ML Homework 7 Report

- Student Info
    1. Student ID: 310555024
    2. Student Name: 林廷翰

- Code
    1. Kernel Eigenfaces
        - Common Parts
            1. Parse Arguments

```python
# Set up the input parameters, and return args.
def parseArguments():
    parse = argparse.ArgumentParser()

    # The algorithm will be used, 0 -> PCA, 1-> LDA.
    parse.add_argument('--algo', default=0)
    # Mode of PCA and LDA, 0 -> simple, 1 -> kernel.
    parse.add_argument('--mode', default=0)
    # The number of nearest neighbors used for classification.
    parse.add_argument('--numOfNeighbors', default=5)
    # The kernel type, 0 -> linear, 1 -> RBF.
    parse.add_argument('--kernelType', default=0)
    # The gamma of RBF kernel.
    parse.add_argument('--gamma', default=0.000001)

    return parse.parse_args()
```

The objective of the `parseArguments` function is to parse all the necessary input parameters of all scenarios.

            2. Load Data

```python
def readTrainingImages():
    trainingImages, trainingLabels = None, None
    numOfImages = 0

    # Get the number of images first.
    with os.scandir(f'{imageDirectory}/Training') as directory:
        # Get number of files
        numOfImages = len([file for file in directory if file.is_file()])

    # Read the files.
    with os.scandir(f'{imageDirectory}/Training') as directory:
        trainingLabels = np.zeros(numOfImages, dtype=int)
        # Images will be resized to 29 * 24.
        trainingImages = np.zeros((numOfImages, 29 * 24))

        for index, file in enumerate(directory):
            if file.path.endswith('.pgm') and file.is_file():
                face = np.asarray(Image.open(file.path).resize((24, 29))).reshape(1, -1)
                trainingImages[index, :] = face
                trainingLabels[index] = int(file.name[7:9])

    return trainingImages, trainingLabels
```

```python
def readTestingImages():
    testingImages, testingLabels = None, None
    numOfImages = 0

    # Get the number of images first.
    with os.scandir(f'{imageDirectory}/Testing') as directory:
        # Get number of files
        numOfImages = len([file for file in directory if file.is_file()])

    # Read the files.
    with os.scandir(f'{imageDirectory}/Testing') as directory:
        testingLabels = np.zeros(numOfImages, dtype=int)
        # Images will be resized to 29 * 24.
        testingImages = np.zeros((numOfImages, 29 * 24))

        for index, file in enumerate(directory):
            if file.path.endswith('.pgm') and file.is_file():
                face = np.asarray(Image.open(file.path).resize((24, 29))).reshape(1, -1)
                testingImages[index, :] = face
                testingLabels[index] = int(file.name[7:9])

    return testingImages, testingLabels
```

I used two functions (`readTrainingData` & `readTestingData`) to load the data from corresponding input files.

- Part 1 (PCA, LDA → eigenfaces & fisherfaces, reconstruction)

  1. PCA

     - Overview

```python
# Principal components analysis.
def PCA(mode, numOfNeighbors, kernelType, gamma, trainingImages, trainingLabels, testingImages, testingLabels):
    # Get the number of training images.
    numOfTrainingImages = len(trainingImages)
    numOfTestingImages = len(testingImages)

    # Simple PCA
    if mode == 0:
        matrix = simplePCA(numOfTrainingImages, trainingImages)
    # Kernel PCA
    else:
        matrix = kernelPCA(trainingImages, kernelType, gamma)

    # Find the first 25 largest eigenvectors.
    targetEigenvectors = findTargetEigenvectors(matrix)

    # Transform eigenvectors into eigenfaces.
    transformEigenvectorsToFaces(targetEigenvectors, 0)

    # Randomly reconstruct 10 eigenfaces.
    reconstructFaces(numOfTrainingImages, trainingImages, targetEigenvectors)

    # Classify and predict.
    classifyAndPredict(numOfTrainingImages, numOfTestingImages, trainingImages, trainingLabels, testingImages,
                       testingLabels, targetEigenvectors, numOfNeighbors)

    # Output the diagram.
    outputDiagram()
```

I tried to find the projection W and find the 25 eigenvectors with the largest eigenvalues. After that, I transformed eigenvectors into faces (eigenfaces & fisherfaces), and then reconstruct 10 randomly chosen images.

- Find Projection W (PCA)

```python
def simplePCA(numOfTrainingImages, trainingImages):
    # Compute covariance
    trainingImagesTransposed = trainingImages.T
    mean = np.mean(trainingImagesTransposed, axis=1)
    mean = np.tile(mean.T, (numOfTrainingImages, 1)).T
    difference = trainingImagesTransposed - mean
    covariance = difference.dot(difference.T) / numOfTrainingImages

    return covariance
```

For simple PCA, I just computed the covariance of training images.

2. LDA

- Overview

```python
#Linear discriminative analysis.
def LDA(mode, numOfNeighbors, kernelType, gamma, trainingImages, trainingLabels, testingImages, testingLabels):
    # Get number of each class and the number of training images.
    _, numOfEachClass = np.unique(trainingLabels, return_counts=True)
    numOfTrainingImages = len(trainingImages)
    numOfTestingImages = len(testingImages)

    # Simple LDA
    if not mode:
        matrix = simpleLDA(numOfEachClass, trainingImages, trainingLabels)
    # Kernel LDA
    else:
        matrix = kernelLDA(numOfEachClass, trainingImages, trainingLabels, kernelType, gamma)

    # Find the first 25 largest eigenvectors.
    targetEigenvectors = findTargetEigenvectors(matrix)

    # Transform eigenvectors into eigenfaces.
    transformEigenvectorsToFaces(targetEigenvectors, 1)

    # Randomly reconstruct 10 eigenfaces.
    reconstructFaces(numOfTrainingImages, trainingImages, targetEigenvectors)

    # Classify and predict.
    classifyAndPredict(numOfTrainingImages, numOfTestingImages, trainingImages, trainingLabels, testingImages,
                       testingLabels, targetEigenvectors, numOfNeighbors)

    # Output the diagram.
```

I tried to find the projection W and find the 25 eigenvectors with the largest eigenvalues. After that, I transformed eigenvectors into faces (eigenfaces & fisherfaces), and then reconstruct 10 randomly chosen images.

- Find Projection W (LDA)

```python
def simpleLDA(numOfEachClass, trainingImages, trainingLabels):
    # Compute the overall mean.
    overallMean = np.mean(trainingImages, axis=0)

    # Get mean of each class.
    numOfClasses = len(numOfEachClass)
    classMean = np.zeros((numOfClasses, 29 * 24))

    for label in range(numOfClasses):
        classMean[label, :] = np.mean(trainingImages[trainingLabels == label + 1], axis=0)

    # Compute between-class scatter.
    scatterB = np.zeros((29 * 24, 29 * 24), dtype=float)

    for idx, num in enumerate(numOfEachClass):
        difference = (classMean[idx] - overallMean).reshape((29 * 24, 1))
        scatterB += num * difference.dot(difference.T)

    # Compute within-class scatter.
    scatterW = np.zeros((29 * 24, 29 * 24), dtype=float)
    for idx, mean in enumerate(classMean):
        difference = trainingImages[trainingLabels == idx + 1] - mean
        scatterW += difference.T.dot(difference)

    # Compute Sw^(-1) * Sb.
    matrix = np.linalg.pinv(scatterW).dot(scatterB)

    return matrix
```

For simple LDA, I computed the between-class scatter and within-class scatter according to the following two formulas:

1. Between-Class Scatter

$$\textit{between-class scatter}:$$

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

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

2. Within-Class Scatter

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

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

Then, for simple LDA, I computed the projection W based on between-class scatter and within-class scatter, as the following formula:

$$S_W^{-1} S_B \text{ as } W$$

3. Common Parts

- Find Eigenvectors

```python
def findTargetEigenvectors(matrix):
    # Compute eigenvalues and eigenvectors.
    eigenvalues, eigenvectors = np.linalg.eig(matrix)

    # Get 25 first largest eigenvectors.
    targetIndex = np.argsort(eigenvalues)[::-1][:25]
    targetEigenvectors = eigenvectors[:, targetIndex].real

    return targetEigenvectors
```

I computed all of the eigenvectors, and just chose the 25 first eigenvectors with the largest eigenvalues.

- Transform Eigenvectors to Faces

```python
# Transform eigenvectors into eigenfaces/fisherfaces.
# algo parameter means the algorithm been used, 0 -> PCA, 1-> LDA.
def transformEigenvectorsToFaces(targetEigenvectors, algo):
    faces = targetEigenvectors.T.reshape((25, 29, 24))
    fig = plt.figure(1)
    fig.canvas.set_window_title(f'{"Eigenfaces" if algo == 0 else "Fisherfaces"}')

    for idx in range(25):
        plt.subplot(5, 5, idx + 1)
        plt.axis('off')
        plt.imshow(faces[idx, :, :], cmap='gray')
```

I reshaped the eigenvectors for displaying them as eigenfaces or fisherfaces.

- Face Reconstruction

```python
#Reconstruct the faces from eigenfaces and fisherfaces.
def reconstructFaces(numOfTrainingImages, trainingImages, targetEigenvectors):
    reconstructedImages = np.zeros((10, 29 * 24))
    choice = np.random.choice(numOfTrainingImages, 10)

    for index in range(10):
        reconstructedImages[index, :] = trainingImages[choice[index], :].dot(targetEigenvectors).dot(
            targetEigenvectors.T)

    fig = plt.figure(2)
    fig.canvas.set_window_title('Reconstructed faces')

    for index in range(10):
        # Original image.
        plt.subplot(10, 2, index * 2 + 1)
        plt.axis('off')
        plt.imshow(trainingImages[choice[index], :].reshape((29, 24)), cmap='gray')

        # Reconstructed image.
        plt.subplot(10, 2, index * 2 + 2)
        plt.axis('off')
        plt.imshow(reconstructedImages[index, :].reshape((29, 24)), cmap='gray')
```

I randomly chose 10 images from training images, and reconstructed the faces based on the following formula:

$$\underbrace{xW}_{z}\underline{W}^{\top}$$

- Part 2 (Compute the performance)

  1. Classify and Predict

```python
# Classify and show predict result.
def classifyAndPredict(numOfTrainingImages, numOfTestingImages, trainingImages, trainingLabels, testingImages,
                       testingLabels,
                       targetEigenvectors, numOfNeighbors):
    decorrelatedTraining = decorrelate(numOfTrainingImages, trainingImages, targetEigenvectors)
    decorrelatedTesting = decorrelate(numOfTestingImages, testingImages, targetEigenvectors)
    error = 0
    distance = np.zeros(numOfTrainingImages)

    for testIndex, test in enumerate(decorrelatedTesting):
        for trainIndex, train in enumerate(decorrelatedTraining):
            distance[trainIndex] = np.linalg.norm(test - train)

        minDistances = np.argsort(distance)[:numOfNeighbors]
        predict = np.argmax(np.bincount(trainingLabels[minDistances]))

        if predict != testingLabels[testIndex]:
            error += 1
    print(f'Error count: {error}\nError rate: {float(error) / numOfTestingImages}')
```

The training and testing images are first decorrelated by eigenvectors. And then, I used k nearest neighbors to decide the class of each testing image.

```python
# Decorrelate original images into components space.
def decorrelate(numOfImages, images, eigenvectors):
    decorrelatedImages = np.zeros((numOfImages, 25))

    for index, image in enumerate(images):
        decorrelatedImages[index, :] = image.dot(eigenvectors)

    return decorrelatedImages
```

This is the decorrelate function, it is based on the following formula:

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$$

- Part 3 (kernel PCA, kernel LDA (diff kernels) vs. PCA, LDA)

  1. PCA

     - Overview → It is same as the part 1.
     - Find Projection W (kernel PCA)

```python
def kernelPCA(trainingImages, kernelType, gamma):
    # Compute kernel.
    # Linear
    if kernelType == 0:
        kernel = trainingImages.T.dot(trainingImages)
    # RBF
    else:
        kernel = np.exp(-gamma * cdist(trainingImages.T, trainingImages.T, 'sqeuclidean'))

    # Get centered kernel.
    matrixN = np.ones((29 * 24, 29 * 24), dtype=float) / (29 * 24)
    matrix = kernel - matrixN.dot(kernel) - kernel.dot(matrixN) + matrixN.dot(kernel).dot(matrixN)

    return matrix
```

I computed the gram matrix first (linear and RBF kernel), and then computed the matrix K based on the following formula:

$$K^C = K - \mathbf{1}_N K - K \mathbf{1}_N + \mathbf{1}_N K \mathbf{1}_N$$

2. LDA

- Overview → It is same as the part 1.
- Find Projection W (kernel LDA)

```python
def kernelLDA(numOfEachClass, trainingImages, trainingLabels, kernelType, gamma):
    # Compute kernel.
    numOfClasses = len(numOfEachClass)
    numOfImages = len(trainingImages)

    if not kernelType:
        # Linear
        kernelOfEachClass = np.zeros((numOfClasses, 29 * 24, 29 * 24))
        for idx in range(numOfClasses):
            images = trainingImages[trainingLabels == idx + 1]
            kernelOfEachClass[idx] = images.T.dot(images)
        kernelOfAll = trainingImages.T.dot(trainingImages)
    else:
        # RBF
        kernelOfEachClass = np.zeros((numOfClasses, 29 * 24, 29 * 24))
        for idx in range(numOfClasses):
            images = trainingImages[trainingLabels == idx + 1]
            kernelOfEachClass[idx] = np.exp(-gamma * cdist(images.T, images.T, 'sqeuclidean'))
        kernelOfAll = np.exp(-gamma * cdist(trainingImages.T, trainingImages.T, 'sqeuclidean'))

    # Compute N.
    matrixN = np.zeros((29 * 24, 29 * 24))
    identityMatrix = np.eye(29 * 24)

    for index, num in enumerate(numOfEachClass):
        matrixN += kernelOfEachClass[index].dot(identityMatrix - num * identityMatrix).dot(
            kernelOfEachClass[idx].T)

```

```python
    # Compute M.
    matrixMI = np.zeros((numOfClasses, 29 * 24))

    for index, kernel in enumerate(kernelOfEachClass):
        for rowIndex, row in enumerate(kernel):
            matrixMI[index, rowIndex] = np.sum(row) / numOfEachClass[idx]
    matrixMStar = np.zeros(29 * 24)
    for index, row in enumerate(kernelOfAll):
        matrixMStar[index] = np.sum(row) / numOfImages
    matrixM = np.zeros((29 * 24, 29 * 24))
    for idx, num in enumerate(numOfEachClass):
        difference = (matrixMI[idx] - matrixMStar).reshape((29 * 24, 1))
        matrixM += num * difference.dot(difference.T)

    # Get N^(-1) * M.
    matrix = np.linalg.pinv(matrixN).dot(matrixM)

    return matrix
```

For kernel LDA, I computed the matrix N and matrix M based on the following formulas:

$$M = \sum_{j=1}^{c} l_j (\mathbf{M}_j - \mathbf{M}_*)(\mathbf{M}_j - \mathbf{M}_*)^{\mathrm{T}}$$

$$N = \sum_{j=1}^{c} \mathbf{K}_j (\mathbf{I} - \mathbf{1}_{l_j}) \mathbf{K}_j^{\mathrm{T}}.$$

$$(\mathbf{M}_*)_j = \frac{1}{l} \sum_{k=1}^{l} k(\mathbf{x}_j, \mathbf{x}_k).$$

And then, I computed the projection W based on the following formula:

$$\mathbf{N}^{-1}\mathbf{M}.$$

3. Find Eigenvectors → It is same as the part 1.
4. Transform Eigenvectors to Faces → It is same as the part 1.
5. Face Reconstruction → It is same as the part 1.
6. Classify and Predict → It is same as the part 2.

2. t-SNE

- Common Parts

1. Parse Arguments

```python
def parseArguments():
    parse = argparse.ArgumentParser()

    # Mode for SNE, 0 -> t-SNE, 1 -> symmetric SNE.
    parse.add_argument('--mode', default=0)
    parse.add_argument('--perplexity', default=20.0)

    return parse.parse_args()
```

The objective of the `parseArguments` function is to parse all the necessary input parameters of all scenarios.

2. Load Data

```python
def readInputFile():
    x = np.loadtxt(file_path_of_image)
    label_of_x = np.loadtxt(file_path_of_label)

    return x, label_of_x
```

I used `readInputFile` to load the data from input files.

- Part 1 (symmetric SNE vs. t-SNE, modify code)

1. Original (t-SNE)

```
        num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
```

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

Originally, the program computed the q and gradients based on the following formulas:

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

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

2. Add Symmetric SNE Support

```
    if mode == 0:
        num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
    # symmetric SNE
    else:
        num = np.exp(-1. * np.add(np.add(num, sum_Y).T, sum_Y))
```

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

We used the input parameter - mode to control the scenario (symmetric SNE or t-SNE), and we computed the q and gradients of symmetric SNE based on the following formulas:

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

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

- Part 2 (visualize the embedding)

```python
        # Compute current value of cost function
        if (iteration + 1) % 10 == 0:
            C = np.sum(P * np.log(P / Q))
            print("Iteration %d: error is %f" % (iteration + 1, C))
            image.append(captureState(Y, labels, mode, perplexity))
```

```python
def captureState(Y, labels, mode, perplexity):
    plt.clf()
    plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
    plt.title(f'{"t-SNE" if not mode else "symmetric SNE"}, perplexity = {perplexity}')
    plt.tight_layout()
    canvas = plt.get_current_fig_manager().canvas
    canvas.draw()

    return Image.frombytes('RGB', canvas.get_width_height(), canvas.tostring_rgb())
```

```python
    # Save gif
    filename = f'./output/{"t-SNE" if not mode else "symmetric-SNE"}_{perplexity}.gif'
    os.makedirs(os.path.dirname(filename), exist_ok=True)
    image[0].save(filename, save_all=True, append_images=image[1:], optimize=False, loop=0, duration=200)
```

I captured the state every 10 iterations and stored it to an array. Finally, I output the GIF file based on the record array.

- Part 3 (visualize the distribution of pairwise similarities)

```python
    # Plot pairwise similarities in high-dimensional space and low-dimensional space
    drawSimilarities(P, Q, labels)
```

```python
def drawSimilarities(p, q, labels):
    # Get sorted index.
    index = np.argsort(labels)
    plt.clf()
    plt.figure(1)

    # Plot p.
    log_p = np.log(p)
    sorted_p = log_p[index][:, index]
    plt.subplot(121)
    img = plt.imshow(sorted_p, cmap='gray', vmin=np.min(log_p), vmax=np.max(log_p))
    plt.colorbar(img)
    plt.title('High dim space')

    # Plot q.
    log_q = np.log(q)
    sorted_q = log_q[index][:, index]
    plt.subplot(122)
    img = plt.imshow(sorted_q, cmap='gray', vmin=np.min(log_q), vmax=np.max(log_q))
    plt.colorbar(img)
    plt.title('Low dim space')

    plt.tight_layout()
```

After finishing all iterations, I will output the similarities in high and low dimensions.

- Part 4 (try different perplexity values)

```
def main():
    args = parseArguments()
    # Get parameters.
    mode = int(args.mode)
    perplexity = float(args.perplexity)

    x, label_of_x = readInputFile()

    y = sne(x, label_of_x, mode, 2, 50, perplexity)
```

I set perplexity as an input parameter, so we can try different perplexities to meet our different experiments.

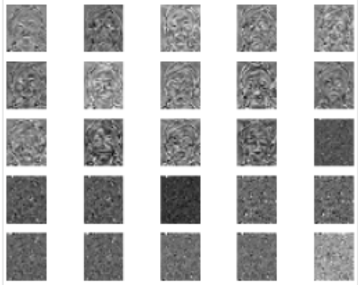
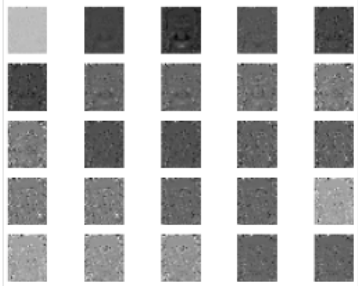
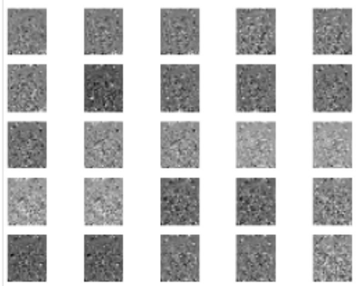
- Experiments & Discussion

  1. Kernel Eigenfaces

     - PCA

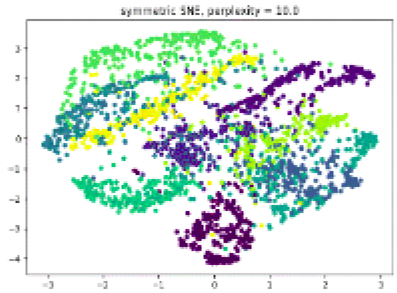
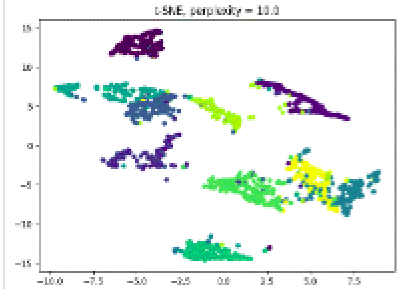
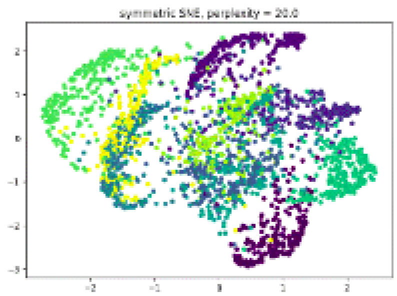
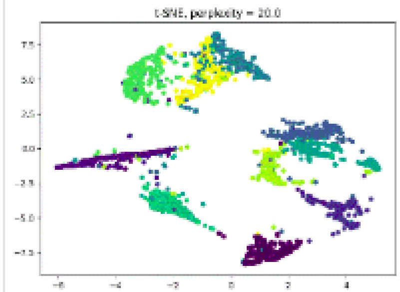
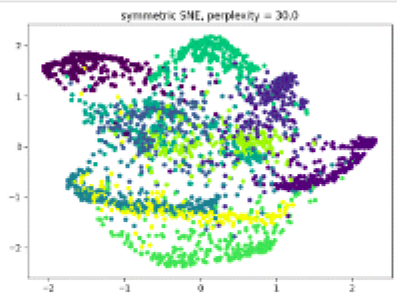
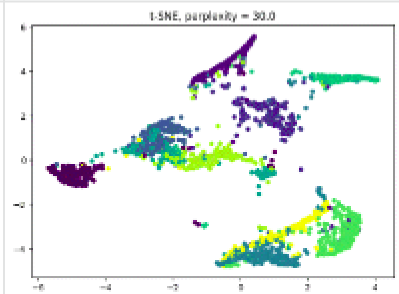
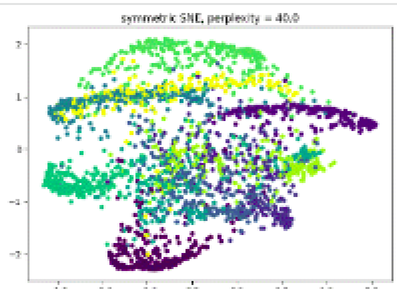
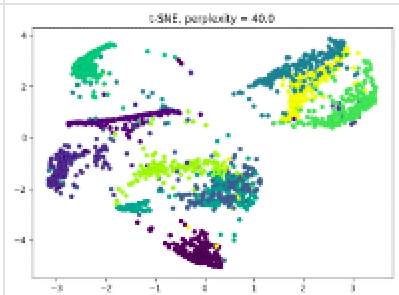
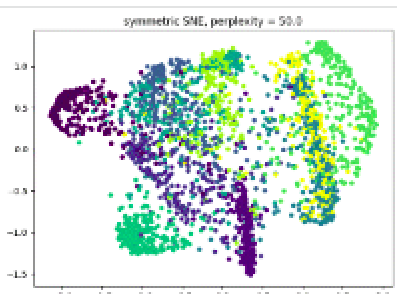
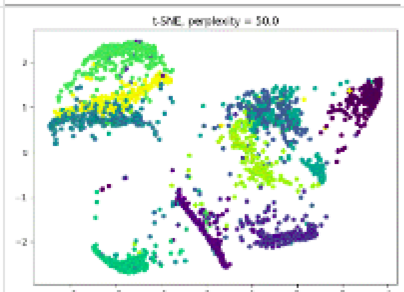
	Eigenfaces	Reconstruction	Performance
Simple			Error count: 4 Error rate: 0.13333333333333333
Linear Kernel			Error count: 4 Error rate: 0.13333333333333333
RBF Kernel			Error count: 4 Error rate: 0.13333333333333333

- LDA

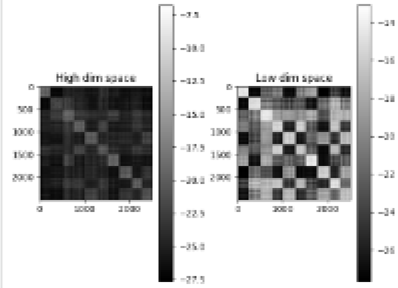
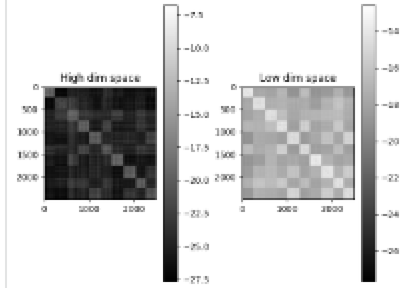
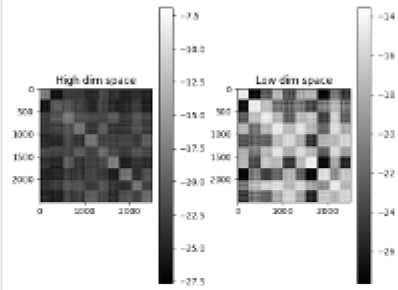
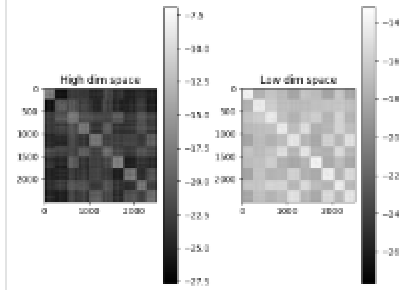
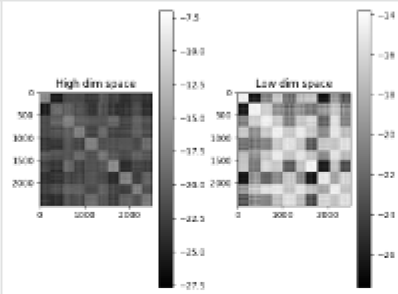
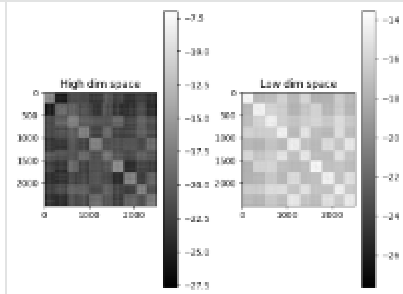
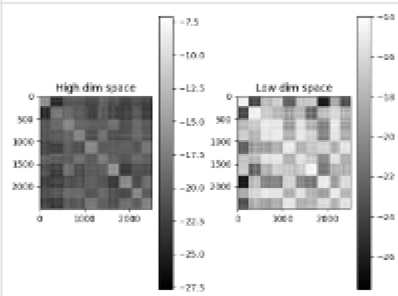
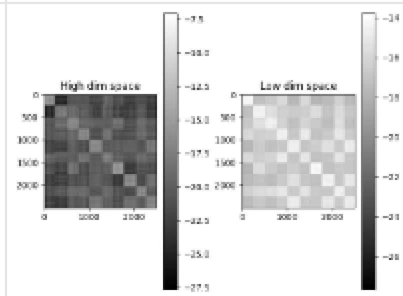
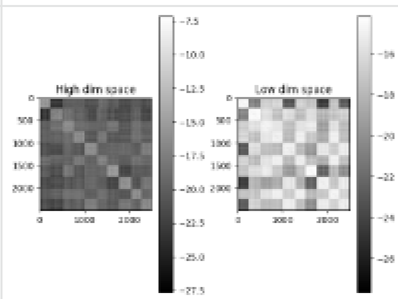
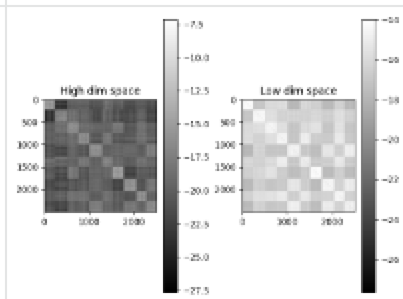
	Fisherfaces	Reconstruction	Performance
Simple			Error count: 1 Error rate: 0.03333333333333333
Linear Kernel			Error count: 19 Error rate: 0.6333333333333333
RBF Kernel			Error count: 8 Error rate: 0.26666666666666666

2. t-SNE

- Embedding

Perplexity	Symmetric SNE	t-SNE
10		
20		
30		
40		
50		

- Pairwise Similarities

Perplexity	Symmetric SNE	t-SNE
10		
20		
30		
40		
50		

- Observations
  1. Kernel Eigenfaces
     - The output of fisher faces are a little bit strange. It’s not as intuitive as the output of eigenfaces.
     - The reconstruction of LDA is not very clear, but the reconstruction of PCA is very clear.
     - The objective of eigenfaces is that we try to output the features of each person. And then we can classify the testing images to corresponding class based on the eigenfaces.
  2. t-SNE
     - The training speed of symmetric SNE is faster that t-SNE.
     - Based on the embedding result, we can clearly find that t-SNE can separate the low dimension point more clearly.
     - Perplexity is the number of neighbors to be used, we can find that the larger perplexity lead to less sensitive to small group.