# ML_HW7 Report

## 1.1.1 Eigenfaces

- Load data

  Since the original image size (231,195) is too big, I follow the paper “Kernel Eigenfaces vs. Kernel Fisherfaces: Face Recognition Using Kernel Methods” resized them into (41,29),  it’s easier to implement.

  Then I flatten the array make it into 2D array.

```
def load_data(foldername):

    filenames = glob.glob('./Yale_Face_Database/'+ foldername + '/*.pgm')

    for file in filenames:
        im = PIL.Image.open(file)
        im = im.resize((29,41),PIL.Image.ANTIALIAS)
        im.save(foldername+'/'+file.split('/')[-1])

    data = []

    for file in filenames:
        data.append(imageio.imread(file))


    filenames = glob.glob('./' + foldername+'/*.pgm')

    data_resize = []

    for file in filenames:
        data_resize.append(imageio.imread(file))

    return filenames,  np.array(data), np.array(data_resize)
```

```
train_file, train, train_resize = load_data("Training")
test_file, test, test_resize = load_data("Testing")
```

```
train_label = [int(filename.split('/')[-1][7:9]) for filename in train_file]
test_label = [int(filename.split('/')[-1][7:9]) for filename in test_file]
```

```
train = train_resize.reshape((train.shape[0], -1))
test = test_resize.reshape((test.shape[0], -1))
train.shape
```

```
(135, 1189)
```

- Calculate the covariance matrix  get the get the eigenfaces and draw out the first 25 eigenfaces.

  Set the number of principle component to 50.

```
def pca(data, cov, n=50):

    data = data - data.mean(axis = 0)
    cov = np.cov(data.T)
    eigenvalue, eigenvector = np.linalg.eig(cov)
    eigenface = (eigenvector[:, np.argsort(eigenvalue)[::-1]])[:,:n]

    return eigenface
```

```
def draw(row, col, figsize, data, title):
    fig, axes = plt.subplots(row, col, sharex=True, sharey=True, figsize=figsize)
    for i, (fp, ax) in enumerate(zip(data, axes.flatten())):
        ax.imshow(fp.real, cmap='gray')
        ax.set_title(title+' - {:d}'.format(i))
    plt.show()
```

```
mean = train.mean(axis = 0)
cov = np.cov((train-mean).T)
```

```
n=50
eigenface = pca(train, cov,n)
draw(5, 5, (16,20), eigenface.T.reshape(n,41,29)[:25,:,:], 'eigenfaces')
```

Result:

- Randomly choose 10 images to reconstruct and draw out.

```
def reconstruct(data, eigenface, chosen_indices):

    r = []
    for i in chosen_indices:
        recontruct = (data[i]@eigenface@eigenface.T).T.reshape(1,41,29)
        r.append(recontruct)

    return r
```

```
chosen_indices = np.random.randint(train.shape[0], size=10)
reconstruct_list = reconstruct(train, eigenface, chosen_indices)
draw(2, 5, (16, 8), train_resize[chosen_indices],'original')
draw(2, 5, (16, 8), np.array(reconstruct_list).reshape((10,41,29)), 'reconstruct')
```

Result:

- Face Recognition and show the accuracy.

  project both training data and testing data on the low dimension, use a nearest neighbor to classify which subject the testing image belongs.

  The accuracy is 0.667.

```
def project(W, data):
    w = (W.T.real@data)
    return w
```

```
def face_recognition(test_data, train_w, w):

    accu = 0
    for face_i in range(test_data.shape[0]):
        test = test_data[face_i]
        test = test - test.mean()
        test_w = project(w, test.T)
        distance = []
        for i in range(train_w.shape[1]):
            distance.append((test_w - train_w[:,i])@ (test_w - train_w[:,i]).T)

        #print(np.argsort(distance)//9)
        cluster = (np.argsort(distance)//9)[0]
        if test_label[face_i] == cluster+1 :
            accu += 1

    return accu/test_data.shape[0]
```

```
mean = train.mean(0)
train_w_pca = project(eigenface, (train-mean).T)
```

```
pca_accu = face_recognition(test, train_w_pca, eigenface)
print(pca_accu)
```

```
0.6666666666666666
```

### 1.1.2 Kernel Eigenfaces

- Try two kinds of kernel, one is Gaussian, another is polynomial.

  Calculate kernel matrix to get the kernel eigenfaces, also use it to get the data in the low dimension space.  Based on the formula below:

$$\mathbf{w}^{\Phi} \cdot \Phi(\mathbf{x}) = \sum_{i=1}^{m} \alpha_i (\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x})) = \sum_{i=1}^{m} \alpha_i k(\mathbf{x}_i, \mathbf{x})$$

```
def get_gaussian_kernel(X, Y, sigma):

    D = cdist(X,Y, 'euclidean')
    K = np.exp(-sigma * D**2)

    return K
```

```
def get_poly_kernel(X, Y, d):

    return (X@Y.T)**d
```

```
method = "gaussian"
if method == "gaussian":
    K_train = get_gaussian_kernel((train-mean),(train-mean),0.000001)
    K_new = get_gaussian_kernel(train-mean,test-test.mean(0),0.000001)
elif method == "poly":
    K_train = get_poly_kernel((train-mean),(train-mean),2)
    K_new = get_poly_kernel(train-mean,test-test.mean(0),2)

N = (train-mean).shape[0]
one_N = np.ones((N, N))/N
K_train = K_train - one_N.dot(K_train) - K_train.dot(one_N) + one_N.dot(K_train).dot(one_N)
```

```
kernel_eigenvalue, kernel_eigenvector = np.linalg.eig(K_train)
kernel_eigenface = (kernel_eigenvector[:, np.argsort(kernel_eigenvalue)[::-1]])[:,:n]
```

```
train_w_kpca = project(kernel_eigenface, K_train)
test_w_kpca = project(kernel_eigenface, K_new)
```

- Face Recognition and show the accuracy. Use the method as PCA.

  The accuracy of Gaussian is 0.733.

  The accuracy of polynomial(degree = 2) is 0.53

```
def kernel_face_recognition(test_w, train_w):

    accu = 0
    for face_i in range(30):
        distance = []
        for i in range(train_w.shape[1]):
            distance.append((test_w[:,face_i] - train_w[:,i])@ (test_w[:,face_i] - train_w[:,i]).T)
        #print(np.argsort(distance)//9)
        cluster = (np.argsort(distance)//9)[0]
        #group, counts = np.unique(first_k, return_counts = True)
        if test_label[face_i] == cluster+1 :
            accu += 1

    return accu/test_resize.shape[0]
```

```
kpca_accu = kernel_face_recognition(test_w_kpca, train_w_kpca)
kpca_accu
```

```
0.7333333333333333
```

### 1.2.1 Fisherfaces

- Get fisherfaces and draw out

  Use LDA to get the first k eigenvector of $(S_w)^{-1}S_B$.

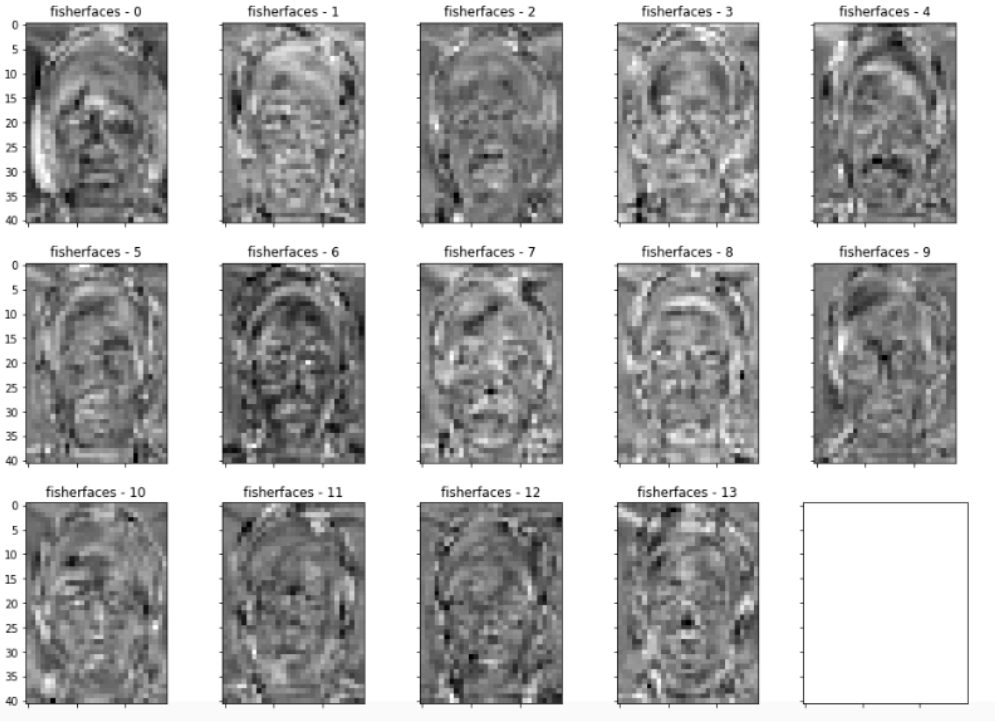
  The biggest k is equal to #class -1, so is 14.

```
def lda(X, y, rank=14):
    def calculate_sw(X, y, unique_y):
        def sj(X):
            mean = X.mean(axis=1)[..., np.newaxis] # (n_dim, 1)
            return (X - mean) @ (X - mean).T

        return np.array([sj(X[:, y == uy]) for uy in unique_y]).sum(axis=0)

    def calculate_sb(X, y, unique_y):
        means = np.array([X[:, y==uy].mean(axis=1) for uy in unique_y]) # (n_group, n_dim)
        return np.array([(-(means - means[i])).T @ (-(means - means[i])) for i in range(means.shape[0])]).sum(axis=0)


    X = X.reshape((X.shape[0], -1)).T # flatten and transpose # (n_dim, n_sample)
    unique_y = np.unique(y)
    sw = calculate_sw(X, y, unique_y) + np.identity(X.shape[0]) # deal with singular matrix issue
    sb = calculate_sb(X, y, unique_y)
    eigenvalues, eigenvectors = np.linalg.eig(np.linalg.inv(sw) @ sb)

    # get first `rank` eigenvectors
    eigenvectors = (eigenvectors[:, np.argsort(eigenvalues)[::-1]])[:, :rank]
    return eigenvectors
```

```
n=14
fisherface = lda(train_resize, train_label, n)
```

```
draw(5,5,(16,20),fisherface.T.reshape(n,41,29)[:25,:,:], 'fisherfaces')
```

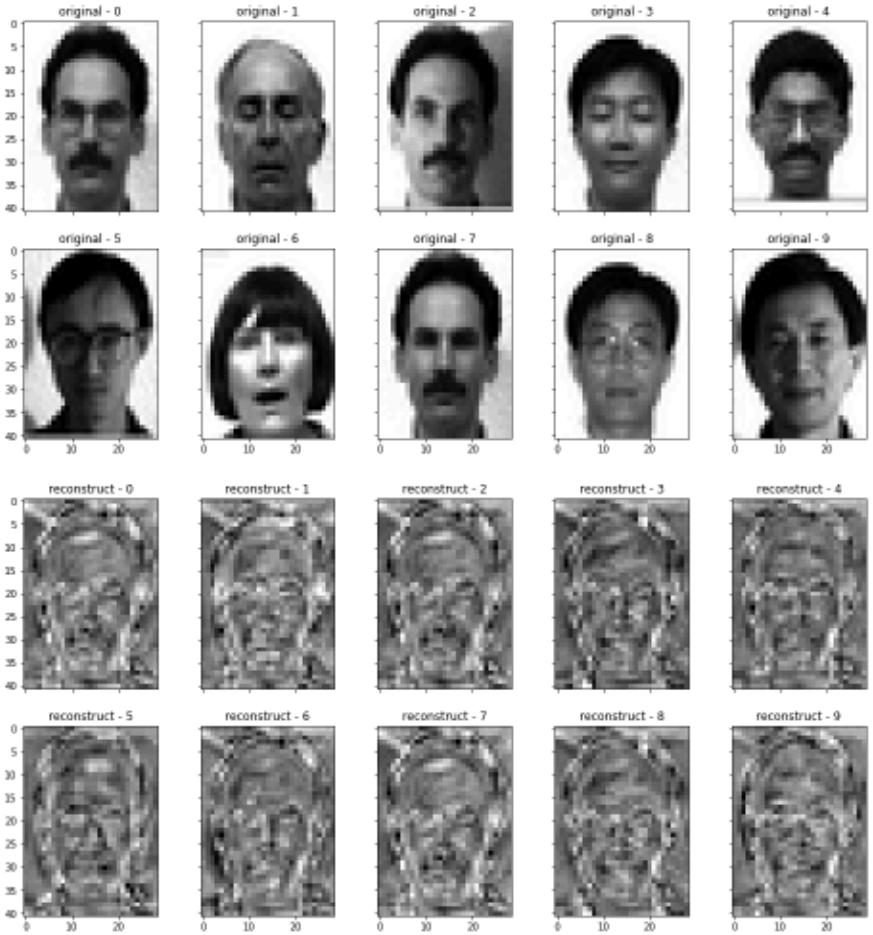
Result:



- Randomly choose 10 images to reconstruct and draw out.

  Although it can't reconstruct well, we can see that the same subject after the reconstruction is the same. It capture the main feature of each subject.

```
chosen_indices = np.random.randint(train.shape[0], size=10)
reconstruct_list = reconstruct(train, fisherface, chosen_indices)
draw(2, 5, (16, 8), train_resize[chosen_indices],'original')
draw(2, 5, (16, 8), np.array(reconstruct_list).reshape((10,41,29)), 'reconstruct')
```




- Face Recognition and show the accuracy

  The accuracy is 0.9

```
train_w_lda = project(fisherface, (train-mean).T)
train_w_lda.shape
lda_accu = face_recognition(test, train_w_lda, fisherface)
print(lda_accu)
```

```
0.9
```

## 1.2.2 Kernel fisherfaces

- Use the method in paper, first get Z and K matrix, and find the kernel fisherfaces.

$$(k_{rs})_{tu} = k(\mathbf{x}_{tr}, \mathbf{x}_{us}) = \Phi(\mathbf{x}_{tr}) \cdot \Phi(\mathbf{x}_{us}) \quad (9)$$

Let $K$ be a $m \times m$ matrix defined by the elements $(K_{tu})_{u=1,\ldots,c}^{t=1,\ldots,c}$, where $K_{tu}$ is a matrix composed of dot products in the feature space $R^f$, i.e.,

$$K = (K_{tu})_{u=1,\ldots,c}^{t=1,\ldots,c} \text{ where } K_{tu} = (k_{rs})_{s=1,\ldots,l_u}^{r=1,\ldots,l_t} \quad (10)$$

Note $K_{tu}$ is a $l_t \times l_u$ matrix, and $K$ is a $m \times m$ symmetric matrix. We also define a matrix $Z$:

$$Z = (Z_t)_{t=1,\ldots,c} \quad (11)$$

where $(Z_t)$ is a $l_t \times l_t$ matrix with terms all equal to $\frac{1}{l_t}$,

$$\begin{aligned} W_{OPT}^{\Phi} &= \arg\max_{W^{\Phi}} \frac{|(W^{\Phi})^T S_B^{\Phi} W^{\Phi}|}{|(W^{\Phi})^T S_W^{\Phi} W^{\Phi}|} \\ &= \arg\max_{W^{\Phi}} \frac{|\boldsymbol{\alpha} K Z K \boldsymbol{\alpha}|}{|\boldsymbol{\alpha} K K \boldsymbol{\alpha}|} = [\mathbf{w}_1^{\Phi} \ldots \mathbf{w}_m^{\Phi}] \end{aligned}$$

The accuracy of gaussian is 0.867
The accuracy of polynomial is 0.8

```
Z = np.zeros((train.shape[0],train.shape[0]))
for i in range(15):
    Z[(i*9):(i+1)*9,(i*9):(i+1)*9] = 1/9

K = np.zeros((train.shape[0],train.shape[0]))
for i in range(15):
    for j in range(15):
        x = train-mean

        if method =="guassian":
            k = get_gaussian_kernel(x[(i*9):(i+1)*9],x[(j*9):(j+1)*9],0.000001)
        elif method == 'poly':
            k = get_poly_kernel(x[(i*9):(i+1)*9],x[(j*9):(j+1)*9],2)

        N = k.shape[0]
        one_N = np.ones((N, N))/N
        k = k - one_N.dot(k) - k.dot(one_N) + one_N.dot(k).dot(one_N)
        K[(i*9):(i+1)*9,(j*9):(j+1)*9] = k
K = K + np.identity(K.shape[0])
```

```
S = np.linalg.inv(K)@Z@K
```

```
kernel_fishervalue, kernel_fishervector = np.linalg.eig(S)
```

```
kernel_fisherface = (kernel_fishervector[:, np.argsort(kernel_fishervalue)[::-1]])[:,:n]
train_w_klda = project(kernel_fisherface, K_train)
test_w_klda = project(kernel_fisherface, K_new)
klda_accu = kernel_face_recognition(test_w_klda, train_w_klda)
print(klda_accu)
```

```
0.8666666666666667
```

- Discussion

Kernel method can map non-linear data into feature space, so it may get better performance in face recognition. In my work, Kernel PCA is better than PCA, however, the accuracy of Kernel LDA is a little bits lower than simple LDA. There is a possible reason I figure out: actually the samples in testing set is small, so the miss recognition of Kernel LDA is just one more than simple LDA. I think it is basically in the same level of the performance. In the experiment of the paper “Kernel Eigenfaces vs. Kernel Fisherfaces: Face Recognition Using Kernel Methods”, the performance of them is quite close too, maybe, if the testing set become bigger, we can see the bigger improvement.

Both LDA is performance better than PCA, I think it’s because we are dealing with face recognition, just need to find out the subject, the LDA can nicely extract the feature of each subject, so it can performance well.

Gaussian kernel is better than polynomial in my experiment. However, I didn’t try every parameter to get the optimal, I think there still have improvement space in both kernel method.

## 2.1 T-sne v.s. symmetric sne

- Modify the given t-sne code a little bit and make it back to symmetric SNE

  Below is the code I have changed.

```
# Compute pairwise affinities
sum_Y = np.sum(np.square(Y), 1)
num = -2. * np.dot(Y, Y.T)
num = np.exp(- 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)

# Compute gradient
PQ = P - Q
for i in range(n):
    dY[i, :] = 2 * np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

sne

```
# Compute pairwise affinities
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)

# 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)
```

t-sne(original)

The change is based on the different of formula.

Symmetric sne: The distribution in high-D and low-D is the same, both are gaussian.

$$p_{ij} = \frac{\exp(-\,||\,x_i - x_j\,||^2\,/(2\sigma^2))}{\sum_{k\neq l}\exp(-\,||\,x_l - x_k\,||^2\,/(2\sigma^2))}$$

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

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

T-sne: to ease crowding problem,the distribution in low-D become student t-distribution.

$$p_{ij} = \frac{\exp(-\,||\,x_i - x_j\,||^2\,/(2\sigma^2))}{\sum_{k\neq l}\exp(-\,||\,x_l - x_k\,||^2\,/(2\sigma^2))}$$

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

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

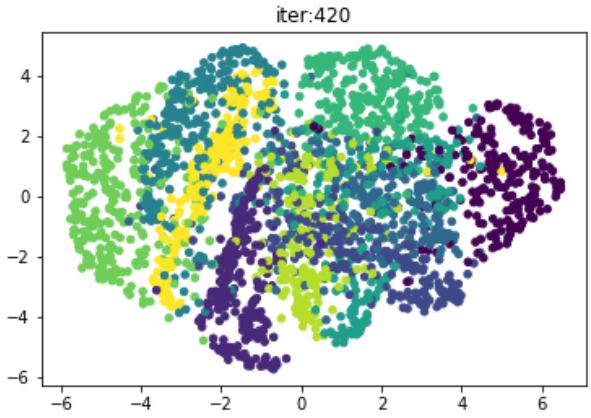
- Embedding

  When abs(error-last_error) < 1e-7, converge. Maximum iterative time = 1000

  Can clearly see T-sne solve the crowding problem.

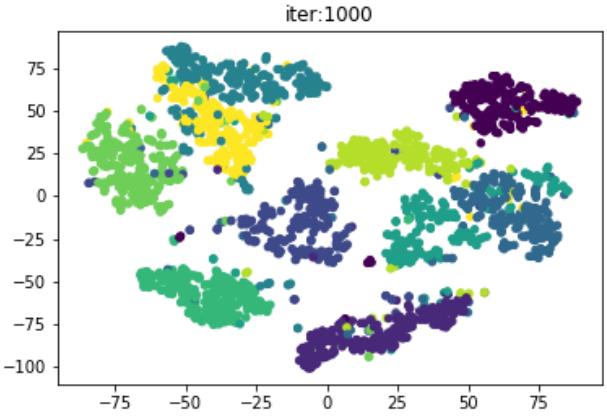
  Symmetric sne:

```
X = np.loadtxt("mnist2500_X.txt")
labels = np.loadtxt("mnist2500_labels.txt")
Y_sne, P_sne, Q_sne = sne(X, 2, 50, 20.0)
pylab.scatter(Y_sne[:, 0], Y_sne[:, 1], 20, labels)
pylab.show()
```




  T-sne:

```
X = np.loadtxt("mnist2500_X.txt")
labels = np.loadtxt("mnist2500_labels.txt")
Y_tsne, P_tsne, Q_tsne = tsne(X, 2, 50, 20.0)
pylab.scatter(Y_tsne[:, 0], Y_tsne[:, 1], 20, labels)
pylab.show()
```

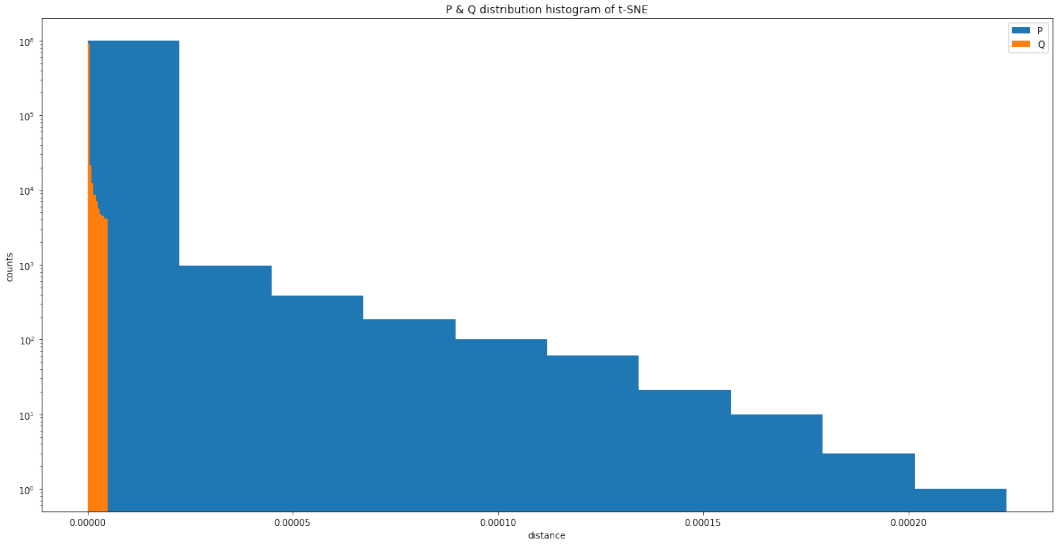


- Distribution of pairwise similarities.

  Also can see T-sne solve the crowding problem.

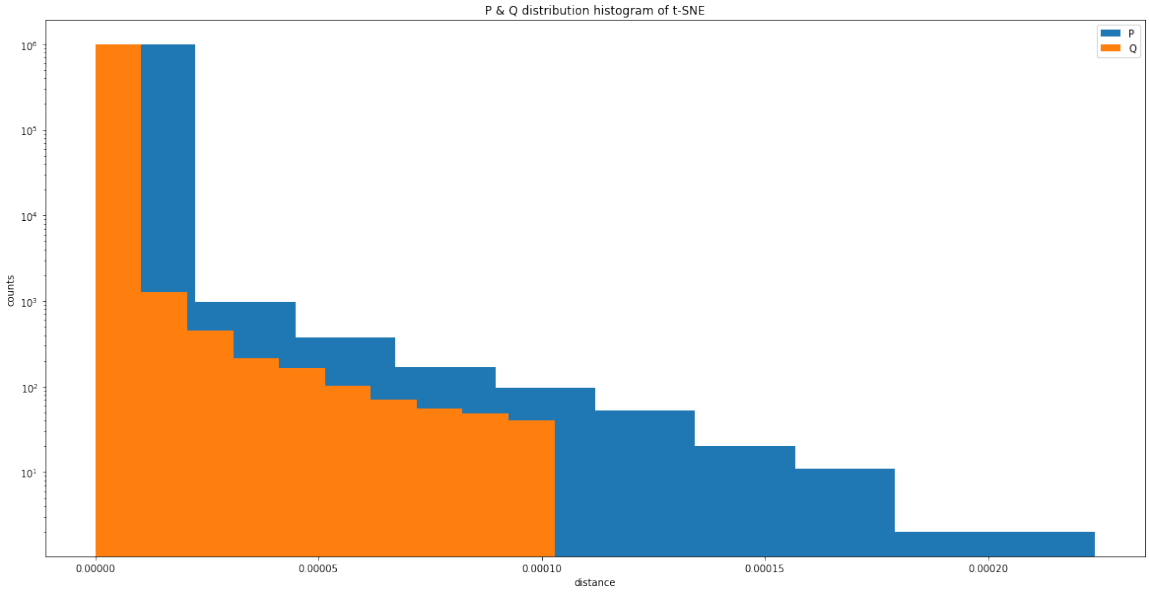
  Symmetric sne:

```
plt.figure(figsize=(20, 10))
plt.hist(np.random.choice(P_sne.flatten(), size=1000000), log=True, label='P')
plt.hist(np.random.choice(Q_sne.flatten(), size=1000000), log=True, label='Q')
plt.legend()
plt.title('P & Q distribution histogram of t-SNE')
plt.xlabel('distance')
plt.ylabel('counts')
```




  T-sne :

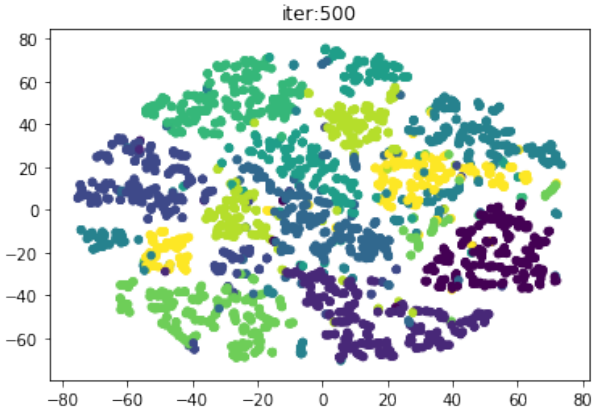
```
plt.figure(figsize=(20, 10))
plt.hist(np.random.choice(P_tsne.flatten(), size=1000000), log=True, label='P')
plt.hist(np.random.choice(Q_tsne.flatten(), size=1000000), log=True, label='Q')
plt.legend()
plt.title('P & Q distribution histogram of t-SNE')
plt.xlabel('distance')
plt.ylabel('counts')
```



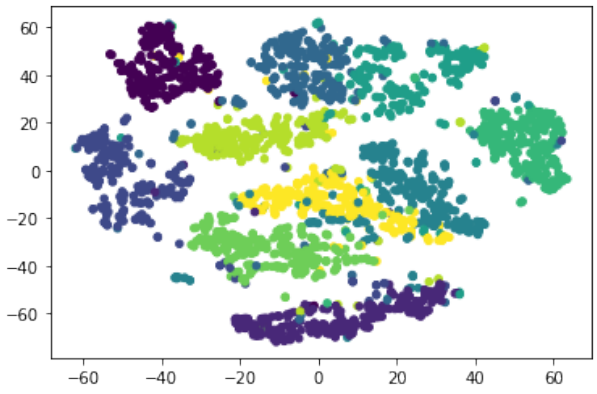
2.2 Different  perplexity

Low perplexity means just few neighbors have influence, may split one group into many groups.
High perplexity means it is depend on the whole structure, but may could not separate nicely.

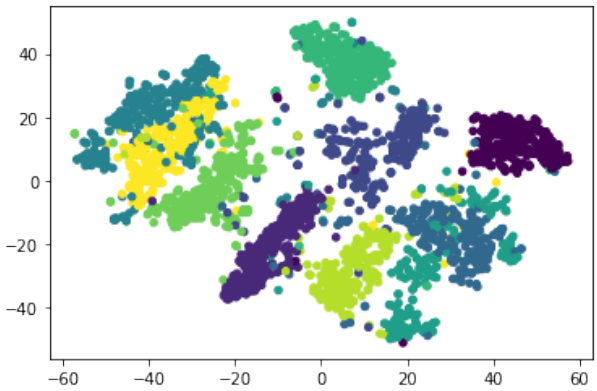
I use t-sne to do the experiment. I set perplexity to 5,20,50,100,200. All iterate 500 times. I think perplexity in the 20 and 50 have the better performance, the within class distance is small, the between class distance is big. Too high or too low perplexity aren't the nice choice.



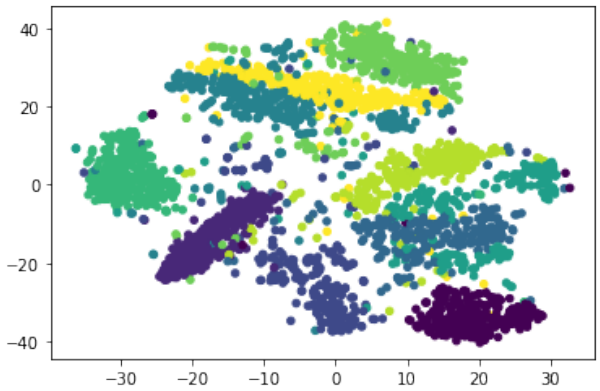

perplexity = 5



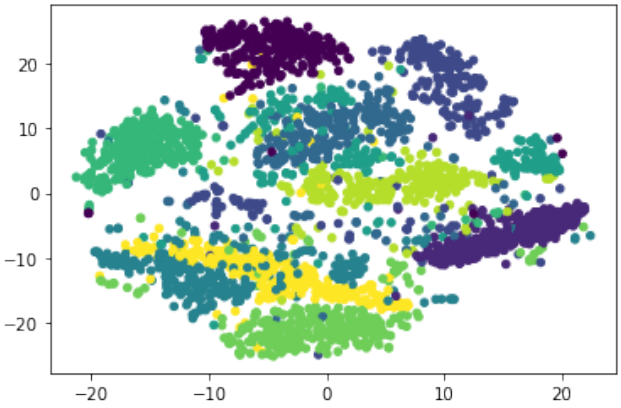
perplexity = 20



perplexity = 50



perplexity = 100



perplexity = 200