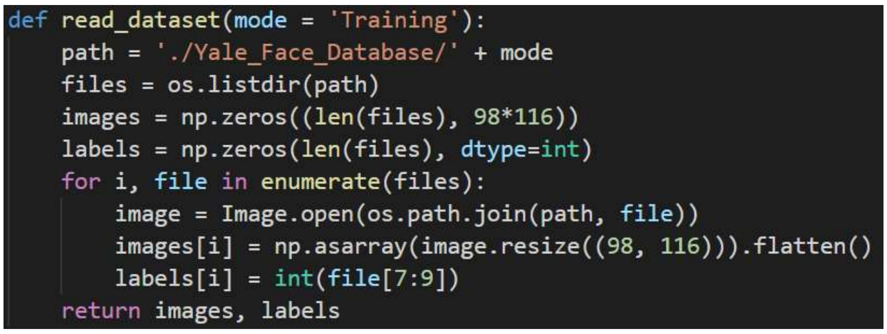
ML Homework 7 Report

I. Code with detail explanations

1. Kernel Eigenfaces

- Read data

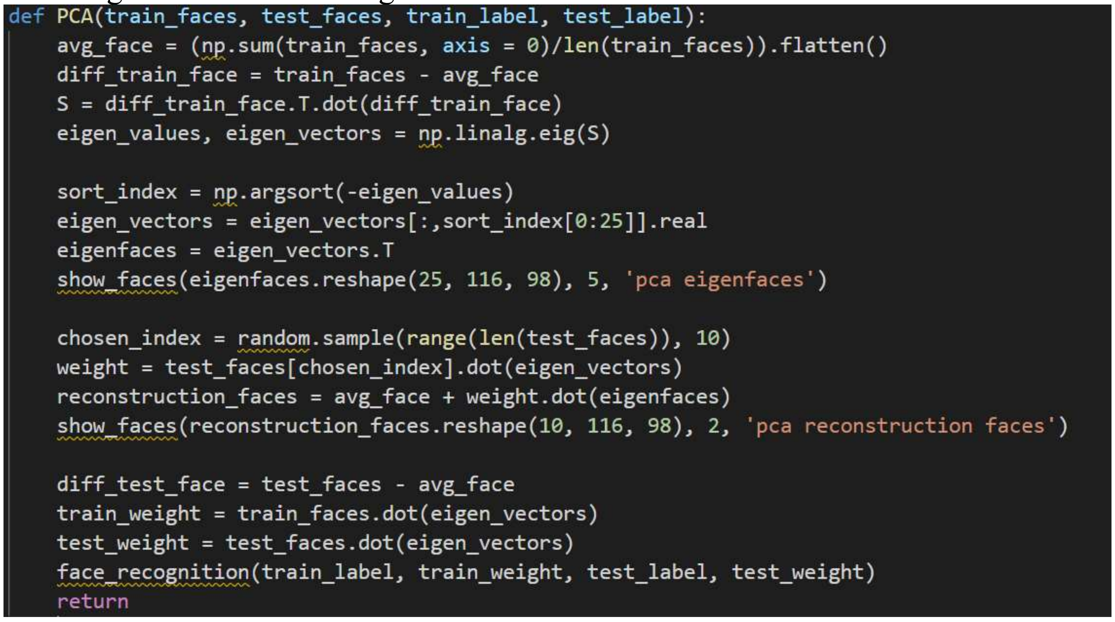
  Load training and testing faces. Resize the images to reduce the complexity of matrix operations and convert the image into a one-dimensional vector



```python
def read_dataset(mode = 'Training'):
    path = './Yale_Face_Database/' + mode
    files = os.listdir(path)
    images = np.zeros((len(files), 98*116))
    labels = np.zeros(len(files), dtype=int)
    for i, file in enumerate(files):
        image = Image.open(os.path.join(path, file))
        images[i] = np.asarray(image.resize((98, 116))).flatten()
        labels[i] = int(file[7:9])
    return images, labels
```

- PCA

  1. Calculate the average face. (avg_face)
  2. Move data to the center by reduce each face by average face. (diff_train_face)
  3. Calculate the covariance matrix $S=AA^T$. (A = diff_train_face.T)
  4. Calculate the eigenvalues and eigenvectors of the covariance matrix S.
  5. Sort the eigenvalues to pick the first 25 eigenvectors and visualize eigenfaces by transpose and resize the eigenvectors.
  6. Compute weight of test faces for every eigenvector by doing dot product of test faces and eigenvectors. Reconstructing faces by multiply the weight to eigenfaces and added to average face. Random choose 10 numbers as indexes and visualize reconstruction faces of these indexes
  7. Compute weight of training faces and test faces. Use 5-nearest neighbor to do face recognition.



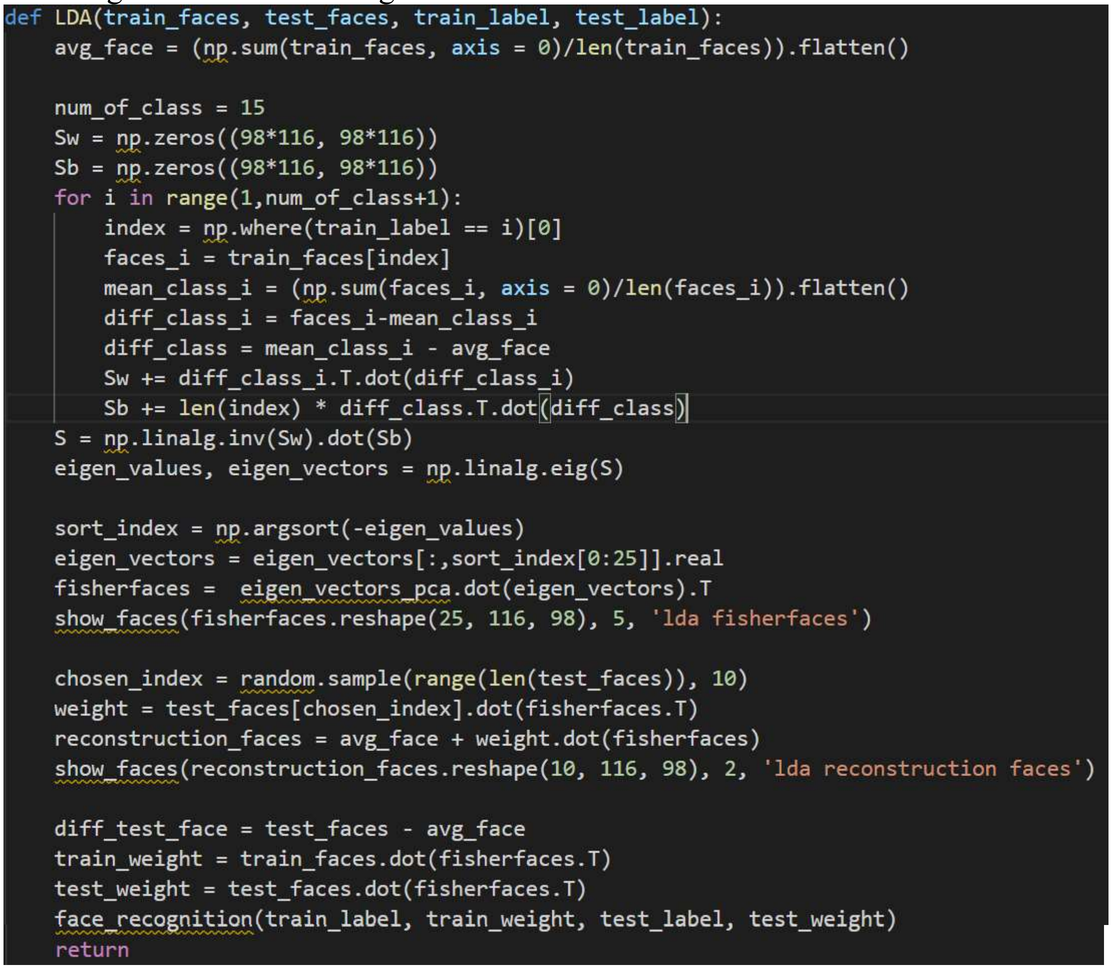
```python
def PCA(train_faces, test_faces, train_label, test_label):
    avg_face = (np.sum(train_faces, axis = 0)/len(train_faces)).flatten()
    diff_train_face = train_faces - avg_face
    S = diff_train_face.T.dot(diff_train_face)
    eigen_values, eigen_vectors = np.linalg.eig(S)

    sort_index = np.argsort(-eigen_values)
    eigen_vectors = eigen_vectors[:,sort_index[0:25]].real
    eigenfaces = eigen_vectors.T
    show_faces(eigenfaces.reshape(25, 116, 98), 5, 'pca eigenfaces')

    chosen_index = random.sample(range(len(test_faces)), 10)
    weight = test_faces[chosen_index].dot(eigen_vectors)
    reconstruction_faces = avg_face + weight.dot(eigenfaces)
    show_faces(reconstruction_faces.reshape(10, 116, 98), 2, 'pca reconstruction faces')

    diff_test_face = test_faces - avg_face
    train_weight = train_faces.dot(eigen_vectors)
    test_weight = test_faces.dot(eigen_vectors)
    face_recognition(train_label, train_weight, test_label, test_weight)
    return
```

- LDA
  1. Calculate the average face. (avg_face)
  2. Calculate the class means of each class. (mean_class_i)
  3. Use the function below to calculate S_W and S_B:

$$\tilde{S}_W = \sum_{j=1}^{k} \sum_{i \in C_j} (\mathrm{y}_i - \tilde{\mathrm{m}}_j)(\mathrm{y}_i - \tilde{\mathrm{m}}_j)^\top :$$

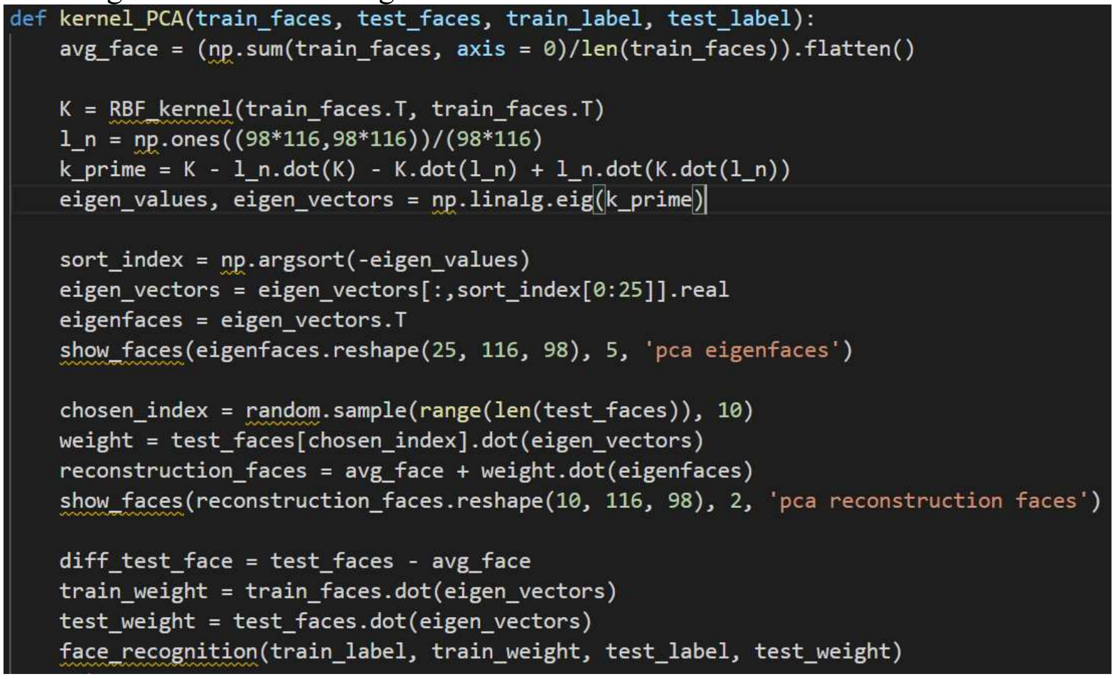
$$\tilde{S}_B = \sum_{j=1}^{k} n_j (\tilde{\mathrm{m}}_j - \tilde{\mathrm{m}})(\tilde{\mathrm{m}}_j - \tilde{\mathrm{m}})^\top$$

     ($y_i - m_j$ = faces_i – mean_class_i = diff_class_i,
     $m_i - m$ = mean_class_i – avg_face = diff_class)
  4. Calculate the covariance matrix $S = S_W^{-1} S_B$.
  5. Calculate the eigenvalues and eigenvectors of the covariance matrix S.
  6. Sort the eigenvalues to pick the first 25 eigenvectors and get fisherfaces by multiply the eigenvectors from PCA with selected eigenvectors. Visualize fisherfaces by transpose and resize the eigenvectors.
  7. Compute weight of test faces for every eigenvector by doing dot product of test faces and eigenvectors. Reconstructing faces by multiply the weight to eigenfaces and added to average face. Random choose 10 numbers as indexes and visualize reconstruction faces of these indexes
  8. Compute weight of training faces and test faces. Use 5-nearest neighbor to do face recognition.



```python
def LDA(train_faces, test_faces, train_label, test_label):
    avg_face = (np.sum(train_faces, axis = 0)/len(train_faces)).flatten()

    num_of_class = 15
    Sw = np.zeros((98*116, 98*116))
    Sb = np.zeros((98*116, 98*116))
    for i in range(1,num_of_class+1):
        index = np.where(train_label == i)[0]
        faces_i = train_faces[index]
        mean_class_i = (np.sum(faces_i, axis = 0)/len(faces_i)).flatten()
        diff_class_i = faces_i-mean_class_i
        diff_class = mean_class_i - avg_face
        Sw += diff_class_i.T.dot(diff_class_i)
        Sb += len(index) * diff_class.T.dot(diff_class)
    S = np.linalg.inv(Sw).dot(Sb)
    eigen_values, eigen_vectors = np.linalg.eig(S)

    sort_index = np.argsort(-eigen_values)
    eigen_vectors = eigen_vectors[:,sort_index[0:25]].real
    fisherfaces =  eigen_vectors_pca.dot(eigen_vectors).T
    show_faces(fisherfaces.reshape(25, 116, 98), 5, 'lda fisherfaces')

    chosen_index = random.sample(range(len(test_faces)), 10)
    weight = test_faces[chosen_index].dot(fisherfaces.T)
    reconstruction_faces = avg_face + weight.dot(fisherfaces)
    show_faces(reconstruction_faces.reshape(10, 116, 98), 2, 'lda reconstruction faces')

    diff_test_face = test_faces - avg_face
    train_weight = train_faces.dot(fisherfaces.T)
    test_weight = test_faces.dot(fisherfaces.T)
    face_recognition(train_label, train_weight, test_label, test_weight)
    return
```

- Kernel PCA
  1. Calculate the average face. (avg_face)
  2. Calculate the RBF kernel. (K)
  3. Center the kernel by the function below:
  $$K' = K - \mathbf{1}_n K - K\mathbf{1}_n + \mathbf{1}_n K \mathbf{1}_n$$
  (k_prime)
  4. Calculate the eigenvalues and eigenvectors of the kernel matrix.
  5. Sort the eigenvalues to pick the first 25 eigenvectors and visualize eigenfaces by transpose and resize the eigenvectors.
  6. Compute weight of test faces for every eigenvector by doing dot product of test faces and eigenvectors. Reconstructing faces by multiply the weight to eigenfaces and added to average face. Random choose 10 numbers as indexes and visualize reconstruction faces of these indexes
  7. Compute weight of training faces and test faces. Use 5-nearest neighbor to do face recognition.

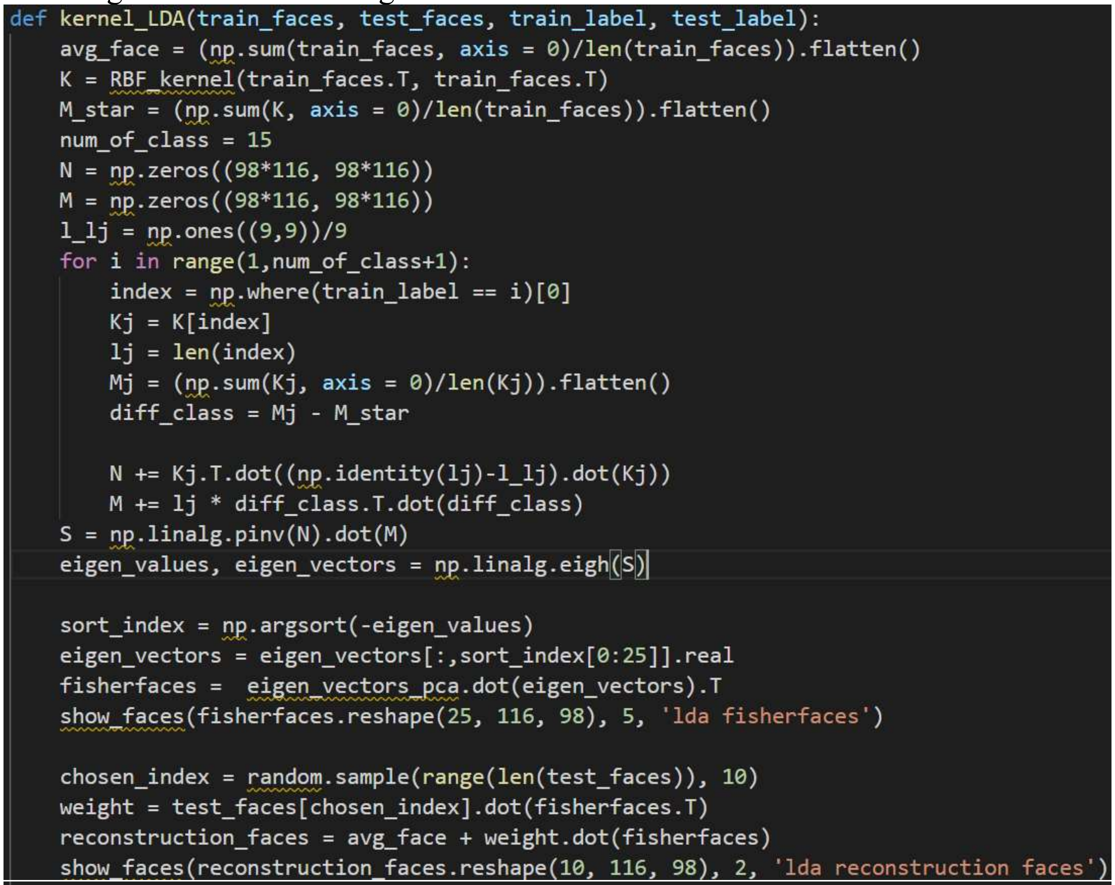


```python
def kernel_PCA(train_faces, test_faces, train_label, test_label):
    avg_face = (np.sum(train_faces, axis = 0)/len(train_faces)).flatten()

    K = RBF_kernel(train_faces.T, train_faces.T)
    l_n = np.ones((98*116,98*116))/(98*116)
    k_prime = K - l_n.dot(K) - K.dot(l_n) + l_n.dot(K.dot(l_n))
    eigen_values, eigen_vectors = np.linalg.eig(k_prime)

    sort_index = np.argsort(-eigen_values)
    eigen_vectors = eigen_vectors[:,sort_index[0:25]].real
    eigenfaces = eigen_vectors.T
    show_faces(eigenfaces.reshape(25, 116, 98), 5, 'pca eigenfaces')

    chosen_index = random.sample(range(len(test_faces)), 10)
    weight = test_faces[chosen_index].dot(eigen_vectors)
    reconstruction_faces = avg_face + weight.dot(eigenfaces)
    show_faces(reconstruction_faces.reshape(10, 116, 98), 2, 'pca reconstruction faces')

    diff_test_face = test_faces - avg_face
    train_weight = train_faces.dot(eigen_vectors)
    test_weight = test_faces.dot(eigen_vectors)
    face_recognition(train_label, train_weight, test_label, test_weight)
```

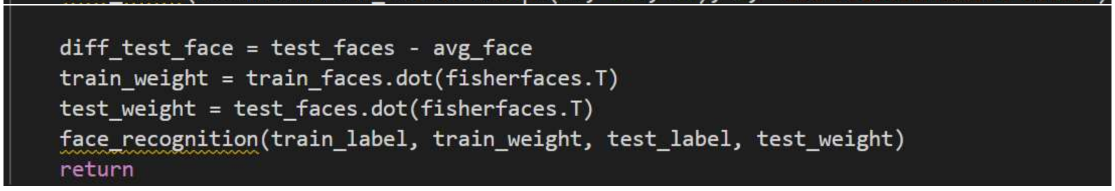
- Kernel LDA
  1. Calculate the average face. (avg_face)
  2. Calculate the RBF kernel. (K)
  3. Use the function below to calculate M and N:
  $$M = \sum_{j=1}^{c} l_j (\mathbf{M}_j - \mathbf{M}_*)(\mathbf{M}_j - \mathbf{M}_*)^{\mathrm{T}}$$
  $$N = \sum_{j=1}^{c} \mathbf{K}_j (\mathbf{I} - \mathbf{1}_{l_j}) \mathbf{K}_j^{\mathrm{T}}.$$
  ($M_j - M_*$ = Mj – M_star = diff_class)
  4. Calculate the covariance matrix $S = N^{-1}M$.
  5. Calculate the eigenvalues and eigenvectors of the covariance matrix S.
  6. Sort the eigenvalues to pick the first 25 eigenvectors and get fisherfaces by multiply the eigenvectors from PCA with selected eigenvectors. Visualize fisherfaces by transpose and resize the

eigenvectors.

7. Compute weight of test faces for every eigenvector by doing dot product of test faces and eigenvectors. Reconstructing faces by multiply the weight to eigenfaces and added to average face. Random choose 10 numbers as indexes and visualize reconstruction faces of these indexes
8. Compute weight of training faces and test faces. Use 5-nearest neighbor to do face recognition.

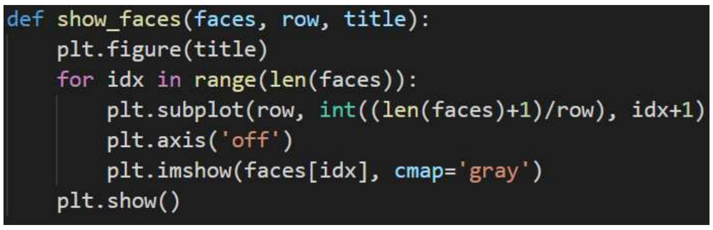


```python
def kernel_LDA(train_faces, test_faces, train_label, test_label):
    avg_face = (np.sum(train_faces, axis = 0)/len(train_faces)).flatten()
    K = RBF_kernel(train_faces.T, train_faces.T)
    M_star = (np.sum(K, axis = 0)/len(train_faces)).flatten()
    num_of_class = 15
    N = np.zeros((98*116, 98*116))
    M = np.zeros((98*116, 98*116))
    l_lj = np.ones((9,9))/9
    for i in range(1,num_of_class+1):
        index = np.where(train_label == i)[0]
        Kj = K[index]
        lj = len(index)
        Mj = (np.sum(Kj, axis = 0)/len(Kj)).flatten()
        diff_class = Mj - M_star

        N += Kj.T.dot((np.identity(lj)-l_lj).dot(Kj))
        M += lj * diff_class.T.dot(diff_class)
    S = np.linalg.pinv(N).dot(M)
    eigen_values, eigen_vectors = np.linalg.eigh(S)

    sort_index = np.argsort(-eigen_values)
    eigen_vectors = eigen_vectors[:,sort_index[0:25]].real
    fisherfaces =  eigen_vectors_pca.dot(eigen_vectors).T
    show_faces(fisherfaces.reshape(25, 116, 98), 5, 'lda fisherfaces')

    chosen_index = random.sample(range(len(test_faces)), 10)
    weight = test_faces[chosen_index].dot(fisherfaces.T)
    reconstruction_faces = avg_face + weight.dot(fisherfaces)
    show_faces(reconstruction_faces.reshape(10, 116, 98), 2, 'lda reconstruction faces')
```



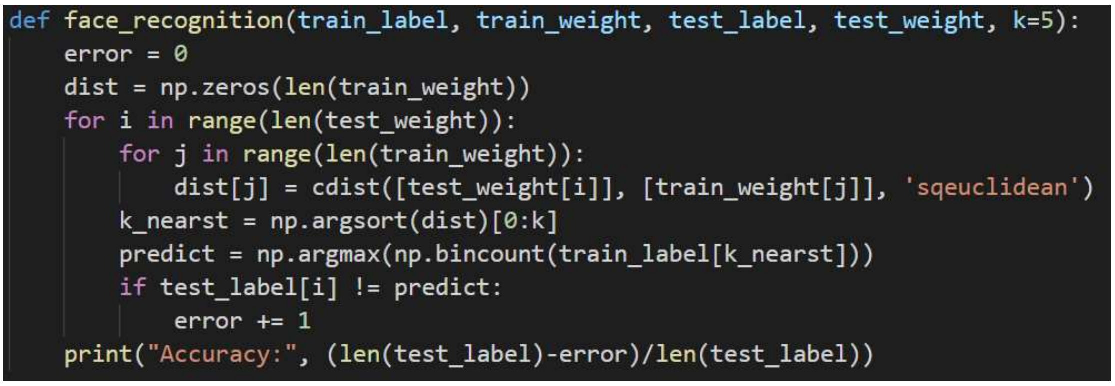
```python
    diff_test_face = test_faces - avg_face
    train_weight = train_faces.dot(fisherfaces.T)
    test_weight = test_faces.dot(fisherfaces.T)
    face_recognition(train_label, train_weight, test_label, test_weight)
    return
```

- Visualize face



```python
def show_faces(faces, row, title):
    plt.figure(title)
    for idx in range(len(faces)):
        plt.subplot(row, int((len(faces)+1)/row), idx+1)
        plt.axis('off')
        plt.imshow(faces[idx], cmap='gray')
    plt.show()
```

- Face recognition
  Calculate the square of Euclidean distance between each pair of test weight and weight, predict by select the most index occurrence from 5 nearest.

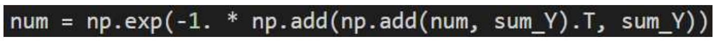


```python
def face_recognition(train_label, train_weight, test_label, test_weight, k=5):
    error = 0
    dist = np.zeros(len(train_weight))
    for i in range(len(test_weight)):
        for j in range(len(train_weight)):
            dist[j] = cdist([test_weight[i]], [train_weight[j]], 'sqeuclidean')
        k_nearst = np.argsort(dist)[0:k]
        predict = np.argmax(np.bincount(train_label[k_nearst]))
        if test_label[i] != predict:
            error += 1
    print("Accuracy:", (len(test_label)-error)/len(test_label))
```

2. t-SNE

   Modify q_{ij} by:

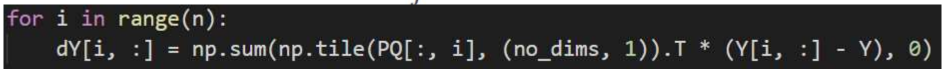
$$q_{ij} = \frac{\exp(-\parallel y_i - y_j \parallel^2)}{\sum_{k\neq l}\exp(-\parallel y_l - y_k \parallel^2)}$$



```python
num = np.exp(-1. * np.add(np.add(num, sum_Y).T, sum_Y))
```

   Modify the gradient by:

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

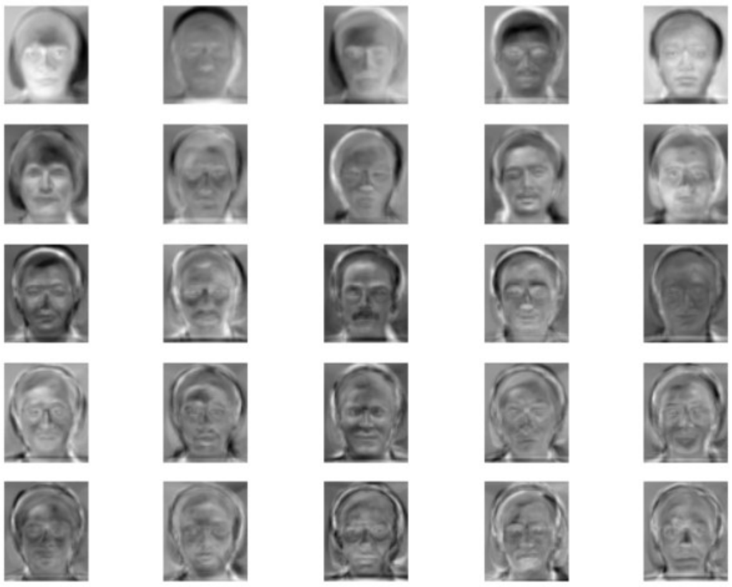


```python
for i in range(n):
    dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

II. Experiments settings and results

1. Kernel Eigenfaces
   - PCA

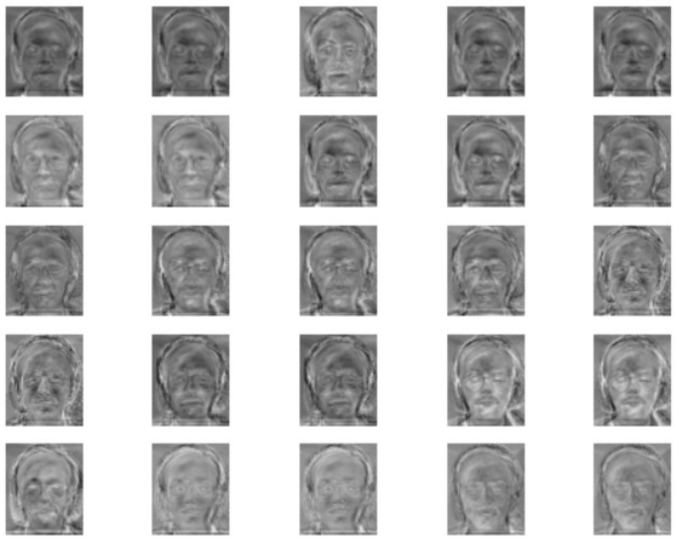
     Eigenfaces:



     Reconstruction faces:

Accuracy: 0.9

- LDA
  Fisherfaces:



Reconstruction faces:

Accuracy: 0.87

- Kernel PCA

  Eigenfaces:

Reconstruction faces:

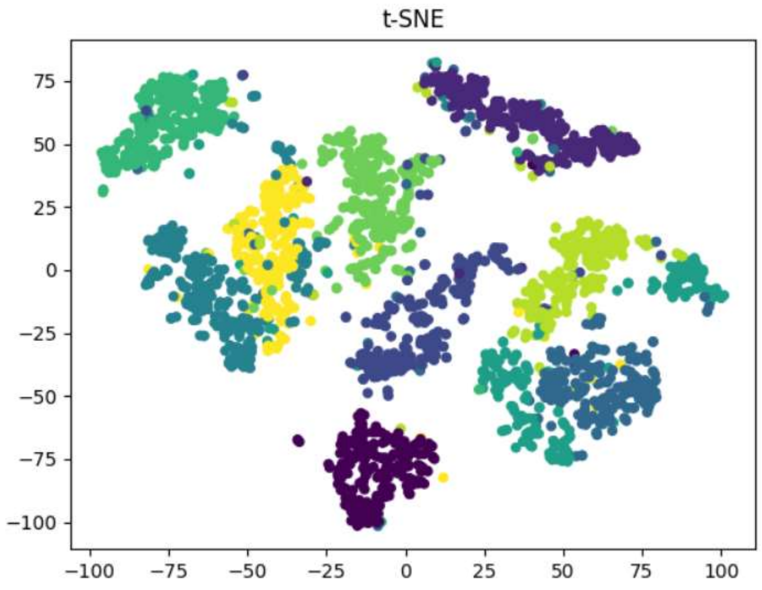
Accuracy: 0.87

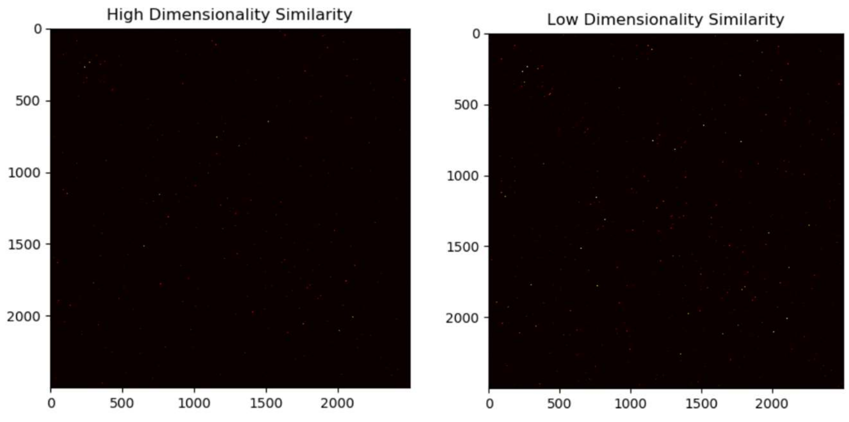
- Kernel LDA
  Eigenfaces:

Reconstruction faces:

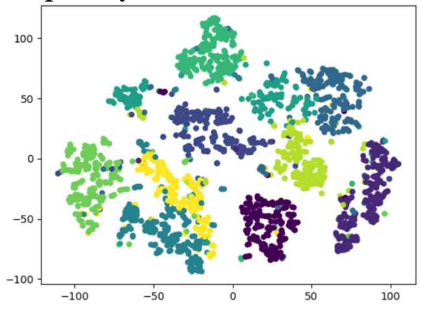
Accuracy: 0.9

2. t-SNE
   - t-SNE

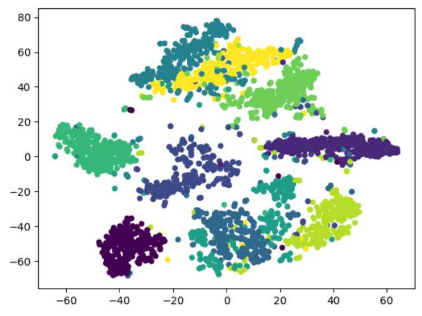




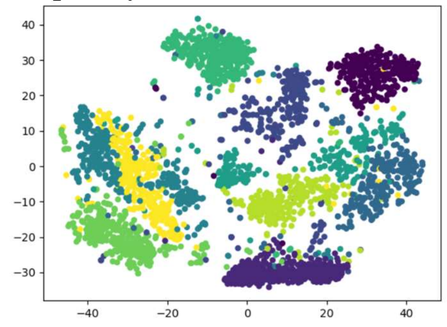
Perplexity = 10



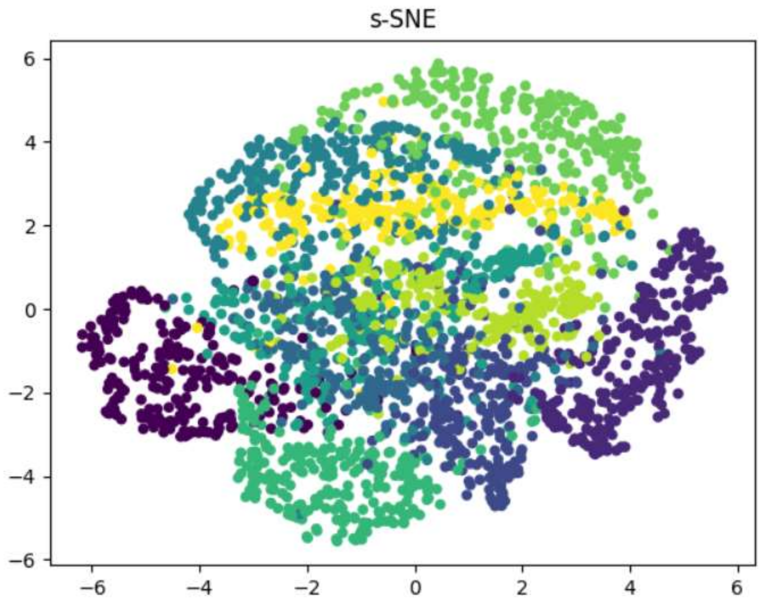
Perplexity = 50

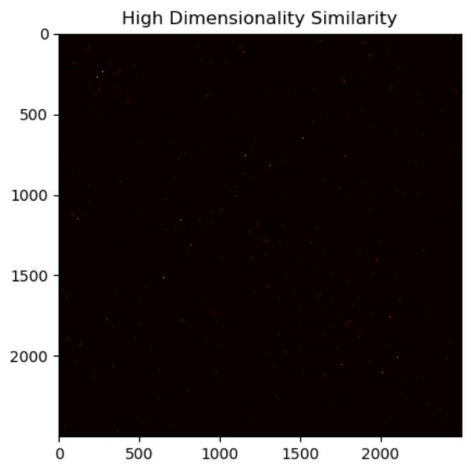


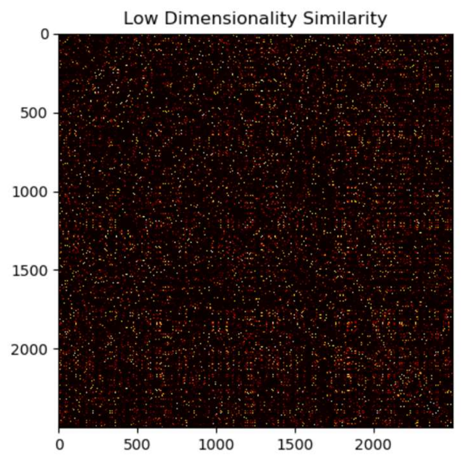
Perplexity = 100



- s-SNE







III. Observations and Discussions

1. Kernel Eigenfaces
    - Dimensionality reduction makes our machine learning model learn faster and more efficient in terms of computational cost. Moreover, this method is to prevent our model suffers from overfitting problem due to its feature selection properties.
    - Due to the insufficient amount of test data, we cannot conclude which face recognition method is better. But from the results of this experiment, PCA performs better than LDA.
    - The main difference between LDA and PCA is that PCA completely processes data for principal component analysis, while LDA handles the difference between categories.
    - Kernel's result is better than non-kernel's just at expected. The standard always finds linear principal components to represent the data in lower dimension. Sometime, we need non-linear principal components. If we apply standard for the below data, it will fail to find good representative direction. Kernel rectifies this limitation.
    - The computational complexity for Kernel to extract principal components take more time compared to Standard.
2. t-SNE
    - The clusters of t-SNE are scattered that Symmetric SNE bounds into crowded problem.
    - t-SNE can quickly separate different classes into different positions, and then modify the distribution in a more general way. Symmetric SNE only moves data slightly in low-dimensional space.
    - When the perplexity increases, points will be more compact in every class.