\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Contents and structure:**

**0. Preparations**

**1. Code with detailed explanations**

* Kernel Eigenfaces
* Part 1
* Part 2
* Part 3
* t-SNE
* Part 1
* Part 2
* Part 3
* Part 4

**2. Experiments and Discussion**

* Kernel Eigenfaces
* Part 1
* Part 2
* Part 3
* t-SNE
* Part 1
* Part 2
* Part 3
* Part 4

**3.Observations and Discussion**

* Summary
* Anything

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Homework #07 Objective:**

1. *Do face recognition using eigenface and fisherface, with and without kernel methods.*
2. *Understand the given t-SNE implementation in python & transform to symmetric SNE.*

**0. Preparations** **(The code is in python programming language)**

First step is to mount the drive from where I will access all the files for the code:

0001.

from google.colab import drive

drive.mount("/content/drive")

Then I added a custom print function, preprint(), to save the outputs of print() in a txt file:

0002.

# Define a custom "pseudo personal print" function

def perprint(message, end = None, file = output\_file):

    print(message, end = end)

    print(message, end = end, file = file)

**1. Code with detailed explanations**

**Note: Important pieces are also explained directly by commenting inside the code for better clarity.**

* **Kernel Eigenfaces**

: This part in the notebook is the concerned section.

I will first explain the parts and functions that support part 1, 2 and 3 (common functions).

I begin with setting up the paths for accessing the Yale Face Face Database’s training and testing folders. I also set up the output directory where we will save the images generated later.

Here is the corresponding code below.

0003

import os

""" I. Kernel Eigenfaces / Fisherfaces """

# Variables for input directories

training\_folder = '/content/drive/MyDrive/…/Yale\_Face\_Database/Training'

testing\_folder = '/content/drive/MyDrive/…/Testing'

# Variable for output directory

output\_folder = '/content/drive/MyDrive/…/HMW\_07/Kernel\_Eigenfaces\_Fisherfaces'

# Ensure output directory exists

os.makedirs(output\_folder, exist\_ok=True)

Next, we will import all the required packages for computing eigenfaces and fisherfaces:

**os**: For interacting with the operating system for file and directory manipulation.

**time**: For measuring durations of certain parts of the code.

**random**: For generating random numbers.

**numpy**: For matrix manipulations and mathematical operations.

**matplotlib.pyplot**: For plotting, displaying and saving figures.

**PIL.Image** For reading images from the input directories.

**scipy.spatial.distance.cdist**: For distance calculation between pairs of vectors.

0004

import os, time

import random

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

from scipy.spatial.distance import cdist

**The first function** created is the read\_dataset() function that will read and preprocess the input images and labels for training and testing. Here is how the function works:

* Reads a dataset of images, resizes them with high quality downsampling, and flattens them into vectors.

It also extracts the labels from the images at position 7:9, location of the labels(integers).

* Input parameters:
* directory (str): The path to the directory containing the images.
* SHAPE (tuple): The desired shape (width, height) for resizing the images.
* Returns:
* tuple: A tuple containing the flattened image data and their labels.

0005

def read\_dataset(directory, SHAPE):

    files = os.listdir(directory)

    images = np.zeros((len(files), SHAPE[0]\*SHAPE[1]), dtype=np.float32)

    labels = np.zeros(len(files), dtype=int)

    ''' Process each file in the directory '''

    for i, file in enumerate(files):

        with Image.open(os.path.join(directory, file)) as image:

            ''' Resize the image with high-quality downsampling '''

            image = image.resize(SHAPE, Image.LANCZOS)

            ''' Flatten the image and store it in the array '''

            images[i] = np.asarray(image, dtype=np.float32).flatten()

            ''' Extract label from the filename and store it '''

            labels[i] = int(file[7:9])

    return images, labels

**This next function** is required for all the rest of this section.

The show\_save\_faces() function will be converting the computed eigenfaces and fisherfaces into images, display those images and save them. Here is its description:

* Convert into a grid of images, display, and optionally save the figure.
* Input parameters:
* faces: List of images (2D arrays: resized eigenfaces and fisherfaces) to be displayed.
* rows: Number of rows in the grid layout.
* title: Title of the figure.
* save\_dir: Directory where the figure should be saved (optional).
* Returns:
* None

0006

def show\_save\_faces(faces, rows, title, save\_dir=None):

    plt.figure(title)

    num\_faces = len(faces)

    cols = int(np.ceil(num\_faces / rows))

    for idx, face in enumerate(faces):

        plt.subplot(rows, cols, idx + 1)

        plt.axis('off')

        plt.imshow(face, cmap='gray')

    if save\_dir:

        file\_path = os.path.join(save\_dir, f'{title}.png')

        plt.savefig(file\_path, bbox\_inches='tight', pad\_inches=0)

    plt.show()

    return

**This next function is also** required for all the rest of this section on eigenfaces and fisherfaces.

The face\_recognition() function will do Face Recognition using k-Nearest Neighbors (KNN):

* This function performs face recognition using the k-Nearest Neighbors (KNN) algorithm. It also calculates the accuracy of the recognition process and prints the result.
* Input parameters:
* train\_label (numpy.ndarray): Labels of the training samples.
* train\_projections (numpy.ndarray): Feature weights (representations) of the training samples.
* test\_label (numpy.ndarray): Labels of the test samples.
* test\_projections (numpy.ndarray): Feature weights (representations) of the test samples.
* k (int, optional): Number of nearest neighbors to consider in KNN. Default is 5.
* title (str, optional): Title for result display. Default is an empty string.
* Returns:
* accuracy (int)

Below in the code you can see the main steps in comments.

0007

def face\_recognition(train\_labels, train\_projections, test\_labels, test\_projections, k=5, title=''):

    error = 0

    dist = np.zeros(len(train\_projections))

    for i in range(len(test\_projections)):

        ''' Calculate squared Euclidean distance between test[i] and train samples '''

        dist = cdist([test\_projections[i]], train\_projections, 'sqeuclidean')

        ''' Find the indices of the k=5 nearest neighbors in the training set and predict '''

        k\_nearest = np.argsort(dist)[0][0:k]

        predict = np.argmax(np.bincount(train\_labels[k\_nearest]))

        ''' Check if the predicted label does not match the true label of the test sample '''

        if test\_labels[i] != predict:

            error += 1

    ''' Calculate and print the accuracy '''

    accuracy = (len(test\_labels) - error) / len(test\_labels)

    perprint(f"K= {k}, {title}Accuracy: {accuracy\*100:.2f} % ")

    return accuracy

* **Part1: 25 eigenfaces & fisherfaces + reconstruction of 10 random images**

In Part 1 we need to do PCA and LDA. So, I built 2 functions for each of them.

**The PCA() function** for Principal Component Analysis (PCA) will be used to get eigenfaces, which are the transpose of the top 25 eigenvectors from the decomposition of the covariance matrix of the training data. The eigenfaces will be used for image reconstruction.

* This function performs PCA for getting the eigenfaces using training datasets of face images. It involves computing eigenfaces (principal components) and using them for dimensionality reduction and then face reconstruction.
* Parameters:
* train\_images: A numpy 2D array containing the training face images.
* test\_images: A numpy 2D array containing the test face images.
* train\_labels: A numpy array of labels corresponding to the training images.
* test\_labels: A numpy array of labels corresponding to the test images.
* SHAPE: A tuple representing the dimensions (width, height) of the face images.
* chosen\_index: Indices for selecting random 10 test images for reconstruction.
* output\_folder: The path to the folder for saving intermediate results (eigenvalues, eigenvectors, visualizations).
* Workflow:
* Average Face Calculation: Computes the mean face from the training dataset.
* Data Centering: Subtracts the average face from each training and testing image.
* Covariance Matrix Computation: Calculates the covariance matrix of the centered training data. The formula is a follow:

A white text on a black background

Description automatically generated

Where X is the train images matrix and Xˉ is its mean. N is the number of training images = 135.

The covariance matrix is MxM matrix where M is the number of features (pixels) in each original data point.

* Eigenvalue Decomposition: Obtains eigenvalues and eigenvectors (eigenfaces) from the covariance matrix.
* Eigenface Selection: Selects the top 25 eigenfaces based on eigenvalues.
* Eigenface Visualization: Displays and saves the eigenfaces with show\_save\_faces() in code 0006.

*For the 25 eigenfaces , we simply resize each vector in W to SHAPE[1]xSHAPE[0] (width, height).*

* Image Reconstruction: Reconstructs test images after projecting them to lower dimension with Eigenfaces, still with eigenfaces, and visualizing the results of reconstruction after PCA with the show\_save\_faces() function.

*For the reconstruction, we project each projected image (z) back to the original space and visualize the result.*

A black background with white text

Description automatically generated

* Returns:
* None: **I will explain the rest of the function in Part 2, in code 0010.** (Following the structure of the report)

0008

def PCA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index,output\_folder):

    avg\_face = np.mean(train\_images, axis=0)

    centered\_train\_faces = train\_images - avg\_face

    # ... Portion left out of the report to make the function shorter

    else:

        print(f"\nDidn't find eigenvalues and eigenvectors for PCA. \nComputing ... \n")

        Cov\_matrix = np.dot(centered\_train\_faces.T, centered\_train\_faces) / (len(train\_images) - 1)

        eigen\_values, eigen\_vectors = np.linalg.eig(Cov\_matrix)

        # ... Portion left out of the report to make the function shorter

    ''' Projection matrix (W from class): '''

    sort\_index = np.argsort(-eigen\_values)

    top\_eigen\_vectors = eigen\_vectors[:, sort\_index[0:25]].real # Projection matrix W

    ''' Here we have our eigenfaces, 25 in total, 1 on each row'''

    eigenfaces = top\_eigen\_vectors.T # Shape = (25, SHAPE[0]\*SHAPE[1])

    ''' Visualize the eigenfaces (reshape and display) '''

    show\_save\_faces(eigenfaces.reshape(25, SHAPE[1], SHAPE[0]), 5, 'PCA eigenfaces', output\_folder)

    ''' Reconstruct the selected test images using the eigenfaces '''

    test\_images\_centered = test\_images[chosen\_index] - avg\_face

    projections = np.dot(test\_images\_centered, eigenfaces.T) # AKA projected centered data

    reconstruction\_faces = avg\_face + np.dot(projections, eigenfaces)

    ''' Visualize the reconstructed faces'''

    show\_save\_faces(reconstruction\_faces.reshape(10, SHAPE[1], SHAPE[0]), 2,

'PCA reconstructed faces', output\_folder)

**The LDA() function** for Linear Discriminant Analysis (LDA) will be used to get fisherfaces, which are the transpose of the top 25 eigenvectors from the decomposition of the ratio of the between-class matrix and within-class matrix, obtained from the labeled training data.

LDA is a supervised dimensionality reduction technique.

The fisherfaces will be used for image reconstruction.

* This function performs LDA for getting the fisherfaces using the labeled training dataset. It involves steps from computing average faces and applying dimensionality reduction through PCA before applying LDA, which helps in handling the high dimensionality of face images.
* Parameters:
  + train\_images (numpy.ndarray): Array of training face images.
  + test\_images (numpy.ndarray): Array of test face images.
  + train\_labels (numpy.ndarray): Labels corresponding to the training images.
  + test\_labels (numpy.ndarray): Labels corresponding to the test images.
  + SHAPE (tuple): Dimensions (height, width) of the face images.
  + chosen\_index (list): Indices of the test images selected for detailed analysis and reconstruction.
  + output\_folder (str): Path to the folder for saving output files (eigenvalues, eigenvectors, visualizations).
  + top (int): Number of top eigenvectors to select in PCA for initial dimensionality reduction before LDA.
* Workflow:
* Average Face Calculation: Computes the mean face from the training dataset.
* Image Centering: Subtracts the average face from each training image.
* PCA Dimensionality Reduction: Applies PCA to reduce image data dimensionality, aiding LDA's effectiveness.

**For this step, the initial dimensionality reduction by PCA affects the quality of the final LDA dimensionality reduction. So, I tried a wide range of initial dimensionality reduction by PCA to determine the best initial reduced dimension by PCA.**

* Scatter Matrix Calculation: Calculates both within-class and between-class scatter matrices.

The purpose of LDA is to maximize the ratio of between-classes to within-classes scatter,

A black background with white text

Description automatically generated

Where μ is the total mean and is the mean of class i.

A black background with white symbols

Description automatically generated

And the optimal projection matrix W can be found by:

A black background with white letters

Description automatically generated

A solution is given by solving the eigenvalue problem:



The ratio of the between-class scatter and within-class matrices will be used in the next step for solving the eigen problem.

The ratio **ratio\_B\_W** of the between class scatter and within class scatter matrices is **top\*top size, where top is the number of dimensions selected by PCA.**

* Fisherfaces Creation: Determines Fisherfaces by finding eigenvalues and eigenvectors from the scatter matrix ratio above.
* Fisherfaces Visualization: Visualizes Fisherfaces with the show\_save\_faces() function in code 0006.

*For the 25 fisherfaces, we simply resize each vector in W to SHAPE[1]xSHAPE[0] (width, height).*

* Image Reconstruction: Reconstructs test images after projecting them to lower dimension with Fisherfaces, still with fisherfaces, and visualizing the results of reconstruction after LDA with the show\_save\_faces() function.

*For the reconstruction, we project each projected image (z) back to the original space and visualize the result.*

A black background with white text

Description automatically generated

* Returns:
* None: **I will explain the rest of the function in Part 2, code 0011.** (Following the structure of the report)

0009

def LDA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, output\_folder, top):

    avg\_face = np.mean(train\_images, axis=0)

    train\_images\_centered = train\_images - avg\_face

    # ... Portion left out of the report to make the function shorter

    ''' Project Training Data onto PCA Space to reduce its dimension before LDA '''

    X\_pca = np.dot(train\_images\_centered, top\_eigen\_vectors\_pca)

    ''' Apply LDA on PCA-transformed Data '''

    num\_classes = len(np.unique(train\_labels))

    num\_features = X\_pca.shape[1]  # = 126, optimal

    S\_W = np.zeros((num\_features, num\_features)) # Within class scatter matrix

    S\_B = np.zeros((num\_features, num\_features)) )) # Between class scatter matrix

    overall\_mean = np.mean(X\_pca, axis=0)

    for i in range(1, num\_classes+1): # Be careful with the range

        class\_indices = np.where(train\_labels == i)[0]

        X\_class = X\_pca[class\_indices]

        class\_mean = np.mean(X\_class, axis=0)

        S\_W += np.dot((X\_class - class\_mean).T, (X\_class - class\_mean))

        N\_i = X\_class.shape[0]

        mean\_diff = (class\_mean - overall\_mean).reshape(num\_features, 1)

        S\_B += N\_i \* (mean\_diff).dot(mean\_diff.T)

    ''' Compute (S\_W ^-1) .dot(S\_B) '''

    ratio\_B\_W = np.dot( np.linalg.inv(S\_W), S\_B)

    ''' Perform eigenvalue decomposition to obtain eigenvalues and eigenvectors '''

    eigen\_values\_lda, eigen\_vectors\_lda = np.linalg.eig(ratio\_B\_W)

    # ... Portion left out of the report to make the function shorter

    top\_eigen\_vectors\_lda  = eigen\_vectors\_lda[:, sort\_index\_lda[0:25]].real

    ''' Compute Fisherfaces '''

    fisherfaces = np.dot(top\_eigen\_vectors\_pca, top\_eigen\_vectors\_lda)

    fisherfaces = fisherfaces.T # Shape = (25, SHAPE[0]\*SHAPE[1])

    ''' Visualize the eigenfaces (reshape and display) '''

    show\_save\_faces(fisherfaces.reshape(25, SHAPE[1], SHAPE[0]), 5, 'LDA fisherfaces', output\_folder)

  ''' Reconstruct the selected test images using the eigenfaces '''

    centered\_test\_images = test\_images[chosen\_index] - avg\_face

    projections = np.dot(centered\_test\_images, fisherfaces.T)

    reconstruction\_faces = avg\_face + np.dot(projections, fisherfaces)

    ''' Visualize the reconstructed faces'''

    show\_save\_faces(reconstruction\_faces.reshape(10, SHAPE[1], SHAPE[0]), 2,

'LDA reconstructed faces', output\_folder)

* **Part 2 : Use PCA and LDA to do face recognition, and compute the performance**

**The PCA() function** for Principal Component Analysis (PCA) is also used for Face Recognition though eigenfaces, which are the transpose of the top 25 eigenvectors from the decomposition of the covariance matrix of the training data.

Following the explanation of the function which started in code 0008, below I will show how the function performs face recognition:

* This function performs PCA then performs face recognition using training and testing datasets of face images as well as the eigenfaces.
* Parameters:
* Described in code 0008 above in Part 1.
* Workflow:
* Previous steps explained in code 0008 in Part 1…
* projections Calculation: Computes projections for both all training and test images in reduced space.
* Face Recognition: Performs k-nearest neighbors (k-NN) classification for face recognition using the computed projections with the **face\_recognition()** function described in details in code 0007 above.
* Returns:
* None

0010

def PCA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index,output\_folder):

    avg\_face = np.mean(train\_images, axis=0)

    centered\_train\_faces = train\_images - avg\_face

    # ... Portion left out of the report to make the function shorter

# ... Initial Parts of the function explained in code 0008

    ''' Calculate projections for both centered training and centered test images '''

    train\_projections = np.dot(centered\_train\_faces, eigenfaces.T)

    centered\_test\_faces = test\_images - avg\_face

    test\_projections = np.dot(centered\_test\_faces, eigenfaces.T)

    ''' Perform face recognition using the calculated projections '''

    accuracy\_list = []

    for k\_neighbors in range(1, 13):

        accuracy = face\_recognition(train\_labels, train\_projections, test\_labels,

test\_projections, k\_neighbors, title='PCA ')

        accuracy\_list.append(accuracy)

    average\_accuracy = sum(accuracy\_list) / len(accuracy\_list)

    return

**The LDA() function** for Linear Discriminant Analysis (LDA) is also used for Face Recognition though fisherfaces, which are the transpose of the top 25 eigenvectors from the decomposition of the ratio of the between-class matrix and within-class matrix, obtained from the labeled training data.

Following the explanation of the function which started in code 0009, below I will show how the function performs face recognition:

* This function performs LDA for getting the fisherfaces then performs face recognignition using training and testing datasets of images.
* Parameters:
* Described in code 0009 above in Part 1.
* Workflow:
* Previous steps explained in code 0009 in Part 1…
* projections Calculation: Computes projections for both all training and test images in reduced space.
* Face Recognition: Performs k-nearest neighbors (k-NN) classification for face recognition using the computed projections with the **face\_recognition()** function described in details in code 0007 above.
* Returns:
* None

0011

def LDA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, output\_folder, top):

    avg\_face = np.mean(train\_images, axis=0)

    train\_images\_centered = train\_images - avg\_face

    # ... Portion left out of the report to make the function shorter

# ... Initial Parts of the function explained in code 0009

    ''' Calculate projections for both centered training and centered test images '''

    train\_projections = np.dot(train\_images\_centered, fisherfaces.T)

    test\_faces\_centered = test\_images - avg\_face

    test\_projections = np.dot(test\_faces\_centered, fisherfaces.T)

    ''' Perform face recognition using the calculated projections '''

    accuracy\_list = []

    for k\_neighbors in range(1, 13):

        accuracy = face\_recognition(train\_labels, train\_projections, test\_labels,

test\_projections, k\_neighbors, title='LDA ')

        accuracy\_list.append(accuracy)

    average\_accuracy = sum(accuracy\_list) / len(accuracy\_list)

    return

* **Part 3: Use kernel PCA and kernel LDA to do face recognition, and compute the performance, then compare different kernel and with PCA/LDA**

In this part, for kernel PCA and kernel LDA I tried 3 kernel functions: RBF (Radial Basis Function) kernel, Polynomial kernel, and linear kernel.

**Below are the 3 kernel functions** as well the **compute\_kernel\_matrix() function** that will be responsible for selecting which type of kernel function will be used to compute the similarity matrix.

Here is more description of that function:

* Compute the kernel matrix between two sets of images.
* Parameters:
* kernel (str): The type of kernel to be used. Options are 'rbf', 'linear', or 'polynomial'.
* images1 (array): The first set of images, represented as an array of data (testing goes here).
* images2 (array): The second set of images, represented as an array of data.
* Returns:
* array: The computed kernel matrix between the two sets of images.

0012

def RBF\_kernel(A, B, gamma=0.1):

    return np.exp(-gamma \* cdist(A, B, 'sqeuclidean')) # Just bad no matter gamma

def linearKernel(A, B):

    return np.dot(A, B.T) # Constant = 0 is best

def polynomialKernel(A, B, coef=0, degree=2, gamma=1):

    return np.power(gamma \* (np.dot(A, B.T)) + coef, degree) # Higher degree gives lower performance

def compute\_kernel\_matrix(kernel, images1, images2):

    if kernel == 'rbf':

        return RBF\_kernel(images1, images2)

    elif kernel == 'polynomial':

        return polynomialKernel(images1, images2)

    elif kernel == 'linear':

        return linearKernel(images1, images2)

    else:

      print("\nEnter a valid kernel name: 'rbf', 'linear', or 'polynomial' \n")

    return

**The kernel\_PCA() function** will be applied to do dimensionality reduction by Kernel Principal Component Analysis (Kernel PCA) and perform Face Recognition, as well as obtain Eigenfaces, do face reconstruction.

* **This function is similar to the PCA function (codes 0008 & 0010), except for the kernel matrix used for the eigen problem, the calculation of the eigenfaces and the possible alternative way to do the image projections into reduced dimension.**
* Parameters:
  + train\_images: Array of training images.
  + test\_images: Array of test images.
  + train\_labels: Labels for the training images.
  + test\_labels: Labels for the test images.
  + SHAPE: Tuple indicating the shape of the images (height, width).
  + chosen\_index: Indices of specific test images for detailed analysis.
  + kernel: Type of kernel used in the computation.
  + output\_folder: Directory path to save output files (eigenvectors, eigenvalues, visualizations).
* The function performs the following steps:
* Computes the average face from training images like in PCA function.
* Centers the training data by subtracting the average face like in PCA function.
* Computes the kernel matrix and its centered version.

The kernel matrix K is NxN, where **N** is the number of training data points.

The kernel matrix K is first computed with the compute\_kernel\_matrix() function using the **centered training images** as inputs, then it is centered following the formula:



where 1N is a N \* N matrix for which each element takes value 1/N.

* Performs eigenvalue decomposition on the centered kernel matrix.
* Sorts eigenvalues and computes the corresponding eigenfaces.

The computation of the Eigenfaces is different than with the PCA function:

A math equations and formulas

Description automatically generated with medium confidence

Where *v* is the transpose of its corresponding eigenface, and X is the centered training images matrix.

Formula found at "Statistical Machine Learning", Summer Term 2020, Ulrike von Luxburg, University of Tübingen, Germany. Video link: <https://www.youtube.com/watch?v=7kfkGyhPDu0&t=328s>

* Visualizes the eigenfaces like in PCA function.
* Projecting images.

There are to options for projecting the images into the reduced dimension:

* + We can compute the dot product between the centered image and the transpose of the eigenfaces.

**projections\_test\_chosen = np.dot(test\_images\_chosen\_centered, eigenfaces.T)**

* + Or we can compute the kernel matrix between the centered images to project and the training data, center the obtained matrix, then compute the dot product of the centered matrix with the top 25 eigenvectors (**Not eigenfaces**) to get the projected images on each row:

**kernel\_test\_chosen = compute\_kernel\_matrix(kernel, test\_images\_chosen\_centered, centered\_train\_faces)**

**kernel\_test\_chosen\_centered = kernel\_test\_chosen - np.mean(kernel\_test\_chosen, axis = 1, keepdims=True)**

**projections\_test\_chosen = np.dot(kernel\_test\_chosen\_centered, sorted\_eigen\_vectors[:, :25])**

* **The rest of the function works like with the PCA function in codes 0008 and 0010, including Face Recognition (just change the calculation of the projections).**
* Returns:
* None

0013

def kernel\_PCA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, kernel, output\_folder):

    # ... Portion left out of the report to make the function shorter

# ... Initial Parts of the function explained in code 0008

        ''' compute kernel matrix'''

        kernel\_matrix = compute\_kernel\_matrix(kernel, centered\_train\_faces, centered\_train\_faces)

        ''' Center the kernel matrix in the unbiased form '''

        one\_N = np.ones(kernel\_matrix.shape) / ( kernel\_matrix.shape[0] - 1 ) ########## Unbiased

        kernel\_centered = (kernel\_matrix

                          - np.dot(one\_N, kernel\_matrix)

                          - np.dot(kernel\_matrix, one\_N)

                          + np.dot(np.dot(one\_N, kernel\_matrix), one\_N))

        # ... Portion left out of the report to make the function shorter

        ''' Perform eigenvalue decomposition on the centered kernel matrix '''

        eigen\_values, eigen\_vectors = np.linalg.eig(kernel\_centered)

# ... Portion left out of the report to make the function shorter

# ... Portion left out of the report to make the function shorter

    ''' Eigenfaces, 25 in total, 1 on each row by getting '''

    eigenfaces = np.dot(centered\_train\_faces.T, sorted\_eigen\_vectors[:, :25])

    eigenfaces = eigenfaces.T  # Shape = (25, SHAPE[0]\*SHAPE[1])

    ''' Visualize the eigenfaces (reshape and display) '''

    show\_save\_faces(eigenfaces.reshape(25, SHAPE[1], SHAPE[0]), 5, f'{kernel}\_PCA eigenfaces', output\_folder)

    ''' Projecting the centered test images onto the 25-D by kernel PCA method '''

    test\_images\_chosen\_centered = test\_images[chosen\_index] - avg\_face

    projections\_test\_chosen = np.dot(test\_images\_chosen\_centered, eigenfaces.T)

    # ... Portion left out of the report to make the function shorter

    ''' Projecting the ALL centered test and train images onto the 25-D by kernel PCA method '''

    test\_images\_centered = test\_images - avg\_face

    K\_test\_train = compute\_kernel\_matrix(kernel, test\_images\_centered, centered\_train\_faces)

    kernel\_tain\_train\_centered = kernel\_centered

    K\_test\_train\_centered = K\_test\_train - np.mean(K\_test\_train, axis = 1, keepdims=True)

    """ Projections of testing data points """

    projections\_test = np.dot(K\_test\_train\_centered, sorted\_eigen\_vectors[:, :25])

    """ Projections of training data points """

    projections\_train = np.dot(kernel\_tain\_train\_centered, sorted\_eigen\_vectors[:, :25])

    ''' Perform face recognition using the calculated projections '''

    accuracy\_list = []

    for k\_neighbors in range(1, 13):

        accuracy = face\_recognition(train\_labels, projections\_train, test\_labels, projections\_test,

k\_neighbors, title=f'{kernel}\_PCA ')

        accuracy\_list.append(accuracy)

    average\_accuracy = sum(accuracy\_list) / len(accuracy\_list)

    return

**The kernel\_LDA() function** will be applied to do dimensionality reduction by kernel Linear Discriminant Analysis (Kernel LDA) and perform Face Recognition, as well as obtain Fisherfaces, do face reconstruction.

* **This function is like the LDA function (codes 0009 & 0011), except for the kernel matrix used for the ratio of between and within class scatter matrices and the calculation of the fisherfaces.**
* Parameters:
  + train\_images: Array of training images.
  + test\_images: Array of test images.
  + train\_labels: Labels for the training images.
  + test\_labels: Labels for the test images.
  + SHAPE: Tuple indicating the shape of the images (height, width).
  + chosen\_index: Indices of specific test images for detailed analysis.
  + kernel: Type of kernel used in the computation.
  + output\_folder: Directory path to save output files (eigenvectors, eigenvalues, visualizations).
* The function performs the following steps:
* Computes the average face from training images like in LDA function.
* Centers the training data by subtracting the average face like in LDA function.
* Computes the kernel matrix and its centered version.

The kernel matrix K is NxN, where **N** is the number of training data points.

The kernel matrix K is first computed with the compute\_kernel\_matrix() function using the **centered training images** as inputs, then it is centered following the formula:



where 1N is a N \* N matrix for which each element takes value 1/N.

* Scatter Matrices Calculation: Computes within-class and between-class scatter matrices in the kernel-induced feature space.  Instead of original data, we compute the within-classes scatter(N) and between-classes scatter(M) with the kernel matrix. According to the [wiki](https://en.wikipedia.org/wiki/Kernel_Fisher_discriminant_analysis) and the paper given in PDF, these two scatter matrices are given by:

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

* Performs eigenvalue decomposition on the centered kernel matrix:

We can get the projected data by solving the eigenvalues of the following formula:



* Sorts eigenvalues and computes the corresponding fisherfaces.

The computation of the Fisherfaces is different than with the LDA function:

A math equations and formulas

Description automatically generated with medium confidence

Where *v* is the transpose of its corresponding fisherface, and X is the centered training images matrix.

Formula found at "Statistical Machine Learning", Summer Term 2020, Ulrike von Luxburg, University of Tübingen, Germany. Video link: <https://www.youtube.com/watch?v=7kfkGyhPDu0&t=328s>

* Visualizes the fisherfaces like in LDA function.
* Projecting images.

We can compute the dot product between the centered image and the transpose of the fisherfaces.

**projections\_test\_chosen = np.dot(test\_images\_chosen\_centered, eigenfaces.T)**

* **The rest of the function works like with the LDA function in codes 0009 and 0011, including Face Recognition (just change the calculation of the projections).**
* Returns:
* None

**0014**

def kernel\_LDA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, kernel, output\_folder):

    # ... Portion left out of the report to make the function shorter

# ... Initial Parts of the function explained in code 0009

        ''' Step 1: Compute the Kernel Matrix for the training data '''

        Kernel\_matrix\_LDA = compute\_kernel\_matrix(kernel, train\_images, train\_images)

        ''' Step 2: Center the Kernel Matrix '''

        one\_N = np.ones(Kernel\_matrix\_LDA.shape) / ( Kernel\_matrix\_LDA.shape[0] - 1 )

        K\_centered = (Kernel\_matrix\_LDA

                          - np.dot(one\_N, Kernel\_matrix\_LDA)

                          - np.dot(Kernel\_matrix\_LDA, one\_N)

                          + np.dot(np.dot(one\_N, Kernel\_matrix\_LDA), one\_N))

        # ... Portion left out of the report to make the function shorter

        ''' Step 3: Compute the Scatter Matrices in the Feature Space '''

        num\_classes = len(np.unique(train\_labels))

        num\_features = K\_centered.shape[1]

        S\_W = np.zeros((num\_features, num\_features))

        S\_B = np.zeros\_like(S\_W) # Just like S\_W

        class\_means = {}

        overall\_mean = np.mean(K\_centered, axis=0)

        for i in np.unique(train\_labels):

            indices = np.where(train\_labels == i)[0]

            X\_class = K\_centered[indices, :]

            class\_mean = np.mean(X\_class, axis=0)

            S\_W += np.dot((X\_class - class\_mean).T, (X\_class - class\_mean))

            N\_i = len(indices)

            mean\_diff = class\_mean - overall\_mean

            S\_B += N\_i \* np.outer(mean\_diff, mean\_diff)

        ''' Step 4: Solve the Eigenvalue Problem '''

        ratio\_B\_W = np.dot( np.linalg.pinv(S\_W), S\_B )

        eigen\_values, eigen\_vectors = np.linalg.eig(ratio\_B\_W)

        # ... Portion left out of the report to make the function shorter

    # ... Portion left out of the report to make the function shorter

    sorted\_indices = np.argsort(-eigen\_values)

    top\_eigen\_vectors = eigen\_vectors[:, sorted\_indices[0:25]]

    ''' Step 5: Here we have our fisherfaces, 25 in total, 1 on each row '''

    fisherfaces = np.dot(centered\_train\_faces.T, top\_eigen\_vectors)

    fisherfaces = fisherfaces.T  # Shape = (25, SHAPE[0]\*SHAPE[1])

    ''' Visualize the fisherfaces (reshape and display) '''

    show\_save\_faces(fisherfaces.reshape(25, SHAPE[1], SHAPE[0]), 5, f'{kernel} LDA fisherfaces', output\_folder)

    ''' Project the selected test images onto the fisherfaces to obtain projections '''

    centered\_test\_images = test\_images[chosen\_index] - avg\_face

    projections\_chosen = np.dot(centered\_test\_images, fisherfaces.T)

    # ... Portion left out of the report to make the function shorter

    ''' Calculate projections for both centered training and centered test images '''

    train\_projections = np.dot(centered\_train\_faces, fisherfaces.T)

    test\_faces\_centered = test\_images - avg\_face

    test\_projections = np.dot(test\_faces\_centered, fisherfaces.T)

    ''' Perform face recognition using the calculated weights '''

    perprint("")

    accuracy\_list = []  # List to store accuracies for each k\_neighbors

    for k\_neighbors in range(1, 13):

        accuracy = face\_recognition(train\_labels, train\_projections, test\_labels, test\_projections,

k\_neighbors, title=f'{kernel} LDA ')

        accuracy\_list.append(accuracy)

    average\_accuracy = sum(accuracy\_list) / len(accuracy\_list)

    return

* **t-SNE**

: This part in the notebook is the concerned section.

I will first explain the parts that support part 1, 2, 3 and 4 (common functions).

I begin with setting up the paths for accessing the MNIST2500 dataset for the 2500 mnist images and their labels. I also set up the output directory where we will save the images and GIFs generated later.

Here is the corresponding code below.

**0015**

""" II. t-SNE """

import os

# Input directory

input\_dir = "/content/drive/MyDrive/…/tsne\_python/tsne\_python"

# Output directory

output\_dir = "/content/drive/MyDrive/…/t\_sym\_sne\_output"

# Ensure output directory exists

os.makedirs(output\_dir, exist\_ok=True)

Next, we will import all additional required packages.

**os**: For interacting with the operating system for file and directory manipulation.

**time**: For measuring durations of certain parts of the code.

**random**: For generating random numbers.

**numpy**: For matrix manipulations and mathematical operations.

**matplotlib.pyplot**: For plotting, displaying and saving figures.

**PIL.Image** For reading images from the input directories.

**scipy.spatial.distance.cdist**: For distance calculation between pairs of vectors.

**imageio**: Used for making the GIFs

**seaborn**: to create a light color palette based on shades of blue for visualizing the high and low dimensional spaces.

**0016**

import time ### New

import numpy as np

#import pylab # removed

import imageio.v2 as imageio ### New

import seaborn as sns ### New

import matplotlib.pyplot as plt ### New

from IPython.display import Image, display ### New

from scipy.spatial.distance import cdist ### New

* **Part 1: Explain implementation of t-SNE and show modifications to get symmetric SNE**

**The first function to look at is the Hbeta()** function that is common to t-SNE and symmetric SNE.

This function calculates the perplexity and P-row for a given precision value (beta) in a Gaussian distribution. It computes the P-row by exponentiating the negative product of input matrix D and beta. The function returns the perplexity (H) and the normalized P-row.

**0017**

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

**The function x2p()** performs a binary search to compute P-values such that each conditional Gaussian has the same perplexity.

It takes an input matrix X, which represents data points, and calculates pairwise distances between them.

The binary search is applied to determine the precision values (beta) for the Gaussian distribution, ensuring a consistent perplexity.

The result is a P-matrix indicating conditional probabilities between data points.

**0018**

def x2p(X=np.array([]), tol=1e-5, perplexity=30.0):

    (n, d) = X.shape

    sum\_X = np.sum(np.square(X), 1)

    D = cdist(X, X, 'sqeuclidean')

    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:

        # 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

    return P

**The function pca()** applies Principal Component Analysis (PCA) to the input NxD array X (N is number of datapoints and D is the dimension of the data points) to reduce its dimensionality to no\_dims dimensions. It preprocesses the data by subtracting the mean and then computes the eigenvectors and eigenvalues of the covariance matrix:

**Covariance matrix = np.dot(X.T, X)**

The eigenvectors are sorted in descending order of eigenvalues, and the first no\_dims eigenvectors are used to transform the data.

The result is a reduced-dimensional representation of the input data that is returned as Y.

**0019**

def pca(X=np.array([]), no\_dims=50):

    (n, d) = X.shape

    X = X - np.tile(np.mean(X, 0), (n, 1))

    (l, M) = np.linalg.eig(np.dot(X.T, X))

    # Sort the eigenvalues and eigenvectors in descending order of eigenvalues

    idx = l.argsort()[::-1]   ### New

    l = l[idx]                ### New

    M = M[:, idx]             ### New

    Y = np.dot(X, M[:, 0:no\_dims])

    return Y

**The mode\_tsne() function** is the modification of the original tSNE() function to now make it capable of performing t-SNE (t-Distributed Stochastic Neighbor Embedding) and symmetric SNE (symmetric-Distributed Stochastic Neighbor Embedding) based on the specified mode (‘t-SNE’ or ‘sym-SNE’), on the dataset represented by the NxD array X.

* It aims to reduce the dimensionality of the data to no\_dims dimensions. The function utilizes PCA for preprocessing and then performs t-SNE or symmetric-SNE iterations to optimize the embedding.
* Parameters:
* directory: Specifies the directory for saving the visualization plots.
* mode: Specifies the mode, either 't-SNE' or 'sym-SNE'.
* interval: Determines the interval for plotting intermediate results during iterations.
* labels: Represents the labels or categories associated with the data points.
* X: NxD array representing the dataset (default is an empty array), (N is number of datapoints and D is their dimension).
* no\_dims: Desired dimensionality of the output (default is 2).
* initial\_dims: Number of dimensions after PCA preprocessing (default is 50).
* perplexity: Perplexity parameter for t-SNE (default is 30.0).
* The function first reduces the dimension of X from D to initial\_dims, computes P probability of distances at PCA reduce high dimension, then initialized the reduced Q low dimension embedding probability of distances and iteratively refines the embedding, adjusting the coordinates of data points in the reduced-dimensional space. It also generates plots at specified intervals to **visualize the optimization process**.
* The mode parameter allows you to choose between regular t-SNE and symmetric t-SNE.
* The function returns the reduced-dimensional representation (Y), P-values for the PCA reduced high-dimension, Q-values for the reduced low-dimension, and the number of generated figures during the optimization process.

**0020**

def mode\_tsne(directory, mode, interval, labels, X=np.array([]), no\_dims=2, initial\_dims=50, perplexity=30.0):

**Now, let’s look at a more specific description of the difference in calculation for t-SNE and for symmetric SNE inside the function**.

The major difference between t-SNE and symmetric SNE is that, **while symmetric SNE uses** Gaussian distribution for the low dimensionality embedding’s probability of distances, **t-SNE uses** Student-T distribution for the low dimensionality embedding’s probability of distances. But both still use Gaussian distribution for the probability of distances at high dimension.

The reason why t-SNE use student-T distribution is because with the Gaussian distribution for the probability of distances of the embedding, the normal distribution of the probabilities of distances tends to go faster near zeros for high probability of distances. Which means that at low embeddings the points will be crowded, which is the crowded problem of symmetric SNE as well as original SNE.

On the other hand, the normal distribution of the probability of distances when using a Student-T distribution to build the embedding Q has a longer tail, meaning that we can get far away points more spread out from the close points in the low dimension.

Since Gaussian distribution may cause crowding problem in low dimension, Student-T distribution can alleviate this problem. In the [implementation of t-SNE by Laurens van der Maaten](https://lvdmaaten.github.io/tsne/), the code for computing joint probability is:

**0021**

sum\_Y = np.sum(np.square(Y), 1)

        num = -2. \* np.dot(Y, Y.T)

        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)

and the corresponding formula is:

A math equations on a black background

Description automatically generated

For the gradient of t-SNE, the code and formula are listed as follows:

**0022**

  for i in range(n):

     dY[i, :] = np.sum(np.tile(PQ[:, i] \* num[:, i], (no\_dims, 1)).T \* (Y[i, :] - Y), 0)

and the corresponding formula is:

A black background with white text

Description automatically generated

The num in the codes above is the numerator of the joint probability formula. Q is the matrix of joint probability distribution in low dimension. dims is the dimension we want to embed, which is 2.

In this Part 1, we want to modify the code and make it back to symmetric SNE. The formula for computing joint probability and gradient in symmetric SNE are shown as follows:

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

We can achieve this by changing the num and dY in t-SNE with the following code:

**0023**

        # ...

elif mode == 'sym-SNE':

            num = np.exp(-1 \* cdist(Y, Y, 'sqeuclidean'))

        # Compute gradient

        # ...

        elif mode == 'sym-SNE':

            for i in range(n):

                dY[i, :] = np.sum(np.tile(PQ[:, i] \* num[:, i], (no\_dims, 1)).T \* (Y[i, :] - Y), 0)

* **Part 2: Visualize the embedding of both t-SNE and symmetric SNE**

**With the plotResult() function**, I will visualize and save the image representation of Y, low embedding of X, at specific intervals of the running of the mode\_tsne() function.

* Parameters:
* Y: Reduced-dimensional representation of the dataset.
* labels: Labels or categories associated with data points.
* idx: Current iteration index during the optimization process.
* interval: Specifies the interval for saving plots.
* method: The embedding method used (e.g., t-SNE).
* perplexity: Perplexity parameter used in the embedding method.
* directory: The directory to save the generated plots.

The resulting plots are saved in the specified directory with filenames indicating the method, perplexity, and iteration. If no interval is specified, the plot is also displayed.

**0024**

def plotResult(Y, labels, idx, interval, method, perplexity, directory):

    plt.clf()

    scatter = plt.scatter(Y[:, 0], Y[:, 1], 20, labels)

    plt.legend(\*scatter.legend\_elements(), loc='center left', bbox\_to\_anchor=(1, 0.5), title='Digit')

    for label in set(labels):

        indices = labels == label

        center\_x = np.mean(Y[indices, 0])

        center\_y = np.mean(Y[indices, 1])

        plt.text(center\_x, center\_y, str(int(label)), fontsize=20, ...)

    plt.title(f'{method}, perplexity: {perplexity}, iteration: {idx}')

    plt.tight\_layout()

    if interval:

        plt.savefig(os.path.join(directory,f"{method}\_perplexity\_{perplexity}\_Y\_{idx//interval:03d}.png"))

    else:

        plt.savefig(os.path.join(directory, f"{method}\_perplexity\_{perplexity}\_Y\_{idx}.png"))

        plt.show()

    return

**The compose\_gif() function** is called at the end of the optimization process.

* It is designed to create a GIF animation from a series of images generated during an optimization process.
* Parameters:
* method: The embedding method used (‘t-SNE’ or ‘sym-SNE’).
* number\_images: The total number of images to include in the GIF, excluding the final image.
* directory: The directory where the images are stored.
* Function Steps:
* Temporary List for Images (gif\_images): Initialize an empty list to store the images that will be read from the directory.
* Read each image using the imageio.imread function.
* Remove the read image file from the directory.
* Add Final Image without deleting from directory.
* Transform and save the list as GIF. Use imageio.mimsave to create a GIF from the list of images.
* Specify the frames per second (fps) for the GIF, in this case, set to 10.
* There is no return

**0025**

def compose\_gif(method, number\_images, directory):

    gif\_images = []

    for i in range(number\_images):

        gif\_image\_path = os.path.join(directory, f"{method}\_perplexity\_{perplexity}\_Y\_{i:03d}.png")

        gif\_images.append(imageio.imread(gif\_image\_path))

        os.remove(gif\_image\_path)

    gif\_image\_path = os.path.join(directory, f"{method}\_perplexity\_{perplexity}\_Y\_final.png")

    gif\_images.append(imageio.imread(gif\_image\_path))

    gif\_image\_path = os.path.join(output\_dir, f"{method}\_perplexity\_{perplexity}.gif")

    imageio.mimsave(gif\_image\_path, gif\_images, fps=10)

* **Part 3: Visualize the distribution of pairwise similarities in both high-dimensional space and low-dimensional space, based on both t-SNE and symmetric SNE.**

**The plot\_High\_Dim\_Low\_Dim() function** will serve in visualizing the pairwise similarities in both high-dimensional space and low-dimensional space, based on both t-SNE and symmetric SNE.

* It creates and saves two heatmap visualizations. Each heatmap represents the pairwise similarities between data points, but one is for the high-dimensional space (P), and the other is for the low-dimensional space (Q).
* Parameters:
* P: Pairwise similarities matrix in the high-dimensional space.
* Q: Pairwise similarities matrix in the low-dimensional space.
* method: The embedding method used (e.g., t-SNE).
* perplexity: Perplexity parameter used in the embedding method.
* directory: The directory where the images are stored.
* Function Steps:
* Initialize Color Palette: Generate a color palette for visualization using seaborn's light palette function.
* Plot, Save and Show High-Dimension Similarities Heatmap.
* Plot, Save and Show Low-Dimension Similarities Heatmap.
* Note: The function assumes that P and Q are square matrices representing pairwise similarities between data points in their respective spaces.
* There is no return.

**0026**

def plot\_High\_Dim\_Low\_Dim(P, Q, method, perplexity, directory):

    pal = sns.light\_palette('blue', as\_cmap=True)

    plt.figure()

    plt.title('High-Dimension Similarities')

    plt.imshow(P, cmap='cividis', interpolation='nearest')

    plt.clim()  # Set color limits

    plt.savefig(os.path.join(directory, f"{method}\_perplexity\_{perplexity}\_High-Dim.png"))

    plt.show()

    plt.figure()

    plt.title('Low-Dimension Similarities')

    plt.imshow(Q, cmap='cividis', interpolation='nearest')

    plt.clim()  # Set color limits

    plt.savefig(os.path.join(directory, f"{method}\_perplexity\_{perplexity}\_Low-Dim.png"))

    plt.show()

* **Part 4: Try to play with different perplexity values.**

We iterate over a list of 6 different perplexity values (perplexities = [5., 20., 30., 40., 50., 100.]) to obtain and visualize results for both t-SNE and symmetric SNE (sym-SNE).

Below is the code:

**0028**

if \_\_name\_\_ == "\_\_main\_\_":

    interval = 10

    perplexities = [5., 20., 30., 40., 50., 100.]

    methods = ['t-SNE', 'sym-SNE']

    for method in methods:

        for perplexity in perplexities:

                Y, P, Q, number\_images = mode\_tsne(output\_dir, method, interval, labels, X, 2, 500

, perplexity)

                plotResult(Y, labels, 'final', None, method, perplexity, output\_dir)

                compose\_gif(method, number\_images, output\_dir)

                plot\_High\_Dim\_Low\_Dim(P, Q, method, perplexity, output\_dir)

The GIFs frame rate is set to 10 fps, you can change it to whatever frame rate by uncommenting this part and changing the value of **new\_frame\_rate.**

**0029**

            """

            # Desired frame rate (replace 10 with your preferred frame rate)

            new\_frame\_rate = 5 # Default is 10

            # Read the input GIF

            gif = imageio.mimread(gif\_image\_path)

            # Write the output GIF with the new frame rate

            imageio.mimsave(gif\_image\_path, gif, fps = new\_frame\_rate)

            """

**The show\_existing\_images() function** comes to be useful when trying different perplexities by helping in not recalculating already existing erplexities.

* It is designed to display existing image files associated with a specific method and perplexity in a specified directory.
* Parameters:
* directory: The directory where the images are stored.
* method: The embedding method used (e.g., t-SNE).
* perplexity: Perplexity parameter used in the embedding method.
* Function Steps:
* Construct Image Paths.
* Display Images: Attempt to display each image using the display function from IPython.display.
* Handle cases where the file is not found or if any other exceptions occur during display.
* Note: The function assumes that the images follow a specific naming convention based on the method and perplexity.
* There is no return.

**0030**

def show\_existing\_images(directory, method, perplexity):

    final\_image\_path = os.path.join(directory, f"{method}\_perplexity\_{perplexity}\_Y\_final.png")

    high\_D\_image\_path = os.path.join(directory, f"{method}\_perplexity\_{perplexity}\_High-Dim.png")

    low\_D\_image\_path = os.path.join(directory, f"{method}\_perplexity\_{perplexity}\_Low-Dim.png")

    paths = [final\_image\_path, high\_D\_image\_path, low\_D\_image\_path]

    for path in paths:

        try:

            display(Image(filename = path))

        except FileNotFoundError:

            print(f"\nThe file {path} does not exist.\n")

        except Exception as e:

            print(f"\nAn error occurred: {e}\n")

**2. Experiments and Discussion**

Here I will share the function settings and the experimental results.

* **Kernel Eigenfaces**

Experiment settings and loading the datasets are done as below:

Here we set the resizing shape described in code 0005 of read\_dataset() function to (98, 116) (width, height), and load the training and testing images.

We also randomly select the indices of the 10 test images that will be reconstructed. This way all the functions will reconstruct the same images, giving a better insight during visualization of the reconstruction by different functions.

SHAPE = (98, 116) # Resize shape (width, height) of image before processing

train\_images, train\_labels = read\_dataset(training\_folder, SHAPE)

test\_images, test\_labels = read\_dataset(testing\_folder, SHAPE)

''' Choose 10 random test images '''

chosen\_index = random.sample(range(len(test\_images)), 10)

* **Part 1: 25 eigenfaces & fisherfaces + reconstruction of 10 random images**

Here we **call the PCA() function** described in code 0008 to show the first 25 eigenfaces and the reconstruction of the chosen 10 random images. We will also assess the run time.

# Reminder: SHAPE = (98, 116)

start\_time = time.time()

PCA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, output\_folder)

perprint(f"PCA executed in: {int(round(time.time() - start\_time))} seconds")

Below are the results:

Figure 01

|  |  |
| --- | --- |
| **25 first eigenfaces** | **10 reconstructed images** |
|  |  |

For LDA, because it involves initially reducing the dimension of the input data, train\_images, before performing LDA, and because the selection of the initial reduced dimension is also critical for the performance of LDA, in the section

A close up of a sign

Description automatically generated

Of the notebook I plotted the performance of LDA on face recognition for **values of initial reduced dimension between 25 and 300.**

Below are the results of that plot:

**Plot 01**

|  |  |
| --- | --- |
| **LDA Performance VS Top PCA Eigenvectors** | **Conclusion** |
|  | From this operation the optimal initial reduced dimension by PCA before performing LDA is  **Initial reduced dimension = 126.** |

**We now call the LDA() function** described in code 0009 to show the first 25 fisherfaces and the reconstruction of the chosen 10 random images. We will also assess the run time.

# Reminder: SHAPE = (98, 116)

start\_time = time.time()

initial\_dim\_reduction = 126

LDA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, output\_folder, initial\_dim\_reduction)

perprint(f"LDA executed in: {int(round(time.time() - start\_time))} seconds")

Below are the results:

**Figure 02**

|  |  |
| --- | --- |
| **25 first fisherfaces** | **10 reconstructed images** |
|  |  |

* **Part 2 : Use PCA and LDA to do face recognition, and compute the performance**

**After calling the PCA() and LDA() functions in Part 1,** they will print out the performance for face recognition using KNN, for values of K neighbors in the range of 1 to 12 (inclusive), as well as the average performance.

Below are the results:

**Performance PCA & LDA**

|  |  |
| --- | --- |
| **Performance from running PCA and KNN** | **Performance from running LDA and KNN** |
| K= 1, PCA Accuracy: 83.33 %  K= 2, PCA Accuracy: 83.33 %  K= 3, PCA Accuracy: 83.33 %  K= 4, PCA Accuracy: 86.67 %  K= 5, PCA Accuracy: 90.00 %  K= 6, PCA Accuracy: 86.67 %  K= 7, PCA Accuracy: 90.00 %  K= 8, PCA Accuracy: 83.33 %  K= 9, PCA Accuracy: 80.00 %  K= 10, PCA Accuracy: 80.00 %  K= 11, PCA Accuracy: 80.00 %  K= 12, PCA Accuracy: 80.00 % | K= 1, LDA Accuracy: 100.00 %  K= 2, LDA Accuracy: 100.00 %  K= 3, LDA Accuracy: 100.00 %  K= 4, LDA Accuracy: 100.00 %  K= 5, LDA Accuracy: 100.00 %  K= 6, LDA Accuracy: 100.00 %  K= 7, LDA Accuracy: 100.00 %  K= 8, LDA Accuracy: 100.00 %  K= 9, LDA Accuracy: 100.00 %  K= 10, LDA Accuracy: 100.00 %  K= 11, LDA Accuracy: 100.00 %  K= 12, LDA Accuracy: 100.00 % |
| **Average: 83.89 %** | **Average: 100.00 %** |

**Discussion for Part 2**

LDA aims to reduce dimensionality with the purpose of maximizing class separability. As shown in the results above, when you select the right initial reduced dimensionality by PCA before performing LDA, the facial recognition accuracy of using KNN after LDA can reach its maximum, here it is a **perfect 100 % accuracy on testing data for initial reduced dimension = 126**.

**Observation for Part 2**

However, if you choose the wrong initial reduced dimension by PCA before doing LDA, the performance can go as low as below 10%, meaning random guess work.

I have actually tried using the full range of the feature length, i.e. SHAPE[0]\*SHAPE[1] = 98\*116 = 11368, but the performance was also very bad.

Thus, from **Plot 01** above, we can conclude that the best performances for LDA will be obtained when the initial reduced dimension by PCA is between 25 and 126 (inclusive).

From **Performance PCA & LDA** above, we can tell that for PCA, the best performance for Face Recognition is when K neighbors is equal to 5 or 7.

In general, LDA has a better performance in terms of face recognition, which seems to align with the goal of LDA to maximize class separability.

From **Figure 01** and **Figure 02**, when looking at the quality of the reconstruction of the faces, we can clearly tell that the reconstruction of the faces after the dimension had been reduced by PCA has the best visual quality.

**Thus, PCA provides a better balance between Compression and Quality.** PCA strikes a good balance between reducing the dimensionality (which is essential for computational efficiency and potentially reducing overfitting) and maintaining a high level of visual fidelity in the reconstructed images.

* **Part 3: Use kernel PCA and kernel LDA to do face recognition, and compute the performance, then compare different kernel and with PCA/LDA**

**We now call the kernel\_PCA() function** described in code 0013 to show the first 25 fisherfaces, the reconstruction of the chosen 10 random images, and its performance with KNN for Face recognition, using 3 different types of kernel functions.

kernel\_PCA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, 'linear', output\_folder)

kernel\_PCA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, 'polynomial', output\_folder)

kernel\_PCA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, 'rbf', output\_folder)

Below are the results:

**Figure 03: Eigenfaces from kernel PCA with 3 kernels**

|  |  |  |
| --- | --- | --- |
| Linear kernel | RBF kernel | Polynomial Kernel |
|  |  |  |

Observation for Part 3 on eigenfaces:

Looking at the Eigenfaces from different kernels, there is not much visible difference, except the eigenfaces for the RBF kernel look a little darker.

**Figure 04: Reconstruction of 10 faces after kernel PCA with 3 kernels**

|  |  |  |
| --- | --- | --- |
| Linear kernel | RBF kernel | Polynomial Kernel |
|  |  |  |

Observation for Part 3 on reconstructed faces after kernel PCA:

The quality of the reconstructed images is lower than with PCA and LDA.

Looking at the reconstructed faces, the quality seems close, but the best quality seems to be the faces from the linear kernel.

Thus, the linear kernel offers the best quality preservation from compression of images.

P**erformance kernel PCA with 3 kernels**

|  |  |  |
| --- | --- | --- |
| Linear kernel | RBF kernel | Polynomial Kernel |
| K= 1, linear\_PCA Accuracy: 80.00 %  K= 2, linear\_PCA Accuracy: 80.00 %  K= 3, linear\_PCA Accuracy: 83.33 %  K= 4, linear\_PCA Accuracy: 83.33 %  K= 5, linear\_PCA Accuracy: 83.33 %  K= 6, linear\_PCA Accuracy: 83.33 %  K= 7, linear\_PCA Accuracy: 80.00 %  K= 8, linear\_PCA Accuracy: 80.00 %  K= 9, linear\_PCA Accuracy: 83.33 %  K= 10, linear\_PCA Accuracy: 83.33 %  K= 11, linear\_PCA Accuracy: 83.33 %  K= 12, linear\_PCA Accuracy: 83.33 %  Average: 82.22 % | K= 1, rbf\_PCA Accuracy: 16.67 %  K= 2, rbf\_PCA Accuracy: 16.67 %  K= 3, rbf\_PCA Accuracy: 13.33 %  K= 4, rbf\_PCA Accuracy: 10.00 %  K= 5, rbf\_PCA Accuracy: 13.33 %  K= 6, rbf\_PCA Accuracy: 13.33 %  K= 7, rbf\_PCA Accuracy: 10.00 %  K= 8, rbf\_PCA Accuracy: 10.00 %  K= 9, rbf\_PCA Accuracy: 6.67 %  K= 10, rbf\_PCA Accuracy: 6.67 %  K= 11, rbf\_PCA Accuracy: 6.67 %  K= 12, rbf\_PCA Accuracy: 6.67 %  Average: 10.83 % | K= 1, polynomial\_PCA Accuracy: 60.00 %  K= 2, polynomial\_PCA Accuracy: 63.33 %  K= 3, polynomial\_PCA Accuracy: 63.33 %  K= 4, polynomial\_PCA Accuracy: 63.33 %  K= 5, polynomial\_PCA Accuracy: 60.00 %  K= 6, polynomial\_PCA Accuracy: 53.33 %  K= 7, polynomial\_PCA Accuracy: 56.67 %  K= 8, polynomial\_PCA Accuracy: 53.33 %  K= 9, polynomial\_PCA Accuracy: 53.33 %  K= 10, polynomial\_PCA Accuracy: 53.33 %  K= 11, polynomial\_PCA Accuracy: 60.00 %  K= 12, polynomial\_PCA Accuracy: 56.67 %  Average: 58.06 % |

Observation for Part 3 on performance of kernel PCA:

The 3 kernel PCA perform less than PCA and LDA. Meaning the use of a kernel method is not really required here.

From the average performances, the linear kernel function by far gives the best results at 82.22%, followed by the polynomial kernel at 58.06% and with the RBF kernel last at a bad 10.83%.

The conclusion from this observation is that the data is highly linear and not complex. Meaning since linear kernel is best and performs best ton linear data, that is the conclusion.

Polynomial kernel follows at 58% meaning the data is more closely linearly separable than with a curve.

**The conclusion** for all the observations on kernel PCA is that the linear kernel PCA offers the best performance as well as the best image quality preservation at a not too bad 82.22% average accuracy.

**We now call the kernel\_LDA() function** described in code 0014 to show the first 25 fisherfaces, the reconstruction of the chosen 10 random images, and its performance with KNN for Face recognition, , using 3 different types of kernel functions.

kernel\_PCA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, 'linear', output\_folder)

kernel\_PCA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, 'polynomial', output\_folder)

kernel\_PCA(train\_images, test\_images, train\_labels, test\_labels, SHAPE, chosen\_index, 'rbf', output\_folder)

Below are the resultrs:

**Figure 05: Fisherfaces from kernel LDA with 3 kernels**

|  |  |  |
| --- | --- | --- |
| Linear kernel | RBF kernel | Polynomial Kernel |
|  |  |  |

Observation for Part 3 on fisherfaces:

The fisherfaces seem not too bad, but the polynomial fisherfaces seem to look better.

**Figure 06: Reconstruction of 10 faces after kernel LDA with 3 kernels**

|  |  |  |
| --- | --- | --- |
| Linear kernel | RBF kernel | Polynomial Kernel |
|  |  |  |

Observation for Part 3 on reconstructed faces after kernel LDA:

The quality of the reconstructed images is lower than with PCA and LDA.

The polynomial kernel offers the best quality among the 3 kernels. Meaning it has a better preservation of quality when doing compression.

P**erformance kernel LDA with 3 kernels**

|  |  |  |
| --- | --- | --- |
| Linear kernel | RBF kernel | Polynomial Kernel |
| K= 1, linear LDA Accuracy: 86.67 %  K= 2, linear LDA Accuracy: 86.67 %  K= 3, linear LDA Accuracy: 86.67 %  K= 4, linear LDA Accuracy: 86.67 %  K= 5, linear LDA Accuracy: 86.67 %  K= 6, linear LDA Accuracy: 86.67 %  K= 7, linear LDA Accuracy: 86.67 %  K= 8, linear LDA Accuracy: 86.67 %  K= 9, linear LDA Accuracy: 86.67 %  K= 10, linear LDA Accuracy: 86.67 %  K= 11, linear LDA Accuracy: 86.67 %  K= 12, linear LDA Accuracy: 86.67 %  Average: 86.67 % | K= 1, rbf LDA Accuracy: 66.67 %  K= 2, rbf LDA Accuracy: 63.33 %  K= 3, rbf LDA Accuracy: 63.33 %  K= 4, rbf LDA Accuracy: 63.33 %  K= 5, rbf LDA Accuracy: 60.00 %  K= 6, rbf LDA Accuracy: 60.00 %  K= 7, rbf LDA Accuracy: 60.00 %  K= 8, rbf LDA Accuracy: 56.67 %  K= 9, rbf LDA Accuracy: 56.67 %  K= 10, rbf LDA Accuracy: 53.33 %  K= 11, rbf LDA Accuracy: 50.00 %  K= 12, rbf LDA Accuracy: 53.33 %  Average: 58.89 % | K= 1, polynomial LDA Accuracy: 86.67 %  K= 2, polynomial LDA Accuracy: 90.00 %  K= 3, polynomial LDA Accuracy: 86.67 %  K= 4, polynomial LDA Accuracy: 86.67 %  K= 5, polynomial LDA Accuracy: 90.00 %  K= 6, polynomial LDA Accuracy: 90.00 %  K= 7, polynomial LDA Accuracy: 93.33 %  K= 8, polynomial LDA Accuracy: 90.00 %  K= 9, polynomial LDA Accuracy: 90.00 %  K= 10, polynomial LDA Accuracy: 93.33 %  K= 11, polynomial LDA Accuracy: 93.33 %  K= 12, polynomial LDA Accuracy: 96.67 %  Average: 90.56 % |

Observation for Part 3 on performance of kernel LDA:

The average performance of both polynomial LDA and Linear LDA are higher than those of PCA. The RBF kernel performances the worst, but above average, and lower than PCA and LDA.

From the fact that the polynomial kernel performed better that the linear kernel, we can conclude that the boundary between classes is separable better with a slightly polynomial curve in higher dimension than with a straight line, because the linear kernel also performed not too bad.

General observation and conclusion

LDA still has the best performance among all the tested methods.

Polynomial kernel has the 2nd best performance. Lineral kernel LDA is 3rd and PCA is 4th at 83.39%. Linear kernel PCA is 5th at 82.22%, not bad. All the rest don’t really have a good performance.

If the goal is to maximize class separability, using LDA is the best choice.

I you want the best balance between class separation and maintaining quality of compression, PCA is the best choice.

* **t-SNE**

By running this code, at the given perplexities, we will get all the results at once.

if \_\_name\_\_ == "\_\_main\_\_":

    interval = 10

    perplexities = [5., 20., 30., 40., 50., 100.]

    methods = ['t-SNE', 'sym-SNE']

    for method in methods:

        for perplexity in perplexities:

            print(f"\nPerform {method} on MNIST dataset.")

            gif\_image\_path = os.path.join(output\_dir, f"{method}\_perplexity\_{perplexity}.gif")

                Y, P, Q, number\_images = mode\_tsne(output\_dir, method, interval, labels, X, 2, 50,

perplexity)

                plotResult(Y, labels, 'final', None, method, perplexity, output\_dir)

                compose\_gif(method, number\_images, output\_dir)

                plot\_High\_Dim\_Low\_Dim(P, Q, method, perplexity, output\_dir)

* **Part 1: Explain implementation of t-SNE and show modifications to get symmetric SNE**

Following the explanation of the modifications done to get move to symmetric SNE, I also observed from the code that the initial exageration needs to be changed when doing symmetric SNE.

    if mode == 't-SNE':

        P = P \* 4.                  # early exaggeration

    elif mode == 'sym-SNE':

        P = P \* 2.                  # early exaggeration

and

        # Stop lying about P-values

        if iter == 100:

            if mode == 't-SNE':

                P = P / 4.

            elif mode == 'sym-SNE':

                P = P / 2.

This is done to keep in line with the coeffiecients in formula of the gradients of both t-SNE and Symmetric SNE discussed before.

Also, I reordered the eigenvectors before returning by the pca function.

    # Sort the eigenvalues and eigenvectors in descending order of eigenvalues

    idx = l.argsort()[::-1]   ### New

    l = l[idx]                ### New

    M = M[:, idx]             ### New

* **Part 2: Visualize the embedding of both t-SNE and symmetric SNE**

From running the code already announced before starting part 1, here are the results for perplexity = 30.0:

|  |  |
| --- | --- |
| t-SNE | Symmetric SNE |
|  |  |

|  |  |
| --- | --- |
| GIF is called t-SNE\_perplexity\_30.0.gif in zip | GIF is called sym-SNE\_perplexity\_30.0.gif in zip |

* **Part 3: Visualize the distribution of pairwise similarities in both high-dimensional space and low-dimensional space, based on both t-SNE and symmetric SNE.**

From running the code already announced before starting part 1, here are the results for perplexity = 30.0:

|  |  |
| --- | --- |
| t-SNE: high dimension on top and low dimension below | Symmetric SNE: high dimension on top and low dimension below |
|  |  |
|  |  |

* **Part 4: Try to play with different perplexity values.**

|  |  |  |
| --- | --- | --- |
| t-SNE | Symmetric SNE | Perplexity |
|  |  | 5.0 |
|  |  | 20.0 |
|  |  | 30.0 |
|  |  | 50.0 |
|  |  | 100.0 |

Results discussion:

In general, for both, higher perplexity means better class separation.

t-SNE:

By tuning the perplexity, we can balance the attention between local and global aspects of our data. For me, I think that perplexity means the number of close neighbors each point has. We can see that with perplexity=5, the points in each group are locate loosely. Each group is divided into a few small groups. When the perplexity grows up, the points in each group locate more and more tightly. At the same time, the gaps between each group become smaller.

Symmetric SNE:

Since symmetric has crowding problem, the differences between each perplexity are not apparent comparing to t-SNE.