ML HW07 Report

a. Code

- Kernel Eigenfaces
  - Part1. Use PCA and LDA to show the first 25 eigenfaces and fisherfaces, and randomly pick 10 images to show their reconstruction

First, I load the PGM file with following code. And I use `re` to extract subject of the image from the filename. I resize the image to 50 * 50 since it may cost lots of time to compute the inverse, covariance and kernel matrix of big matrix

```
In [4]: def readPGM(filename):
            image = Image.open(filename)
            image = image.resize(SHAPE, Image.ANTIALIAS)
            image = np.array(image)
            label = int(re.findall(r'subject(\d+)', filename)[0])
            return [image.ravel().astype(np.float64), label]
```

```
In [5]: def readData(path):
            data = []
            filename = []
            label = []
            for pgm in os.listdir(path):
                res = readPGM(f'{path}/{pgm}')
                data.append(res[0])
                filename.append(pgm)
                label.append(res[1])
            return [np.asarray(data), np.asarray(filename), np.asarray(label)]
```

PCA eigenface

We use PCA to compute the eigenfaces. The goal of PCA is to find a orthogonal projection `W` in which the data `x` after projection `z = Wx` will have maximum variance. We can achieve this by solving the eigenvalue problem of the covariance matrix. For this assignment, we want to find the first 25 eigenfaces, so the orthogonal projection `W` is composed of 25 first largest eigenvectors of covariance matrix.

The following code is the implementation of PCA. We compute the covariance matrix first, then compute the eigenvalues and eigenvectors. Finally, we can obtain the projection matrix W.

```
In [6]: def PCA(X, dims):
            mu = np.mean(X, axis=0)
            cov = (X - mu) @ (X - mu).T
            eigen_val, eigen_vec = np.linalg.eigh(cov)
            eigen_vec = (X - mu).T @ eigen_vec
            for i in range(eigen_vec.shape[1]):
                eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
            idx = np.argsort(eigen_val)[::-1]
            W = eigen_vec[:, idx][:, :dims].real
            return [W, mu]
```

LDA fisherface

We use LDA to compute the fisherfaces. The purpose of LDA is to maximize the ratio of between-classes to within-classes scatter,

$$Scatter_{between-classes} = S_b = \sum_{i=1}^{c} N_i(\mu_i - \mu)(\mu_i - \mu)^T$$
$$Scatter_{within-classes} = S_w = \sum_{i=1}^{c} \sum_{x_j \in X_c} (x_j - \mu_i)(x_j - \mu_i)^T$$

where `mu` is the total mean and `mu_i` is the mean of class `i`.

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i$$
$$\mu_i = \frac{1}{|X_i|} \sum_{x_j \in X_i} x_j, i \in \{1, ..., c\}$$

And the optimal projection `W` can be found by:

$$argmax_W \frac{|W^T S_B W|}{|W^T S_W W|}$$

A solution is given by solving the eigenvalue problem:

$$S_W^{-1} S_B v_i = \lambda_i v_i$$

Implementation of LDA. First, compute the scatter matrix(`Sb`, `Sw`). Then, solve the eigenvalue problem and obtain the projection matrix with `dims` first largest eigenvectors.

```
In [7]: def LDA(X, label, dims):
            (n, d) = X.shape
            label = np.asarray(label)
            c = np.unique(label)
            mu = np.mean(X, axis=0)
            S_w = np.zeros((d, d), dtype=np.float64)
            S_b = np.zeros((d, d), dtype=np.float64)
            for i in c:
                X_i = X[np.where(label == i)[0], :]
                mu_i = np.mean(X_i, axis=0)
                S_w += (X_i - mu_i).T @ (X_i - mu_i)
                S_b += X_i.shape[0] * ((mu_i - mu).T @ (mu_i - mu))
            eigen_val, eigen_vec = np.linalg.eig(np.linalg.pinv(S_w) @ S_b)
            for i in range(eigen_vec.shape[1]):
                eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
            idx = np.argsort(eigen_val)[::-1]
            W = eigen_vec[:, idx][:, :dims].real
            return W
```

- Part2. Use PCA and LDA to do face recognition, and compute the performance. You should use k nearest neighbor to classify which subject the testing image belongs to.
  We simply use the function(`PCA`, `LDA`) in last part to compute the projection matrix `W`, and project the training data(`X`) and testing data(`test`) to lower dimension first with the following code. Here we pick the first 25 eigenvectors to compose `W` for both PCA and LDA.

```
W, mu = PCA(data, 25)
X_proj = (X - mu) @ W
test_proj = (test - mu) @ W
faceRecognition(X_proj, X_label, test_proj, test_label, 'PCA')

W = LDA(data, label, 25)
X_proj = X @ W
test_proj = test @ W
faceRecognition(X_proj, X_label, test_proj, test_label, 'LDA')
```

do the face recognition with KNN. We compute the distance(`D`) matrix between testing data and training data first. Assume there are `m` testing data and `n` training data. The size of distance matrix is `m * n`, and sort each row in ascending order. By extracting the first `k` neighbors and inspecting their labels, we can assign each testing data to a class. The code below is the implementation of face recognition.

```
In [15]: def faceRecognition(X, X_label, test, test_label, method, kernel_type=None):
             if kernel_type is None:
                 print(f'Face recognition with {method} and KNN:')
             else:
                 print(f'Face recognition with Kernel {method}({kernels[kernel_type - 1]}) and KNN:')
             dist_mat = []
             for i in range(test.shape[0]):
                 dist = []
                 for j in range(X.shape[0]):
                     dist.append((distance(X[j], test[i]), X_label[j]))
                 dist.sort(key=lambda x: x[0])
                 dist_mat.append(dist)
             for k in K:
                 correct = 0
                 total = test.shape[0]
                 for i in range(test.shape[0]):
                     dist = dist_mat[i]
                     neighbor = np.asarray([x[1] for x in dist[:k]])
                     neighbor, count = np.unique(neighbor, return_counts=True)
                     predict = neighbor[np.argmax(count)]
                     if predict == test_label[i]:
                         correct += 1
                 print(f'K={k:>2}, accuracy: {correct / total:>.3f} ({correct}/{total})')
             print()
```

- Part3. Use kernel PCA and kernel LDA

Kernel PCA

The main differences between kernel PCA and PCA is that, instead of computing the covariance matrix of the pixels, we use kernel function to represent the relation between the images.

we have to centralize the kernel since centered data is required to perform an effective PCA. Below is the formula for centralization, where `1N` is a `N * N` matrix for which each element takes value `1/N`.

$$K^C = K - 1_N K - K 1_N + 1_N K 1_N$$

Implementation of kernel PCA. Here I use three different kinds of kernel to perform the recognition, which are linear kernel, polynomial kernel and radial basis function kernel. After solving the eigenvalue problem of the kernel matrix, the product of the kernel and eigenvectors is the projection of the data in feature space. And we can use `dims` to decide the size of our principle components. I set `dims` to 25 here.

In [11]:
```
def getKernel(X, kernel_type):
    if kernel_type == 1:
        kernel = linearKernel(X)
    elif kernel_type == 2:
        kernel = polynomialKernel(X, 5, 10, 2)
    else:
        kernel = rbfKernel(X, 1e-7)
    return kernel
```

In [12]:
```
def kernelPCA(X, dims, kernel_type):
    kernel = getKernel(X, kernel_type)
    n = kernel.shape[0]
    one = np.ones((n, n), dtype=np.float64) / n
    kernel = kernel - one @ kernel - kernel @ one + one @ kernel @ one
    eigen_val, eigen_vec = np.linalg.eigh(kernel)
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
    idx = np.argsort(eigen_val)[::-1]
    W = eigen_vec[:, idx][:, :dims].real
    return kernel @ W
```

Then we apply KNN to those new coordinates and compute the performance. The code of KNN and face recognition in section is just the same as which in Part 2. Face recognition.

Kernel LDA

Instead of original data, we compute the within-classes scatter(`N`) and between-classes scatter(`M`) with the kernel matrix. According to the wiki and the paper given in PDF, these two scatter matrices is given.

$$M = \sum_{m=1}^{c} l_m (M_m - M_*)(M_m - M_*)^T$$

$$N = \sum_{m=1}^{c} K_m (I - 1_{l_m}) K_m^T$$

$$(M_m)_i = \frac{1}{l_m} \sum_{k=1}^{m_j} k(x_j, x_i^k)$$

$$(M_*)j = \frac{1}{n} \sum_{i=1}^{n} k(x_j, x_i)$$

$$(K_m)_{ij} = k(x_i, x_j^{k=m})$$

And we can get the projected data by solving the eigenvalues of the following formula.

$$N^{-1} M \alpha_i = \lambda_i \alpha_i$$

Implementation. Compute `N` and `M` by iterating through each subject, then solve the eigenvalue. Extract the first `dims` eigenvectors as the projection of original data. I take the first 25 eigenvectors.

```
In [1]: def kernelLDA(X, label, dims, kernel_type):
            label = np.asarray(label)
            c = np.unique(label)
            kernel = getKernel(X, kernel_type)
            n = kernel.shape[0]
            mu = np.mean(kernel, axis=0)
            N = np.zeros((n, n), dtype=np.float64)
            M = np.zeros((n, n), dtype=np.float64)
            for i in c:
                K_i = kernel[np.where(label == i)[0], :]
                l = K_i.shape[0]
                mu_i = np.mean(K_i, axis=0)
                N += K_i.T @ (np.eye(l) - (np.ones((l, l), dtype=np.float64) / l)) @ K_i
                M += l * ((mu_i - mu).T @ (mu_i - mu))
            eigen_val, eigen_vec = np.linalg.eig(np.linalg.pinv(N) @ M)
            for i in range(eigen_vec.shape[1]):
                eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
            idx = np.argsort(eigen_val)[::-1]
            W = eigen_vec[:, idx][:, :dims].real
            return kernel @ W
```

Then apply the KNN to compute the performance.

```
new_coor = kernelLDA(data, label, 25, kernel_type)
new_X = new_coor[:X.shape[0]]
new_test = new_coor[X.shape[0]:]
faceRecognition(new_X, X_label, new_test, test_label, 'LDA', kernel_type)
```

- t-SNE
  - Part1. Try to modify the code a little bit and make it back to symmetric SNE

    The major difference between t-SNE and symmetric SNE is that, instead of Gaussian distribution, Student-T distribution is used in 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, the code for computing joint probability is shown as follows

```
if method == 'tsne':
    num = 1 / (1 + scipy.spatial.distance.cdist(Y, Y, 'sqeuclidean'))
else:
    num = np.exp(-1 * scipy.spatial.distance.cdist(Y, Y, 'sqeuclidean'))
num[range(n), range(n)] = 0
Q = num / np.sum(num)
Q = np.maximum(Q, 1e-12)
```

    and the corresponding formula is

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

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

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

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

    The `num` in the code 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.

we want to modify the code and make it back to symmetric SNE. The formula for computing joint probability and gradient in symmetric SNE

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

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

Part2. Visualize the embedding of both t-SNE and symmetric SNE. Details of the visualization Embedding result for each iteration during the t-SNE and symmetric SNE with the following code. `Y` are the 2-D coordinates after embedding. `label` are the corresponding digits for which each point in `Y` belongs to.

```
In [2]: def plotResult(Y, labels, idx, interval, method, perplexity):
            plt.clf()
            scatter = plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
            plt.legend(*scatter.legend_elements(), loc='lower left', title='Digit')
            plt.title(f'{method}, perplexity: {perplexity}, iteration: {idx}')
            plt.tight_layout()
            if interval:
                plt.savefig(f'./{method}_{perplexity}/{idx // interval}.png')
            else:
                plt.savefig(f'./{method}_{perplexity}/{idx}.png')
```

- Part3. Visualize the distribution of pairwise similarities in both highdimensional space and low-dimensional space, based on both t-SNE and symmetric SNE
  For the visualization of pairwise similarity, we can simply plot the probability of high dimension(`P`) and low dimension(`Q`).

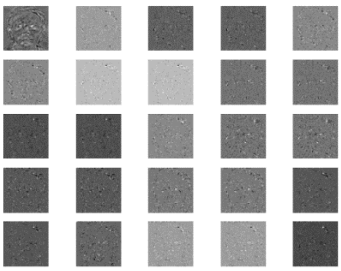
```
In [7]: def plotHighDLowD(P, Q, method, perplexity):
            pal = sns.light_palette('blue', as_cmap=True)
            plt.clf()
            plt.title('High-D Similarity')
            plt.imshow(P, cmap=pal)
            plt.savefig(f'./{method}_{perplexity}/High-D.png')

            plt.clf()
            plt.title('Low-D Similarity')
            plt.imshow(Q, cmap=pal)
            plt.savefig(f'./{method}_{perplexity}/Low-D.png')
```

- Part4. Try to play with different perplexity values. Observe the change in visualization and explain it in the report

b. Experiments setting and results & Discussion

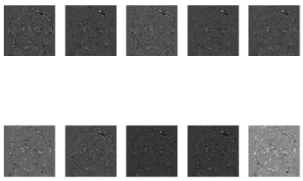
- Kernel Eigenfaces
  - Part1. Use PCA and LDA to show the first 25 eigenfaces and fisherfaces, and randomly pick 10 images to show their reconstruction

25 eigenfaces 25 fisherfaces



Reconstruction

PCA LDA



- Part2. Use PCA and LDA to do face recognition, and compute the performance. You should use k nearest neighbor to classify which subject the testing image belongs to.

```
Face recognition with PCA and KNN:
K= 1, accuracy: 0.833 (25/30)
K= 3, accuracy: 0.833 (25/30)
K= 5, accuracy: 0.900 (27/30)
K= 7, accuracy: 0.900 (27/30)
K= 9, accuracy: 0.833 (25/30)
K=11, accuracy: 0.800 (24/30)
```

```
Face recognition with LDA and KNN:
K= 1, accuracy: 0.767 (23/30)
K= 3, accuracy: 0.833 (25/30)
K= 5, accuracy: 0.867 (26/30)
K= 7, accuracy: 0.867 (26/30)
K= 9, accuracy: 0.800 (24/30)
K=11, accuracy: 0.733 (22/30)
```

- Part3. Use kernel PCA and kernel LDA

```
Face recognition with Kernel PCA(linear kernel) and KNN:
K= 1, accuracy: 0.800 (24/30)
K= 3, accuracy: 0.833 (25/30)
K= 5, accuracy: 0.833 (25/30)
K= 7, accuracy: 0.800 (24/30)
K= 9, accuracy: 0.833 (25/30)
K=11, accuracy: 0.867 (26/30)

KernelLDA not implemented
Face recognition with Kernel LDA(linear kernel) and KNN:
K= 1, accuracy: 0.833 (25/30)
K= 3, accuracy: 0.833 (25/30)
K= 5, accuracy: 0.800 (24/30)
K= 7, accuracy: 0.800 (24/30)
K= 9, accuracy: 0.767 (23/30)
K=11, accuracy: 0.733 (22/30)
```

```
Face recognition with Kernel PCA(polynomial kernel) and KNN:
K= 1, accuracy: 0.800 (24/30)
K= 3, accuracy: 0.833 (25/30)
K= 5, accuracy: 0.867 (26/30)
K= 7, accuracy: 0.833 (25/30)
K= 9, accuracy: 0.833 (25/30)
K=11, accuracy: 0.867 (26/30)

KernelLDA not implemented
Face recognition with Kernel LDA(polynomial kernel) and KNN:
K= 1, accuracy: 0.800 (24/30)
K= 3, accuracy: 0.767 (23/30)
K= 5, accuracy: 0.800 (24/30)
K= 7, accuracy: 0.767 (23/30)
K= 9, accuracy: 0.800 (24/30)
K=11, accuracy: 0.733 (22/30)
```

```
Face recognition with Kernel PCA(rbf kernel) and KNN:
K= 1, accuracy: 0.833 (25/30)
K= 3, accuracy: 0.833 (25/30)
K= 5, accuracy: 0.800 (24/30)
K= 7, accuracy: 0.767 (23/30)
K= 9, accuracy: 0.833 (25/30)
K=11, accuracy: 0.800 (24/30)

KernelLDA not implemented
Face recognition with Kernel LDA(rbf kernel) and KNN:
K= 1, accuracy: 0.767 (23/30)
K= 3, accuracy: 0.800 (24/30)
K= 5, accuracy: 0.767 (23/30)
K= 7, accuracy: 0.733 (22/30)
K= 9, accuracy: 0.767 (23/30)
K=11, accuracy: 0.633 (19/30)
```

We can see that linear and polynomial kernel had a better performance than rbf kernel had in both kernel PCA and kernel LDA. And the performance of normal PCA is better than all three

kinds of kernel methods. For the kernel LDA, the polynomial kernel and rbf kernel perform better than the normal LDA. Besides, the average accuracy of kernel PCA is higher than one of kernel LDA, just like the result in **Part 2. Face recognition**. I don't know which kernel is more suitable for face recognition. But I think if I do the grid search to find the proper set of parameters, all these kernel can obtain nice performance.

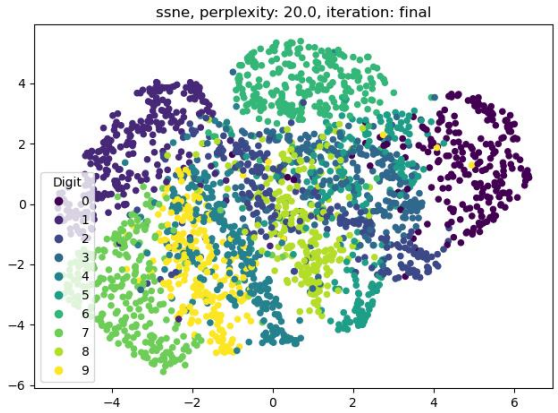
- t-SNE
  - Part1. Try to modify the code a little bit and make it back to symmetric SNE

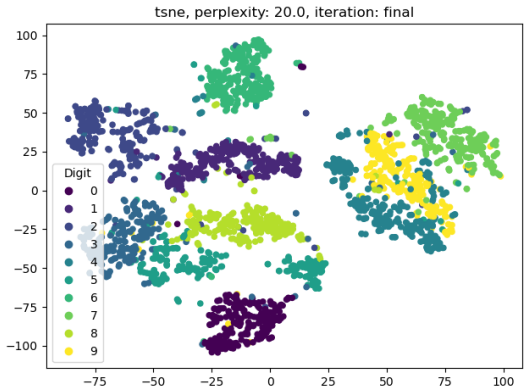
    We can achieve this by changing the `num` and `dY` in t-SNE with the following code,

```
for i in range(n):
    if method == 'tsne':
        dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (dims, 1)).T * (Y[i, :] - Y), axis=0)
    else:
        dY[i, :] = np.sum(np.tile(PQ[:, i], (dims, 1)).T * (Y[i, :] - Y), axis=0)
```

  - Part2. Visualize the embedding of both t-SNE and symmetric SNE. Details of the visualization

    We can see that there is a serious crowding problem in symmetric SNE. After symmetric SNE, the points scatter between -6 and 6. And the boundary between each group is not clear. Groups of points are inlaid with each other. While in t-SNE, the crowding problem is solved. Notice that the points scatter between -100 and 100. It is much looser than symmetric SNE, and the gaps between each groups are clear.





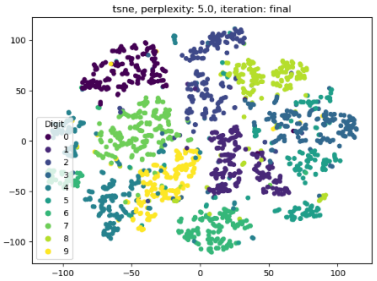
  - Part3. Visualize the distribution of pairwise similarities in both highdimensional space and low-dimensional space, based on both t-SNE and symmetric SNE

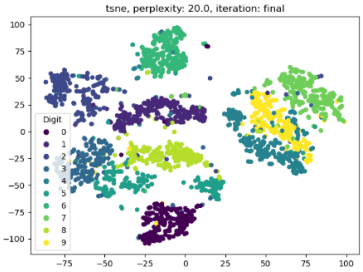
    For the visualization of pairwise similarity, we can simply plot the probability of high dimension(`P`) and low dimension(`Q`).

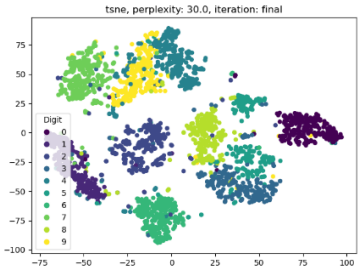
  - Part4. Try to play with different perplexity values. Observe the change in visualization and explain it in the report

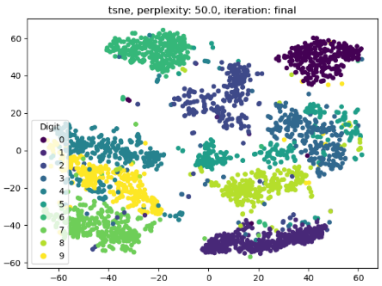
    t-SNE

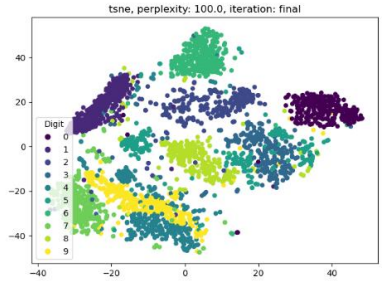
    By tuning the perplexity, we can balance the attention between local and global aspects of our data. For me, I think that perplexity means the number of close neighbors each point has. We can see that with perplexity=5, the points in each group locate loosely. Each group is divided into few small groups. 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.





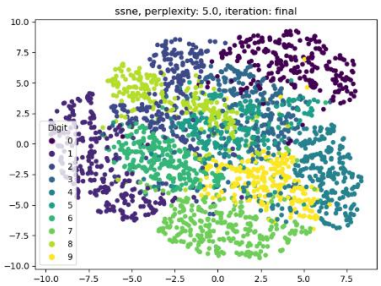


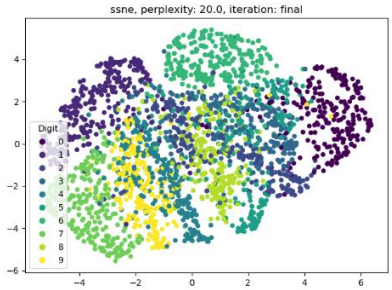


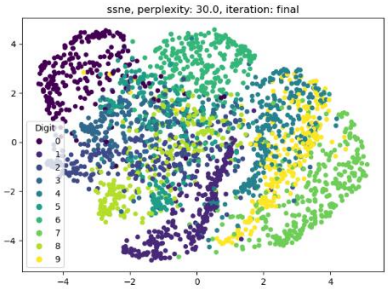


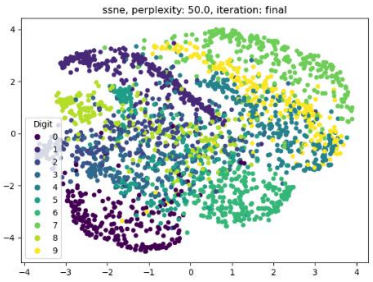
S-SNE

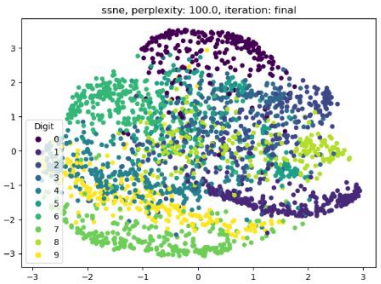
Since symmetric has crowding problem, the differences between each perplexity are not apparent comparing to t-SNE.











c. Observations and discussion

1. Eigenfaces and fisherfaces do not looks like components that makes up faces. In other words, they are not glasses, hair style, mustache, or something else. However, it is reasonable, since both PCA and LDA do not put constraint on requiring all values positive. Thus, it can has negative values. This allows images to add and subtract things. As a result, all eigenfaces and fisher faces looks like a face itself.
2. PCA has a better performance than LDA by looking at reconstruction result.
3. Kernel's result is better than non-kernel's just at expected. This is same as previous homework's result, kernel is a powerful trick. The reconstruction looks having more details in kernel methods.
4. From the result, it can be seen easily that Symmetric SNE bounds into crowded problem. We can not really distinguish different classes without color label. As to t-SNE, we modify low-dimension

distribution to t-distribution. It helps pairwise points that has far distance in high-dimensional space to have even further distance in low-dimensional space. This results in clear grouping result.

5. Perplexity is a major hyper parameter in both symmetric SNE and t-SNE. It considers how many neighbors data have. Generally, larger data need to set larger value on perplexity. It would allows data to have more neighbor, and not to be so sensitive to small group.