Data Loading

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
data = np.genfromtxt('mnist_X.csv', delimiter=',')
# data
label = np.genfromtxt('mnist_label.csv', delimiter=',')
```

From numpy, we can use genfromtxt to help us loading data as numpy array. The shape of data will be (5000, 784) in two dimension and label will be (5000, 1) in two dimension too.

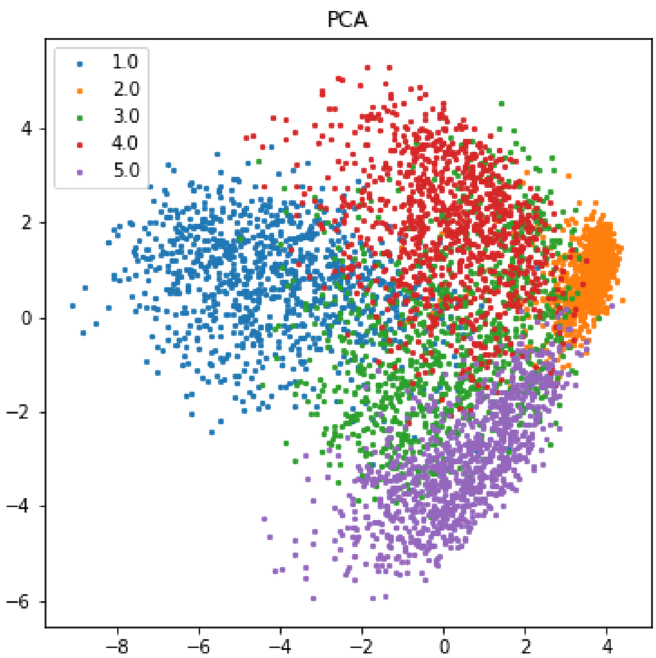
PCA

```
def myPCA(X, n_component=2):
    X_mean = np.mean(X, axis=0)
#     X_std = np.std(X, axis=0)
    X_scaled = (X - X_mean)
    cov = np.cov(X_scaled.T)
    eig_vals, eig_vecs = np.linalg.eig(cov)

#     print("Eigenvals:\n{}\n".format(eig_vals))
#     print("Eigenvecs:\n{}\n".format(eig_vecs))

    eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:, i]) for i in range(len(eig_vals))]
    eig_pairs.sort(key=lambda x: x[0], reverse=True)
    d = np.array([eig[1].real for eig in eig_pairs[:n_component]])
    data = np.dot(X_scaled, d.T)
    return data
```

My PCA function is follow from lecture slide, first of all, we need to standardize the data. We start to calculate the mean vector ***X_mean*** from the whole data ***X*** and subtract it. Then we need to find out the covariance of matrix. In np.cov(), it follows by $\Sigma = \frac{1}{n-1}\left((\mathbf{X}-\bar{\mathbf{x}})^T(\mathbf{X}-\bar{\mathbf{x}})\right)$ The eigenvectors and eigenvalues of a covariance matrix represent the "core" of a PCA. In order to decide which eigenvectors can dropped without losing too much information for the construction of lower-dimensional subspace, we need to inspect the corresponding eigenvalues from highest to lowest in order choose the top ***d*** eigenvectors.



LDA

From the following code, we need to calculate each class's mean vectors.

```
mean_vectors = []
for l in np.unique(label):
    mean_vectors.append(np.mean(data[np.where(label == l)], axis=0))
```

Second, we will compute two 784 * 784 dimensional matrices, which is within-class (***S_W***) and the between-class (***S_B***) scatter matrix.

$$\mathrm{S_W} = \sum_{i=1}^{c} S_i$$

, where $\mathrm{S_i} = \sum_{x \in D_i}^{n} (x - m_i)(x - m_i)^T$

```
S_W = np.zeros((784, 784))
for l, mean_vector in zip(range(1,6), mean_vectors):
    S = (data[label == l] - mean_vector).T.dot((data[label == l] - mean_vector))
    S_W += S.T
```

$$\mathrm{S_B} = \sum_{i=1}^{c} N_i (m_i - m)(m_i - m)^T$$

```
overall_mean = np.mean(data, axis=0)
S_B = np.zeros((784, 784))
for i, mean_vector in enumerate(mean_vectors):
    n = data[label == i+1].shape[0]
    mean_vector = mean_vector.reshape(784, 1)
    overall_mean = overall_mean.reshape(784, 1)
    S_B += n * (mean_vector - overall_mean).dot((mean_vector - overall_mean).T)
```

Next, we will solve the generalized eigenvalue problem for the matrix $S_W^{-1} S_B$ to obtain the linear discriminants. In practical issue we find the homework can not be invertible, so we use np.pinv function which is pseudo inverse

```
eig_vals, eig_vecs = np.linalg.eig(np.linalg.pinv(S_W).dot(S_B))
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]

eig_pairs = sorted(eig_pairs, key=lambda k: k[0], reverse=True)

print('Eigenvalues in decreasing order:\n')
for i in eig_pairs[:2]:
    print(i[0])
```

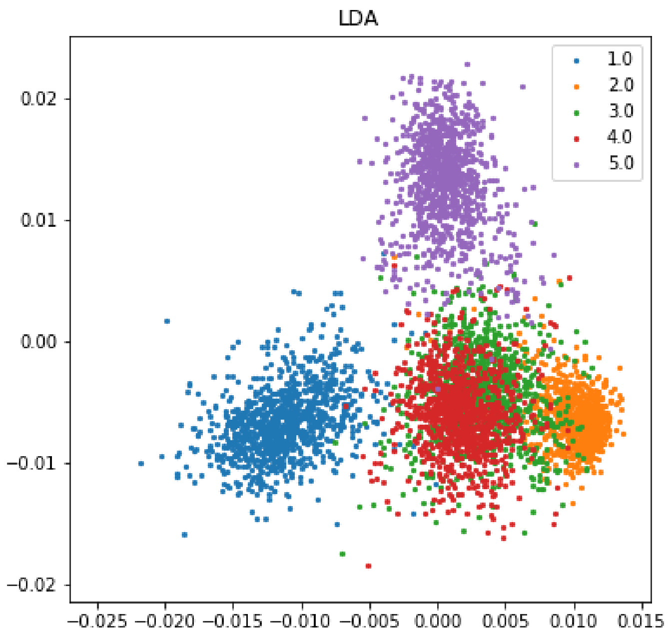
```
Eigenvalues in decreasing order:

8.110844814439524
5.327113124015034 5
```

Finally, we sorting the eigenvectors by decreasing eigenvalues to select linear discriminants for the new feature subspace (2-dim) and choosing *k* eigenvectors with the largest eigenvalues.

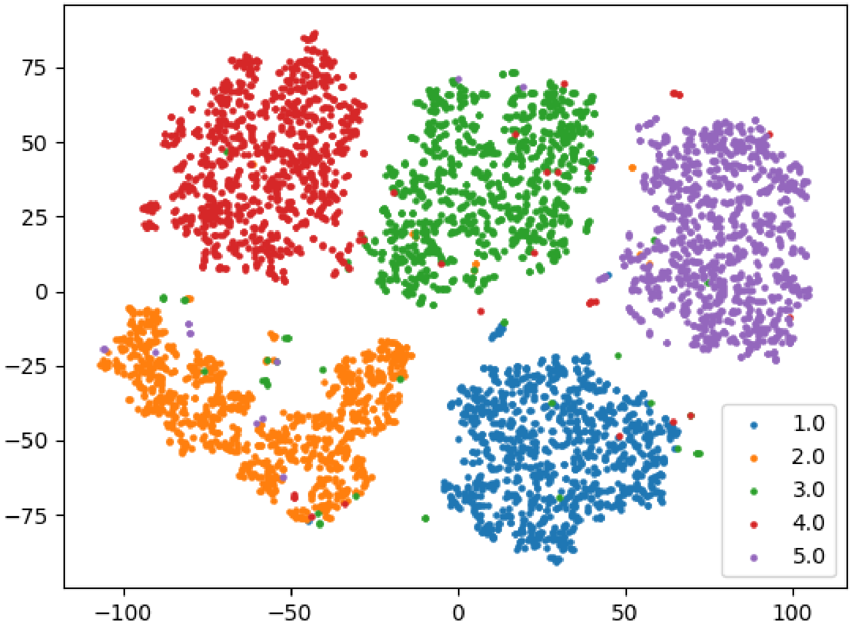
```
W = np.hstack((eig_pairs[0][1].reshape(784,1), eig_pairs[1][1].reshape(784,1)))
X_lda = data.dot(W.real)
X_lda
```

```
array([[-0.01290896, -0.00656254],
       [-0.01024413, -0.00873476],
       [-0.00697849, -0.00613905],
       ...,
       [-0.00347863,  0.01377908],
       [-0.00440569,  0.00617939],
       [ 0.00126217,  0.01254343]])
```

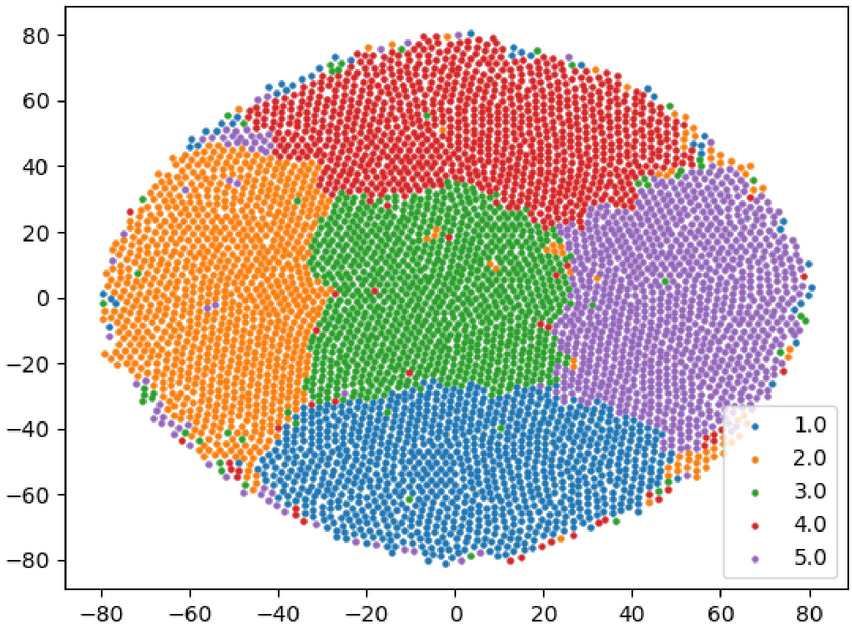


t-SNE & Symmetric SNE

t-SNE



Symmetric SNE



From the picture above we can see the symmetric SNE is very crowd but t-SNE does not. Because of the t-distribution use distribution with longer-tail, such that data should be further away in low-D in order to achieve low probability, we can find out that t-SNE has very good result.

The differernt between t-SNE and Symmetric SNE is the following formula:

$$q_{ij} = \frac{\exp(-\|y_i - y_j\|^2)}{\sum_{k\neq l}\exp(-\|y_l - y_k\|^2)} \rightarrow \text{Symmetric SNE}$$

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

According to the formula above, we trace the code and turn the t-SNE method into symmetric SNE method:

```
# Compute pairwise affinities
sum_Y = np.sum(np.square(Y), 1)
num = -2. * np.dot(Y, Y.T)
# t-SNE method
# num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))

# symmetric SNE method
num = np.exp(-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)
```

```
def pca(X=np.array([]), no_dims=50):
    """
        Runs PCA on the NxD array X in order to reduce its dimensionality to
        no_dims dimensions.
    """

    print("Preprocessing the data using PCA...")
# Implement PCA here
    X_mean = np.mean(X, axis=0)
#     X_std = np.std(X, axis=0)
    X_scaled = (X - X_mean)
    cov = np.cov(X_scaled.T)
    eig_vals, eig_vecs = np.linalg.eig(cov)

#     print("Eigenvals:\n{}\n".format(eig_vals))
#     print("Eigenvecs:\n{}\n".format(eig_vecs))

    eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:, i]) for i in range(len(eig_vals))]
    eig_pairs.sort(key=lambda x: x[0], reverse=True)
    d = np.array([eig[1].real for eig in eig_pairs[:no_dims]])
    data = np.dot(X_scaled, d.T)
    return data
```

The ***pca*** function in ***tsne_symmetric_SNE.py*** is same as ***myPCA*** function.

Eigenface

```
import PIL
imgs = []
for i in range(40):
    for j in range(10):
        pic = PIL.Image.open('./att_faces/s{}/{}.pgm'.format(i+1, j+1))
        imgs.append(np.array(pic).reshape(-1))
imgs = np.array(imgs)
print(imgs.shape)
```

For the preprocessing, we use PIL to load 400 PGM files and transform the data into numpy array.

```
imgs
```

```
array([[ 48,  49,  45, ...,  47,  46,  46],
       [ 60,  60,  62, ...,  32,  34,  34],
       [ 39,  44,  53, ...,  29,  26,  29],
       ...,
       [125, 119, 124, ...,  36,  39,  40],
       [119, 120, 120, ...,  89,  94,  85],
       [125, 124, 124, ...,  36,  35,  34]], dtype=uint8)
```

We use the ***myPCA*** function to decomposition the eigenvectors, and select the important 25 eigenfaces. The values of the 25 eigenfaces follows the picture below.

```
eigenface = myPCA(imgs.T, n_component=25)
eigenface = eigenface / np.linalg.norm(eigenface, axis=0)
print(eigenface.shape)
eigenface
```

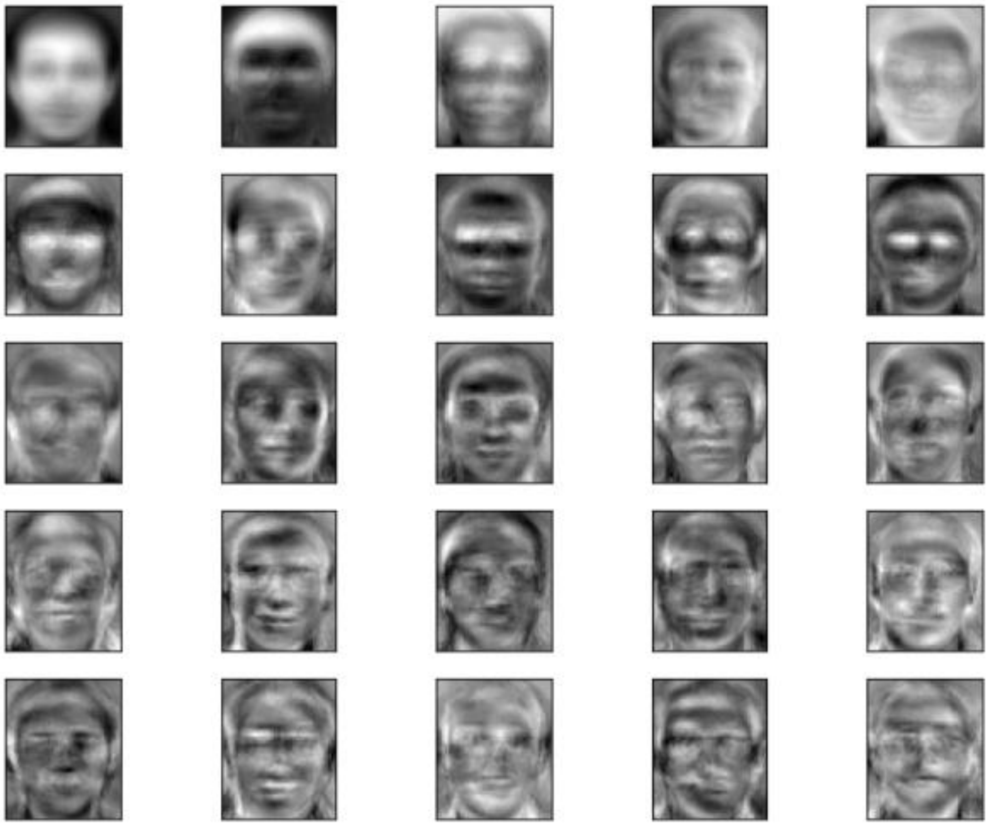
```
(10304, 25)

array([[-0.00837344, -0.01248618,  0.01881755, ...,  0.01048953,
        -0.00271995,  0.00518574],
       [-0.00839009, -0.01243664,  0.01883255, ...,  0.00974882,
        -0.00197758,  0.0057063 ],
       [-0.00827368, -0.01246577,  0.01877995, ...,  0.01037806,
        -0.0025246 ,  0.0063196 ],
       ...,
       [-0.01285105, -0.00166376, -0.00997099, ...,  0.00242194,
        -0.01134842, -0.00122039],
       [-0.01305508, -0.00161541, -0.00836337, ...,  0.00460115,
        -0.00925658, -0.0052217 ],
       [-0.01328849, -0.00292062, -0.00794709, ..., -0.00146082,
        -0.00948866, -0.00590528]])
```

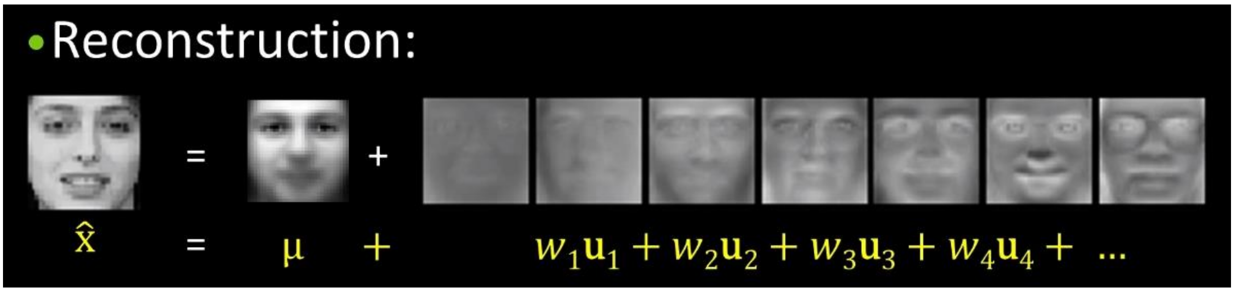
Next, we use np.reshape() to show the images from those 25 eigenfaces.

```python
fig, ax = plt.subplots(5, 5, figsize=(10, 8))

for i in range(25):
    ax[int(i/5), i%5].imshow(eigenface.T[i].reshape(112, 92), cmap='gray')
    ax[int(i/5), i%5].get_xaxis().set_visible(False)
    ax[int(i/5), i%5].get_yaxis().set_visible(False)
plt.show()
```



In order to reconstruct new face, we will need the mean face of all the face data, and we follow this formula:

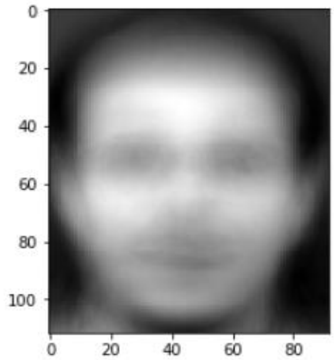


$$\text{new}_{\text{face}} = \text{mean}_{\text{face}} + \sum_{i=1}^{25} w_i u_i$$

Where w is the weight and u is the eigenface we pick previous.

```
mean_face = np.mean(imgs, axis=0)
plt.imshow(mean_face.reshape(112, 92), cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x20033dc5be0>
```



From the picture below, we use ***reconstruction*** function to reconstruct face from 25 eigenfaces. First, we will create a weights matrix which is used to determine how important the eigenfaces we will use. Next, we dot weights with specific index we pick for the picture and 25 eigenfaces, we can get a reconstruction face after we add the mean of all the face.

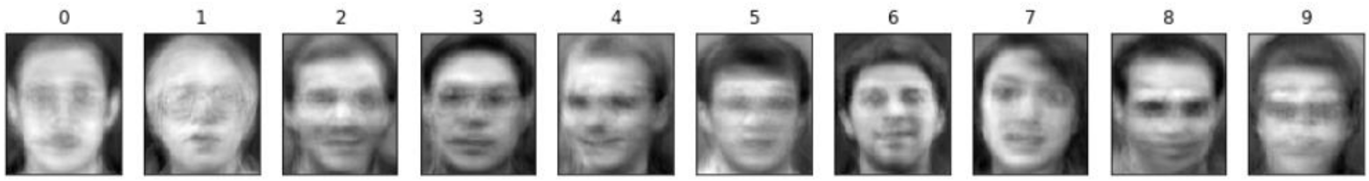
```
def reconstruction(mean, eigenface, mu, image_index):
    weights = np.dot(mean, eigenface.T)
    centered_vector = np.dot(weights[image_index], eigenface)
    recovered_image = (mu + centered_vector).reshape(112, 92)
    return recovered_image
```

Reconstruct 10 different person

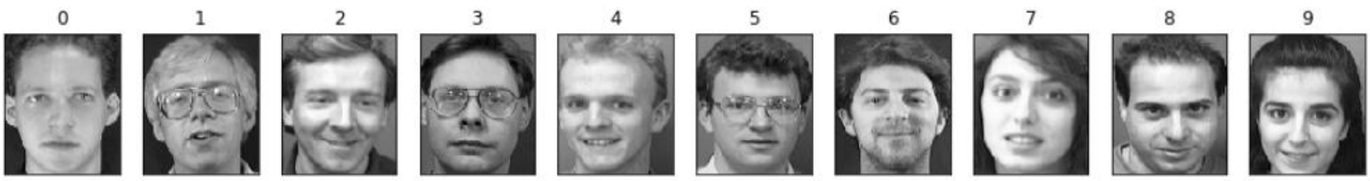
```
imgs_mean = imgs - np.mean(imgs, axis=0)
mean_face = np.mean(imgs, axis=0)

recovered_images=[reconstruction(imgs_mean, eigenface.T, mean_face, i*10) for i in range(10)]

fig, ax = plt.subplots(1, 10, figsize=(16, 4))
for n in range(len(recovered_images)):
    new_face = recovered_images[n]
    ax[n].imshow(new_face, cmap='gray')
    ax[n].set_title("{}".format(n))
    ax[n].get_xaxis().set_visible(False)
    ax[n].get_yaxis().set_visible(False)
plt.show()
```



The picture below is the original face we picked



The reconstruction pictures is blurred, it’s because we just use fewer eigenfaces to describe the face, if we use more than 25 eigenfaces, the picture we be reconstructed more and more clearly.