u, v):
    return u @ v.T

def RBFkernel(u, v):
    gamma = 1e-10
    dist = np.sum(u ** 2, axis=1).reshape(-1, 1) + np.sum(v
        ** 2, axis=1) - 2 * u @ v.T
```

```
7
8       return np.exp(-gamma * dist)
9
10  def linear_and_RBFkernel(u, v):
11      return linear_kernel(u, v) + RBFkernel(u, v)
```

Code 9 : *Kernels*

3rd step : PCA and LDA kernel

PCA kernel implies to calculate a kernel matrix using a kernel as seen previously to transform data into a higher-dimensional feature space. Like *PCA*, this method should compute eigenvalues and eigenvectors and retain the first 25 eigenvectors and normalise them on the kernel matrix calculated.

```
1   def PCA_kernel(data, kernel_type):
2       # Centering data
3       mean = np.mean(data, axis=0)
4       centered_data = data - mean
5
6       # Computing kernel
7       kernel = kernel_type(centered_data, centered_data)
8
9       # Computing eigenvalue and eigenvector using covariance
            matrix
10      eigenvalue, eigenvector = np.linalg.eigh(kernel)
11
12      W = eigenvector_processing(eigenvalue, eigenvector)
13
14      return W, kernel
```

Code 10 : *PCA using kernel*

With *LDA*, we also use a kernel function to transform data into a higher-dimensional feature space. And like *LDA*, this method compute the inter and intra-class diffusion matrix to obtain the transformation matrix and compute eigenvalues and eigenvectors. An other time, *LDA with kernel* keep only the 25 first eigenvectors.

$$\text{Diffusion matrix intra-class:} \quad S_W = \sum_{i=1}^{c} \sum_{\mathbf{x} \in X_i} (\phi(x) - \mathbf{m}_i)(\phi(x) - \mathbf{m}_i)^T$$

$$\text{Diffusion matrix inter-class:} \quad S_B = \sum_{i=1}^{c} N_i (\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^T$$

$$\text{Transformation matrix:} \quad S_W^{-1} S_B \mathbf{w} = \lambda \mathbf{w}$$

```python
def LDA_kernel(data, kernel_type):
    # Centering data
    mean = np.mean(data, axis=0)
    centered_data = data - mean

    Z = np.ones((centered_data.shape[0], centered_data.shape
       [0])) / 9

    # Computing kernel
    kernel = kernel_type(centered_data, centered_data)

    # Computing dispersion matrix
    Sw = kernel @ kernel      # Dispersion matrix intra-class
    Sb = kernel @ Z @ kernel  # Dispersion matrix inter-class

    # Compute transformation matrix
    Sw_Sb = np.linalg.pinv(Sw) @ Sb
    eigenvalue, eigenvector = np.linalg.eigh(Sw_Sb)

    W = eigenvector_processing(eigenvalue, eigenvector)

    return W, kernel
```

Code 11 : *LDA using kernel*

4th step : Predictions

To make predictions, it's the same process as above, but we centre the data (training and test) with which we calculate a kernel function.

```python
def prediction(train_img, train_label, test_img, test_label,
   W, output_name, kernel_bool=False, train_kernel=None,
   kernel_type=None):
    # Initialization of variables
    errors = []

    for k in range(1, 30):
        error = 0   # Error

        # Centering data
        mean = np.mean(train_img, axis=0)
        centered_train = train_img - mean
        centered_test = test_img - mean

        # Computing kernel
        test_kernel = kernel_type(centered_test,
```

```
            centered_train)

        xW_train = train_kernel @ W
        xW_test = test_kernel @ W

        for i in range(len(test_img)):
            # Distance initialization
            distance = np.zeros(len(train_img))

            for j in range(len(train_img)):
                distance[j] = np.sum((xW_test[i] - xW_train[j
                    ]) ** 2)

            # Prediction by using the nearest neighbor
            neighbors = np.argsort(distance)[:k]
            prediction = np.argmax(np.bincount(train_label[
                neighbors]))

            if test_label[i] != prediction:
                error += 1

        errors.append(error / 30 * 100)

    # Plotting errors
    plot_errors(errors, output_name)
```

Code 12 : *Predictions and computing error*

3.2 Stochastic Neighbor Embedding (SNE)

SNE is also a non-linear dimensionality reduction technique commonly used for visualizing high-dimensional data in a lower-dimensional space while preserving structure. It operates by modeling the pairwise similarities between data points in both the high-dimensional and low-dimensional spaces, with a focus on preserving local neighborhood relationships.

Loading data

Before implementing t-SNE, we start by loading data. As explained is the previous section, the dataset is a part of MNIST dataset.

1st step : Initial dimension reduction with PCA

The first step of implementing SNE is to call the *pca* function to reduce the dimensional space from the original data. PCA centres the data and calculates the principal

components to finally returns the reduced data.

```
def pca(X=np.array([]), no_dims=50):
    print("Preprocessing the data using PCA...")
    (n, d) = X.shape
    X = X - np.tile(np.mean(X, 0), (n, 1))
    (l, M) = np.linalg.eig(np.dot(X.T, X))
    Y = np.dot(X, M[:, 0:no_dims])
    return Y
```

Code 13 : *PCA function*

2nd step : Compute conditional probabilities P

Now, we need to compute conditional probabilities P using *x2p* function. It based on the distance between points on reduced data. And *x2p* use *Hbeta* to adjust the values of P according to the specified perplexity via a binary search. *Hbeta* calculates the entropy and normalizes the values to obtain the conditional probabilities.

```
def Hbeta(D=np.array([]), beta=1.0):
    # Compute P-row and corresponding perplexity
    P = np.exp(-D.copy() * beta)
    sumP = sum(P)
    H = np.log(sumP) + beta * np.sum(D * P) / sumP
    P = P / sumP
    return H, P
```

Code 14 : *Hbeta function*

```
def x2p(X=np.array([]), tol=1e-5, perplexity=30.0):
    # Initialize some variables
    print("Computing pairwise distances...")
    (n, d) = X.shape
    sum_X = np.sum(np.square(X), 1)
    D = np.add(np.add(-2 * np.dot(X, X.T), sum_X).T, sum_X)
    P = np.zeros((n, n))
    beta = np.ones((n, 1))
    logU = np.log(perplexity)

    # Loop over all datapoints
    for i in range(n):

        # Print progress
        if i % 500 == 0:
            print("Computing P-values for point %d of %d..."
                % (i, n))

```

```
        # Compute the Gaussian kernel and entropy for the
          current precision
        betamin = -np.inf
        betamax = np.inf
        Di = D[i, np.concatenate((np.r_[0:i], np.r_[i+1:n]))]
        (H, thisP) = Hbeta(Di, beta[i])

        # Evaluate whether the perplexity is within tolerance
        Hdiff = H - logU
        tries = 0
        while np.abs(Hdiff) > tol and tries < 50:

            # If not, increase or decrease precision
            if Hdiff > 0:
                betamin = beta[i].copy()
                if betamax == np.inf or betamax == -np.inf:
                    beta[i] = beta[i] * 2.
                else:
                    beta[i] = (beta[i] + betamax) / 2.
            else:
                betamax = beta[i].copy()
                if betamin == np.inf or betamin == -np.inf:
                    beta[i] = beta[i] / 2.
                else:
                    beta[i] = (beta[i] + betamin) / 2.

            # Recompute the values
            (H, thisP) = Hbeta(Di, beta[i])
            Hdiff = H - logU
            tries += 1

        # Set the final row of P
        P[i, np.concatenate((np.r_[0:i], np.r_[i+1:n]))] =
          thisP

    # Return final P-matrix
    print("Mean value of sigma: %f" % np.mean(np.sqrt(1 /
      beta)))
    return P
```

Code 15 : *x2p function*

3rd step : SNE

Within *sne*, various variables are initialized, like the initial position of Y points by using a random generation and also gradients, gains and momentums.

For each iteration, new Q probabilities are computed. For *t-SNE*, Q is based on the Student distribution with a t degree of freedom. And for *Symmetric SNE*, Q is based on the exponential distribution.

$$\text{t-SNE:} \quad Q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|y_k - y_l\|^2)^{-1}}$$

$$\text{Symmetric SNE:} \quad Q_{ij} = \frac{e^{-\|y_i - y_j\|^2}}{\sum_{k \neq l} e^{-\|y_k - y_l\|^2}}$$

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

    # If we need to use t-sne method
    if method == 't-sne':
        num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
    # Else, if we need to use symetric sne method
    elif method == "symmetric-sne":
        num = np.exp(-np.add(np.add(num, sum_Y).T, sum_Y))
    else:
        print("Wrong method chosen !")
        break
    num[range(n), range(n)] = 0.
    Q = num / np.sum(num)
    Q = np.maximum(Q, 1e-12)
```

Code 16 : *Q computing*

After that, we need to calculate the gradient and update positions. Its calculted by the difference between P and Q. Also, gains and momentums are used to updated the position of Y of points to perform the convergence.

$$\text{t-SNE:} \quad \frac{\partial C}{\partial y_i} = 4 \sum_j \left((P_{ij} - Q_{ij}) \left(1 + \|y_i - y_j\|^2\right)^{-1} (y_i - y_j) \right)$$

$$\text{Symmetric SNE:} \quad \frac{\partial C}{\partial y_i} = 2 \sum_j (P_{ij} - Q_{ij})(y_i - y_j)$$

```
    # Compute gradient
    PQ = P - Q

    # If we need to use t-sne method
```

```
    if method == 't-sne':
        for j in range(n):
            dY[j, :] = np.sum(np.tile(PQ[:, j] * num[:, j], (
               no_dims, 1)).T * (Y[j, :] - Y), 0)
    # Else, if we need to use symetric sne method
    elif method == "symmetric-sne":
        for j in range(n):
            dY[j, :] = np.sum(np.tile(PQ[:, j], (no_dims, 1))
               .T * (Y[j, :] - Y), 0)
```

Code 17 : *Computing gradient and updating positions*

4th step : Visualization

Using this method, the classification was saved as a png for direct analysis in the following section. Each images generated highlights the different clusters detected in different colours. Also, a GIF is created to show the evolution of each iteration. The following Code, show how to save figures :

```
def save_figure(Y, labels, perplexity, method, init,
   name_file):
    if init:
        pylab.figure()
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)

    # If we need to use t-sne method
    if method == 't-sne':
        save_dir = f"sne_results/t-SNE/perplexity={int(
           perplexity)}"

        # Creating directory
        if not os.path.exists(save_dir):
            os.makedirs(save_dir)

        pylab.savefig(f'{save_dir}/{name_file}.png')
    # Else, if we need to use symetric sne method
    elif method == "symmetric-sne":

        save_dir = f"sne_results/symmetric_SNE/perplexity={
           int(perplexity)}"

        # Creating directory
        if not os.path.exists(save_dir):
            os.makedirs(save_dir)

        pylab.savefig(f'{save_dir}/{name_file}.png')
```

```
25      pylab.cla()
```

Code 18 : *Function to save classification every 50 epochs*

Also, *plot* help to generate P and Q distributions to analyse the similarity probabilities of the points. And, we can use histogram to plot pairwise similarities in both high-dimensional space and low-dimensional space.

```
1  def plot(P, Q, perplexity, method):
2      # If we need to use t-sne method
3      if method == 't-sne':
4          pylab.hist(P.flatten(),bins = 30,log = True)
5          pylab.savefig(f'sne_results/t-SNE/perplexity={int(
               perplexity)}/P.png')
6          pylab.cla()
7          pylab.hist(Q.flatten(),bins = 30,log = True)
8          pylab.savefig(f'sne_results/t-SNE/perplexity={int(
               perplexity)}/Q.png')
9      # Else, if we need to use symetric sne method
10     elif method == "symmetric-sne":
11         pylab.hist(P.flatten(),bins = 30,log = True)
12         pylab.savefig(f'sne_results/symmetric_SNE/perplexity
               ={int(perplexity)}/P.png')
13         pylab.cla()
14         pylab.hist(Q.flatten(),bins = 30,log = True)
15         pylab.savefig(f'sne_results/symmetric_SNE/perplexity
               ={int(perplexity)}/Q.png')
```

Code 19 : *Function to plot histogram of P and Q distributions*

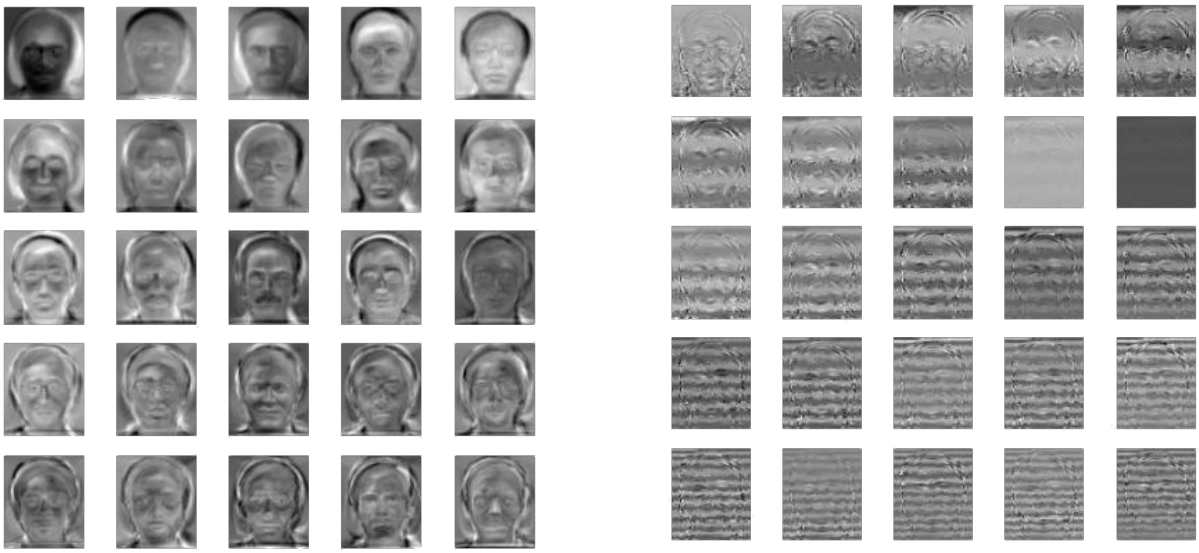
5$^{\text{th}}$ step : Experiments

t-SNE like *Symmetric SNE* are trained on differents value of the perplexity : *perplexity = [5, 15, 30, 50, 100]*. But, different results will be explain in the following part.

4 Experiments

4.1 Kernel Eigenfaces

When examining **Figure 4**, the results provided by PCA (**Subfigure a**) without using a kernel show quite interesting outcomes. The contours of the faces are easily distinguishable, in contrast to the LDA method where they are largely too blurred. The first few faces are discernible, but for most of them, we can hardly see anything. This is likely due to the dimensionality reduction being too small relative to the number of classes. In this study, we have 15 classes/emotions per face and the preserved dimensionality space is 25. This is why the fisherfaces are completely blurred/grey.

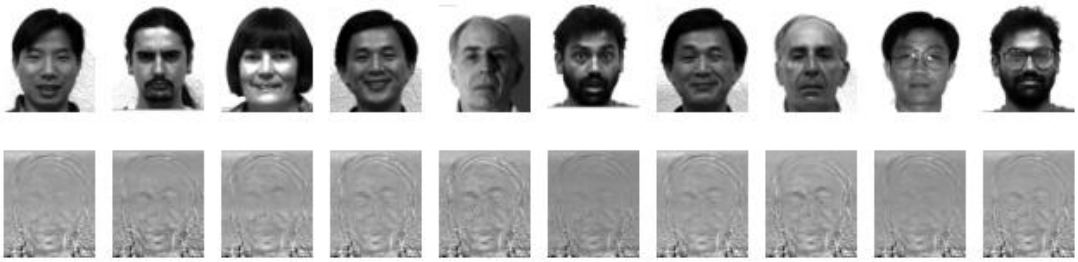


(a) PCA : Eigenfaces **(b)** LDA : Fisherfaces

Figure 4: Comparison of PCA and LDA after reduction dimension

In terms of reconstruction (**Figure 5**), the solution provided by PCA is quite interesting and highlights certain faces and features that are easily distinguishable. On the other hand, LDA only reconstructs the contours of the faces but does not differentiate between the faces and generates only one.

(a) PCA : Eigenfaces



(b) LDA : Fisherfaces

Figure 5: Comparison of PCA and LDA after reconstruction

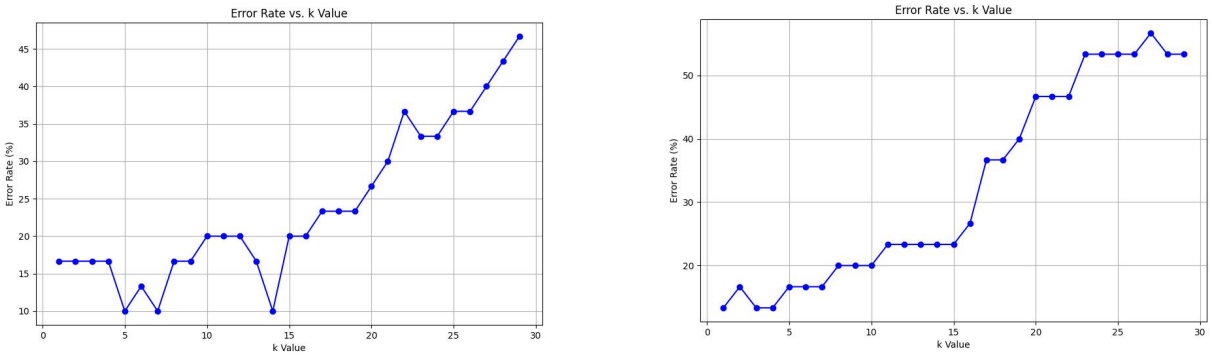
Error between PCA and LDA

As discussed in **Section 3**, different values of k for k-NN were tested for k = [1:30]. This allowed us to obtain a graph highlighting the most interesting number of clusters to minimize the error. As expected, the error increased as the value of k increased. However, PCA shows that it is possible to obtain the same error with 5, 7, or 14 clusters, whereas LDA minimizes the error for 1, 3, and 4 clusters. Both PCA and LDA achieve a minimum error of 10, but for different numbers of clusters. To compare them while minimizing the error for each, we could take k = 5 for an error of 10 for PCA and 16.67 for LDA.

Table 1: Comparison of mean error on k=[1:30]

Method	**Mean error**
PCA	24.25
LDA	32.41

As shown in **Table 1**, PCA has a significantly lower error, making it more performant.



(a) PCA **(b)** LDA

Figure 6: Comparison of error between PCA and LDA for different k values

Error between PCA and LDA with kernels

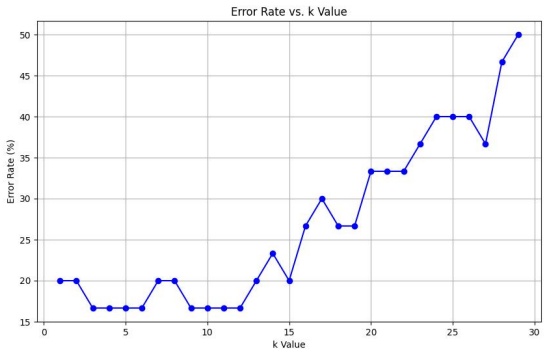
By testing different kernel configurations for PCA and LDA, we obtain varied results. Firstly, using a linear kernel and linear + RBF on PCA leads to the same result.

In all cases, the error increases as k increases except for LDA with an RBF kernel, where it is lower with k=30.

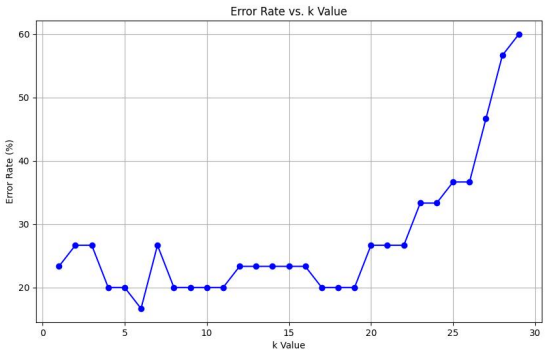
Overall, PCA achieves better results and minimizes the error for many k values (k=3,4,5,6,9,10,11,12) to 16.67 for a linear kernel or using both linear and RBF kernels simultaneously. Meanwhile, LDA minimizes the error for k=6 with a linear kernel at 16.67 error and k=3,4 for a 16.67 error.

Table 2: Comparison of mean error on k=[1:30]

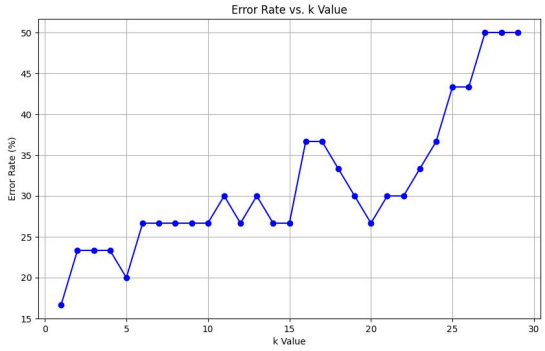
Method	**Kernel**	**Mean error**
PCA	Linear	26.78
PCA	RBF	31.38
PCA	Linear + RBF	26.78
LDA	Linear	27.59
LDA	RBF	40.23
LDA	Linear + RBF	31.49



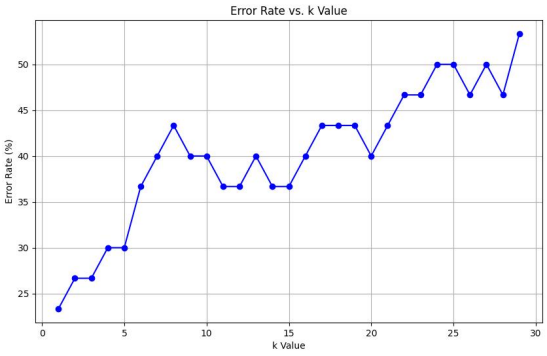
(a) PCA with linear kernel



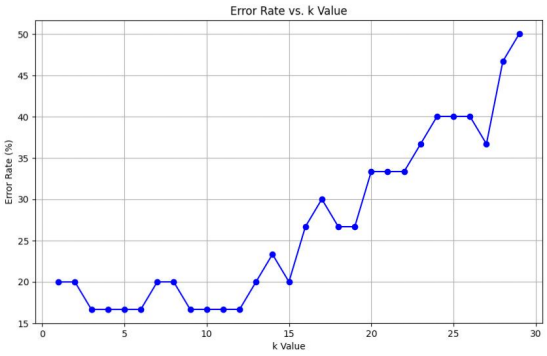
(b) LDA with linear kernel



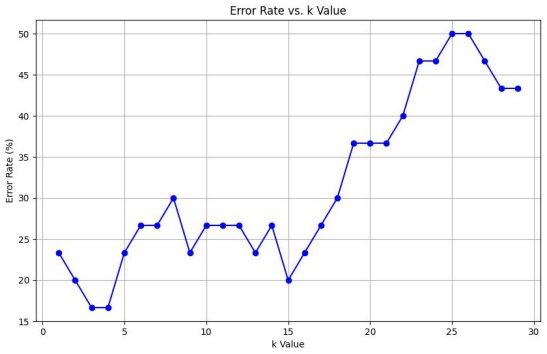
(c) PCA with RBF kernel



(d) LDA with RBF kernel



(e) PCA with linear and RBF kernel



(f) LDA with linear and RBF kernel

Figure 7: Comparison of error between PCA and LDA with kernel for different k values

In conclusion, PCA achieves better results than LDA, likely because the chosen dimensionality reduction is not adequate. However, the obtained results are more interesting without a kernel, indicating that increasing the dimensional space is not useful.

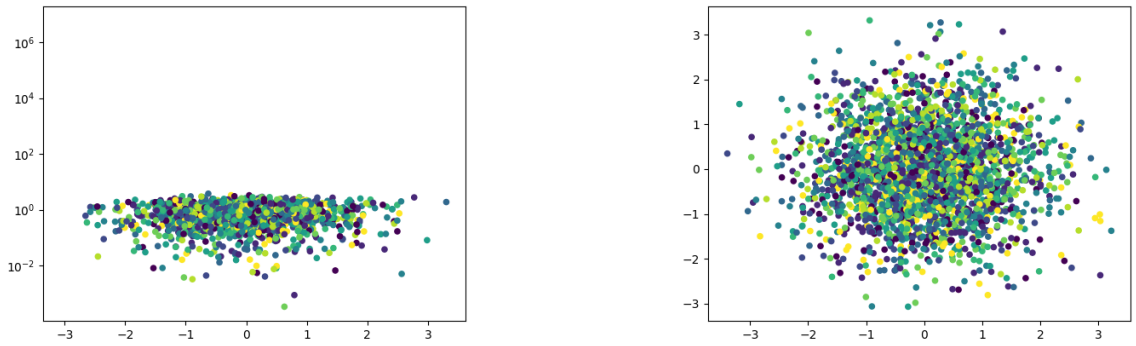
4.2 Stochastic Neighbor Embedding (SNE)

In this study, all results were tested with the following hyper-parameters :

- max_iter = 1000
- initial_momentum = 0.5

- final_momentum = 0.8
- eta = 500
- min_gain = 0.01

From the initialization, we can see that the distribution between a symmetric application with a normal distribution and an initialization using the Student's t-distribution changes significantly. With a Student's t-distribution, the classes will be much more dispersed compared to a symmetric distribution.

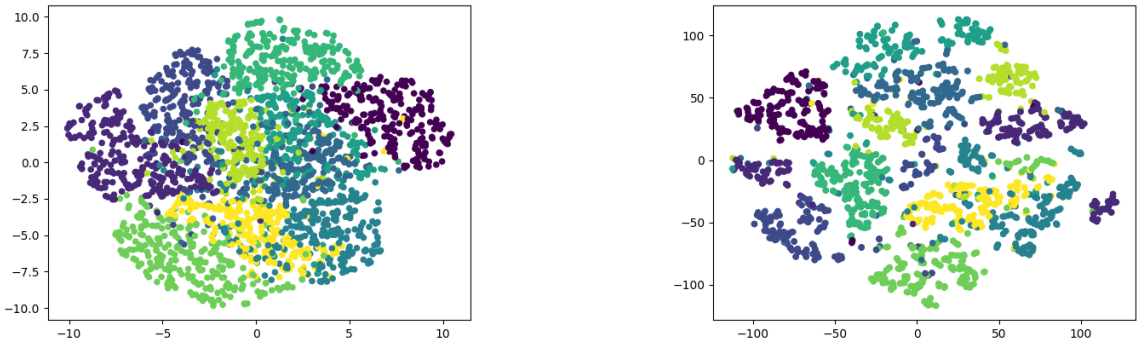


(a) SNE with perplexity=5 **(b)** t-SNE with perplexity=5

Figure 8: Comparison of Symmetric SNE and t-SNE on initialization

This difference greatly influences the results. Where SNE will be more clustered/overcrowded, we have the opposite with t-SNE, which tends to further separate the clusters.

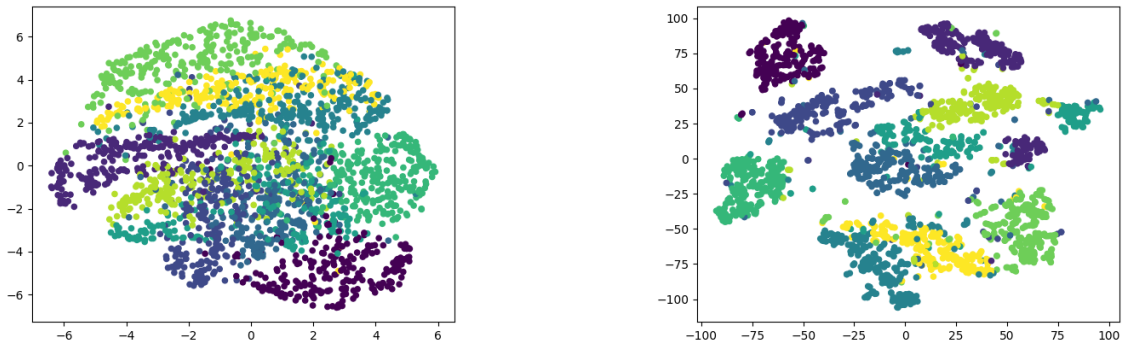
Both methods were set with the same number of hyper-parameters, including the number of iterations and perplexity (a perplexity is the number of close neighbors for each point). Typical values for perplexity are 5, 15, 30, 50, and 100. All tests converged for 1000 epochs as shown in the gif format in *sne_results* folder of the project.



(a) SNE with perplexity=5 **(b)** t-SNE with perplexity=5

Figure 9: Comparison of Symmetric SNE and t-SNE with perplexity=5 on 1000 epochs

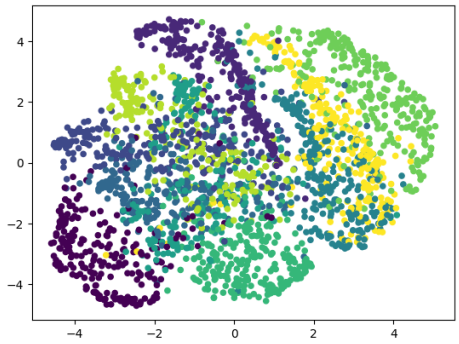
When the perplexity is set to 5, the results of SNE are very clustered, but the classes remain distinguishable, especially for those that are more spread out. This is not an issue for t-SNE, which tends to separate clusters more. However, for both methods, some clusters, especially those located in the center, tend to merge with others and are more dispersed.



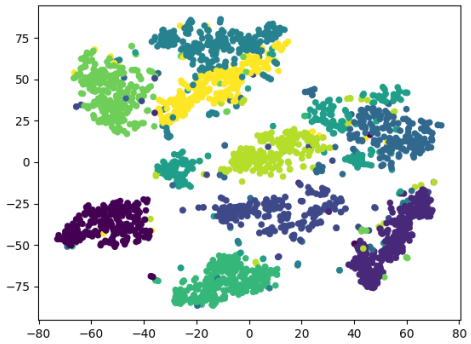
(a) SNE with perplexity=15 **(b)** t-SNE with perplexity=15

Figure 10: Comparison of Symmetric SNE and t-SNE with perplexity=15 on 1000 epochs

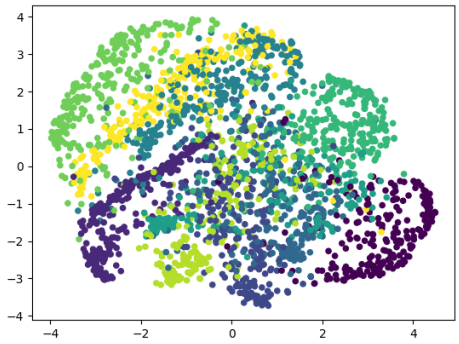
As the perplexity increases (**Figure 11**), we can already see that the clusters are more dispersed for SNE, whereas they are more grouped for t-SNE.



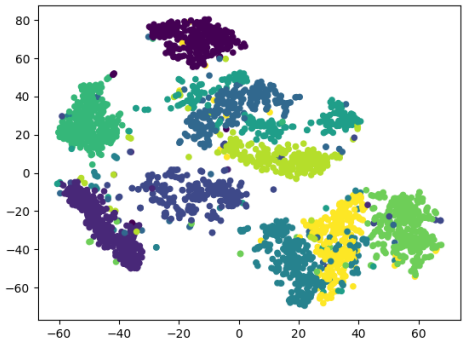
(a) SNE with perplexity=30



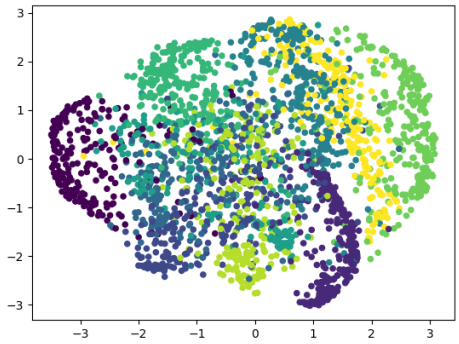
(b) t-SNE with perplexity=30



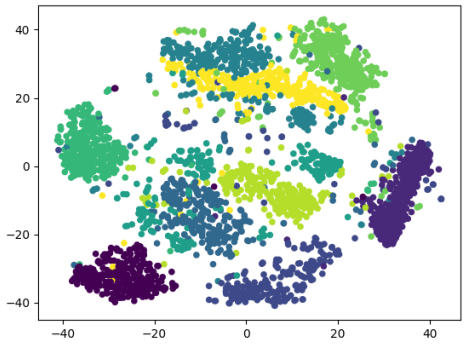
(c) SNE with perplexity=50



(d) t-SNE with perplexity=50



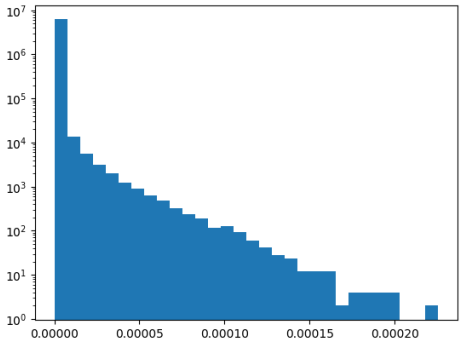
(e) SNE with perplexity=100



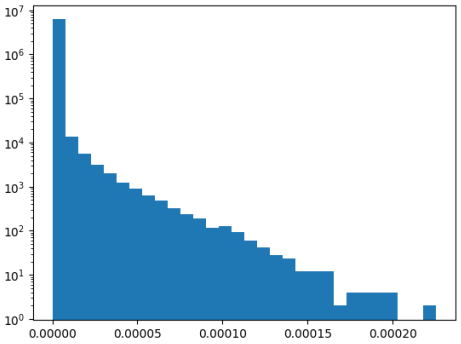
(f) t-SNE with perplexity=100

Figure 11: Comparison of Symmetric SNE and t-SNE with perplexity=30, 50 and 100 on 1000 epochs

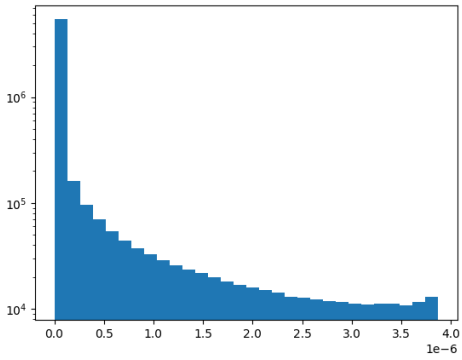
From the figure below (**Figure 12**), both methods have similar pairwise similarities with a perplexity of 30 in high-dimensional and low-dimensional spaces. The difference will be seen if we reduce the perplexity.



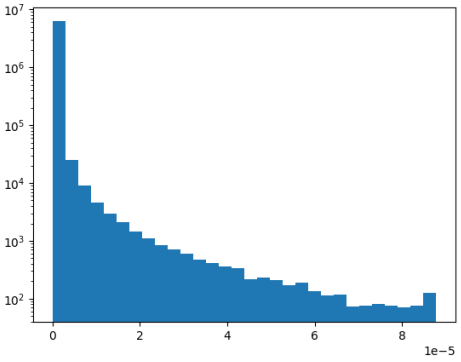
(a) SNE high-dimension



(b) t-SNE high-dimension



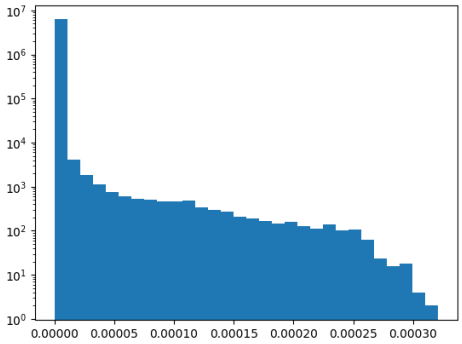
(c) SNE low-dimension



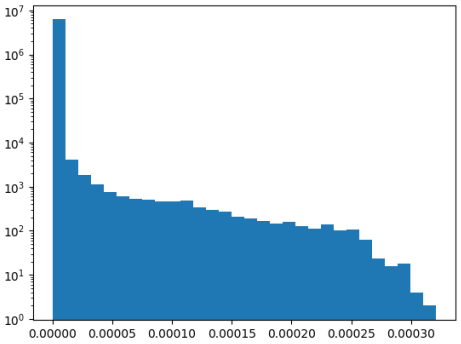
(d) t-SNE low-dimension

Figure 12: Comparison of Symmetric SNE and t-SNE with perplexity=30 on 1000 epochs

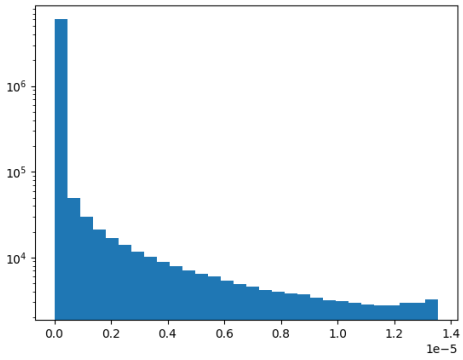
When the perplexity is set to 5 (**Figure 13**), we can see that the difference in similarities becomes higher. This makes sense since with lower perplexity, the projection will be less clearly separated between each class in the low-dimensional space, and the similarities will also be lower.



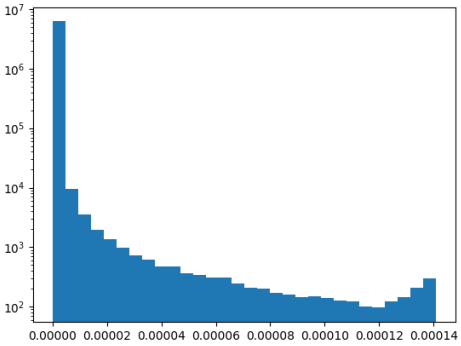
(a) SNE high-dimension



(b) t-SNE high-dimension



(c) SNE low-dimension



(d) t-SNE low-dimension

Figure 13: Comparison of Symmetric SNE and t-SNE with perplexity=5 on 1000 epochs

Finally, t-SNE is a better method for visualizing dimensionality reduction compared to symmetric SNE, which can create greater separation among the data classes.

5 Observations and Discussion

5.1 Kernel Eigenfaces

Eigenfaces, derived from PCA, are used for facial recognition by reducing the dimensionality of image data while preserving essential features. Dimension reduction methods such as PCA and LDA each have their advantages and disadvantages.

PCA is an unsupervised method that aims to find the directions of maximum variance in the data, while LDA is a supervised method that aims to find the projection that best separates the classes in the data.

The addition of kernels enhances these methods by transforming the data into a higher-dimensional space, which could help to better separate non-linear data. However, this can also increase computational complexity and does not necessarily guarantee improved results, as observed in our study.

In summary, the choice of dimension reduction method strongly depends on the specific characteristics of the data and the objectives of the analysis. PCA without a kernel showed better overall performance in our project, probably due to the structure and volume of the data analyzed.

5.2 Stochastic Neighbor Embedding (SNE)

1. The visualization of optimization using animations is better for visualizing the movement of data projected through videos. For this reason, it is possible to visualize GIFs for different perplexity values.

2. SNE, like t-SNE, has the same limitation: they cannot project new data using the formed data, which requires applying the method from the beginning. This is unlike PCA, which allows obtaining the projection matrix W after forming a dataset, which can then be used to project new data.

3. SNE starts by projecting high-dimensional data into low-dimensional space in a way that resembles the high-dimensional space as closely as possible. However, due to the use of Gaussian distribution to obtain the conditional probability in the low-dimensional space, the projection still suffers from an overcrowding problem.

4. Compared to the previous point, there is a significant difference between t-SNE and SNE in terms of probability scaling in low-dimensional space because t-SNE uses the t-distribution to calculate the conditional probability in this space. As a result, the projection of the data is more spread out, as indicated by the longer length of the t-SNE graph in the range of data projection, making the data representation clearer in terms of separation.

5. The greater the perplexity, the faster the pairwise similarities converge.

6. Symmetric SNE converges more quickly than t-SNE.

7. Perplexity has a greater effect on t-SNE than on symmetric SNE because symmetric SNE has a crowdedness problem.

6 Conclusion

In this study, we explored and compared different dimension reduction methods to make an analysis and visualize data by using PCA and LDA with kernel etension and also SNE and t-SNE. Both on different case with 2 different dataset.

The main points are as follows :

- PCA is a method to reduce the dimensional space by preserving important informations on data and eigenfaces are efficient for facial recognition.
- LDA is a supervised method optimizing separation between class. It's very useful for classification tasks.
- Stochastic neighbourhood methods, in particular t-SNE, are excellent for visualising data by reducing dimension while maintaining the local structure of the data. t-SNE has shown better cluster separation than SNE.
- Perplexity is very important in the results obtained by SNE and t-SNE. Different values of perplexity can lead to different distributions of data in the reduced space.

In conclusion, the choice of dimension reduction method depends strongly on the characteristics of the data and the specific objectives of the analysis. In our study, PCA and t-SNE showed robust performance for dimension reduction and data visualisation. The results highlight the importance of proper selection of methods and hyper-parameters to obtain optimal results.

References

[1] Vungarala, S. K. (2023, april 30). PCA vs LDA—No more confusion ! Medium, https://medium.com/@seshu8hachi/pca-vs-lda-no-more-confusion-fc21fb8d06e9.

[2] Tandia, M. F. (2020, july 27). Visualization Method : SNE vs t-SNE. LinkedIn : Log In or Sign Up, https://www.linkedin.com/pulse/visualization-method-sne-vs-t-sne-implementation-using-tandia.

[3] Ming-Hsuan Yang. (2002). Kernel Eigenfaces vs. Kernel Fisherfaces: Face Recognition Using Kernel Methods, https://faculty.ucmerced.edu/mhyang/papers/fg02.pdf.

[4] t-SNE. (s. d.), Laurens van der Maaten. https://lvdmaaten.github.io/tsne/.