ML HW7

0510008 藍挺毓

I. Kernel Eigenfaces / Fisherfaces

A. Source Code

1. Simply load pgm files.

```python
def load_faces(split="Training", root="./Yale_Face_Database"):
    # config=["centerlight", "happy", "noglasses", "normal", "rig
    root = os.path.join(root, split)
    files = os.listdir(root)
    num_file = len(files) # 135
    label = np.zeros(num_file, dtype=int)
    faces = np.zeros((num_file, 97*115))
    for idx, f in enumerate(files):
        path = os.path.join(root, f)
        im = Image.open(path) # 195*213=45045
        im = im.resize((97,115))
        im = np.asarray(im)
        row, col = im.shape
        im = im.reshape(1,-1)
        faces[idx,:] = im

        label[idx] = int(files[idx][7:9])

    return faces.T, num_file, row, col, files, label # 231, 195
```

2. Main for Eigenfaces

follows the algorithm. First compute differences between each image with mean vector. Then, compute covariance matrix. The last step is use largest eigen values to choose eigen vectors. These eigen vectors is our eigenfaces, just need to resize back to image size.

```python
if __name__ == "__main__":
    faces, num_file, row, col, train_filename, train_label = load_faces() #(45045,num_file=135)
    test_faces, test_num_file, _, _ , test_filename, test_label= load_faces(split="Testing") #(45045, test_num_file=30)
    avg_face = (np.sum(faces, axis=1)/num_file).ravel() #(11155,)
    diff_faces = faces.copy() #(11155,135)
    print(avg_face.shape, diff_faces.shape, faces.shape)
    for i in range(row*col):
        diff_faces[i,:] = faces[i,:]-avg_face[i]

    #-----------------------------------------------------------
    S = diff_faces.dot(diff_faces.T)

    eigenvalue, eigenvector = np.linalg.eig(S)
    np.save("S_4.npy", S)
    np.save("eigenvalue_4.npy", eigenvalue)
    np.save("eigenvector_4.npy", eigenvector)
    # eigenvalue = np.load("eigenvalue_4.npy")
    # eigenvector = np.load("eigenvector_4.npy")
    # S = np.load("S_4.npy")
    #-----------------------------------------------------------

    sort_idx = np.argsort(eigenvalue)[::-1]
    selected_eigenvector = eigenvector[:,sort_idx[0:25]].real
    eigenfaces = selected_eigenvector.reshape(row, col, 25)
    eigenfaces = np.moveaxis(eigenfaces, -1, 0)
    show_eigenfaces(eigenfaces)
    show_reconstruction_faces(test_faces, test_num_file, eigenfaces)

    train_coeff = decompose_faces(eigenfaces, faces, num_file)
    test_coeff = decompose_faces(eigenfaces, test_faces, test_num_file)
    classify(test_coeff, test_filename, test_num_file, test_label, train_coeff, train_filename, num_file, train_label)
```

3. To reconstruct faces, we need to decompose input data into eigenfaces. We can use dot product to find out coefficient between different eigenfaces for reconstruction.

```
def decompose_faces(eigenfaces, faces, num):
    coeff = np.zeros((num, 25))
    for i in range(num):
        for eig in range(25):
            coeff[i,eig] = ( faces[:,i].reshape(1,-1) ).dot( eigenfaces[eig,:,:].reshape(-1,1) )
    return coeff
```

4. To build reconstruction faces, just use pre-calculate coefficient. Add each eigenfaces up with weight of coefficient.

```
def build_recon_face(eigenfaces, coeff, i):
    result = np.zeros((row, col))
    for eig in range(25):
        result += eigenfaces[eig,:,:]*coeff[i,eig]
    return result
```

5. Use k-nearest neighbor to find testing images' class. I choose 5 here. It is a fine-tuned parameters. With k=5, I get best performance.

```
def classify(test_coeff, test_filename, test_num, test_label, train_coeff, train_filename, train_num, train_label):
    k = 5
    predict = np.zeros(test_num, dtype=int)
    error = 0

    dist = np.zeros(train_num)
    for i in range(test_num):
        for j in range(train_num):
            dist[j] = np.linalg.norm(test_coeff[i]-train_coeff[j])
        min_dist = np.argsort(dist)[:k]
        k_predict = train_label[min_dist]
        predict[i] = np.argmax(np.bincount(k_predict))
        if test_label[i]!=predict[i]:
            error += 1
    print("error num:", error, "error rate:", error/test_num)
```

6. Main for fisherfaces: As to fisherface, I first do pca, which is exactly the same as above. The different part here is do LDA.

```
if __name__ == "__main__":
    train_faces, train_num, row, col, train_filename, train_label = load_faces() #(45045,num_file=135)
    test_faces, test_num, _, _ , test_filename, test_label= load_faces(split="Testing") #(45045, test_num_file=30)

    eigenfaces = pca(train_faces, train_num)
    train_coeff = decompose_faces(eigenfaces.reshape(25,row,col), train_faces, train_num)

    w = lda(train_coeff, train_label, 25)
    fish_faces = np.dot(eigenfaces.T ,w).T

    show_eigenfaces(fish_faces.reshape(25,row,col))
    show_reconstruction_faces(test_faces, test_num, fish_faces.reshape(25,row,col))

    test_coeff = decompose_faces(eigenfaces.reshape(25,row,col), test_faces, test_num)
    classify(test_coeff, test_filename, test_num, test_label, train_coeff, train_filename, train_num, train_label)
```

7. LDA

   It required to compute Sb and Sw. Sb means how far points are within a class. In other words, between-class covariance matrix. Sw, within-class covariance matrix, means how far different classes are.

```
class_mean = np.zeros((row*col, 15))
Sw = np.zeros((row*col, row*col))
Sb = np.zeros((row*col, row*col))
for c in range(15):
    mask = train_label==(c+1)
    c_faces = train_faces[:,mask] #(45045,9)
    class_mean[:,c] = (np.sum(c_faces, axis=1)/len(c_faces[0])).ravel()

    diff = c_faces.copy() #(45045,9)
    for i in range(row*col):
        diff[i,:] -= class_mean[i,c]
    Sw += diff.dot(diff.T)

    diff_mean = np.subtract(class_mean[:,c], avg_face)
    Sb += diff_mean.dot( diff_mean.T ) * len(c_faces[0])
```

   Then, compute inverse of Sw, and do inner product with Sb. After this, just simply calculate eigenvectors and eigenvalues. The rest of the story are the same. Simply pick 25 eigenvectors with largest eigenvalues.

```
Sw_inv = np.linalg.inv(Sw)
np.save("Sw_inv_3.npy", Sw_inv)
# Sw_inv = np.load("Sw_inv.npy")

S = Sw_inv.dot(Sb)
eigenvalue, eigenvector = np.linalg.eig(S)
```

```
sort_idx = np.argsort(eigenvalue)[::-1]
selected_eigenvector = eigenvector[:,sort_idx[0:25]].real
```

   Fisherface is found eigenvector inner product with found eigenfaces

```
w = lda(train_coeff, train_label, 25)
fish_faces = np.dot(eigenfaces.T ,w).T
```

8. The rest part of fisher faces are calling PCA's function to show reconstruction faces, and do k-nearest neighbor.

```
show_eigenfaces(fish_faces.reshape(25,row,col))
show_reconstruction_faces(test_faces, test_num, fish_faces.reshape(25,row,col))

test_coeff = decompose_faces(eigenfaces.reshape(25,row,col), test_faces, test_num)
classify(test_coeff, test_filename, test_num, test_label, train_coeff, train_filename, train_num, train_label)
```

9. Kernel PCA: Just change this part of code in PCA. Calculate kernel before calculate eigen.

```
K = kernel(faces, type="RBF")
print(K.shape)
M = row*col
MM = np.zeros((M, M))/M
cenK = K - MM.dot(K) - K.dot(MM) + MM.dot(K).dot(MM)

eigenvalue, eigenvector = np.linalg.eig(K)
```

10. Kernel functions are same as previous homework. Just write kernel equation to code.

```
def kernel(X, mode="RBF"):
    if mode=="linear":
        K = X.dot(X.T)

    elif mode=="RBF":
        gamma = 0.0001
        dist = cdist(X,X,'sqeuclidean')
        K = np.exp( -gamma * dist)

    return K
```

11. Kernel LDA:

    I added kernel before summing up all Sw and Sb

```
Sw += kernel( np.dot((Xi-meanClass).T, (Xi-meanClass)) )
Sb += kernel( (meanClass - meanTotal).reshape(-1,1)*n )
```

B. Result

1. PCA: eigenfaces and reconstruction

classification error rate:

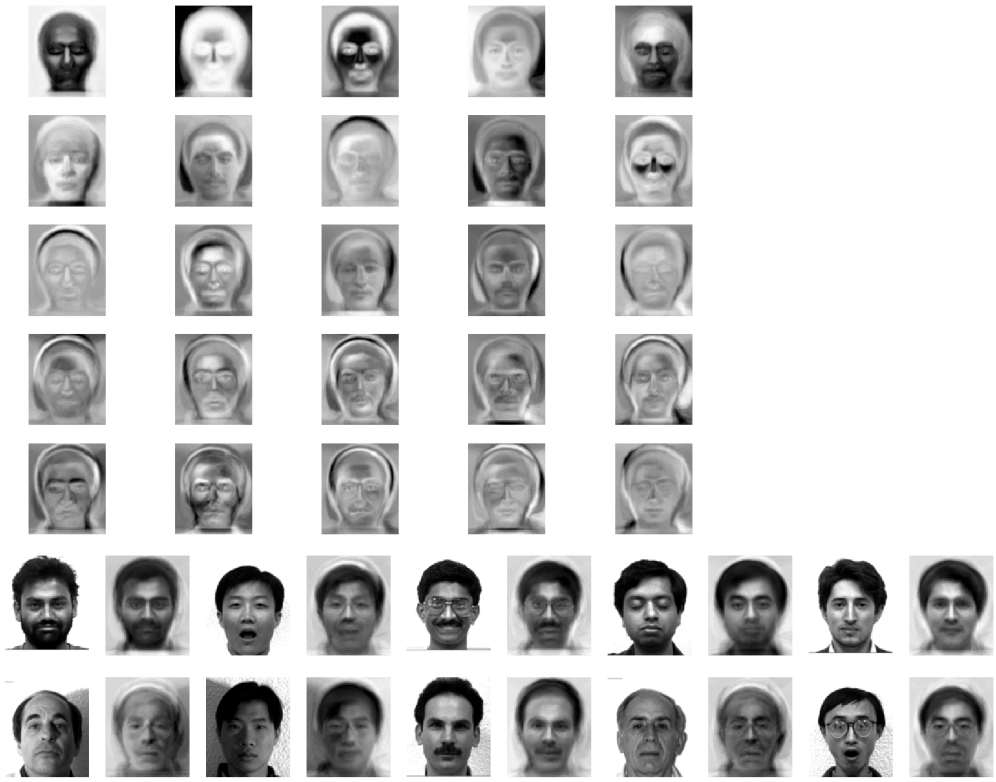
```
error rate: 0.1
```

2. LDA: fisherfaces and reconstruction

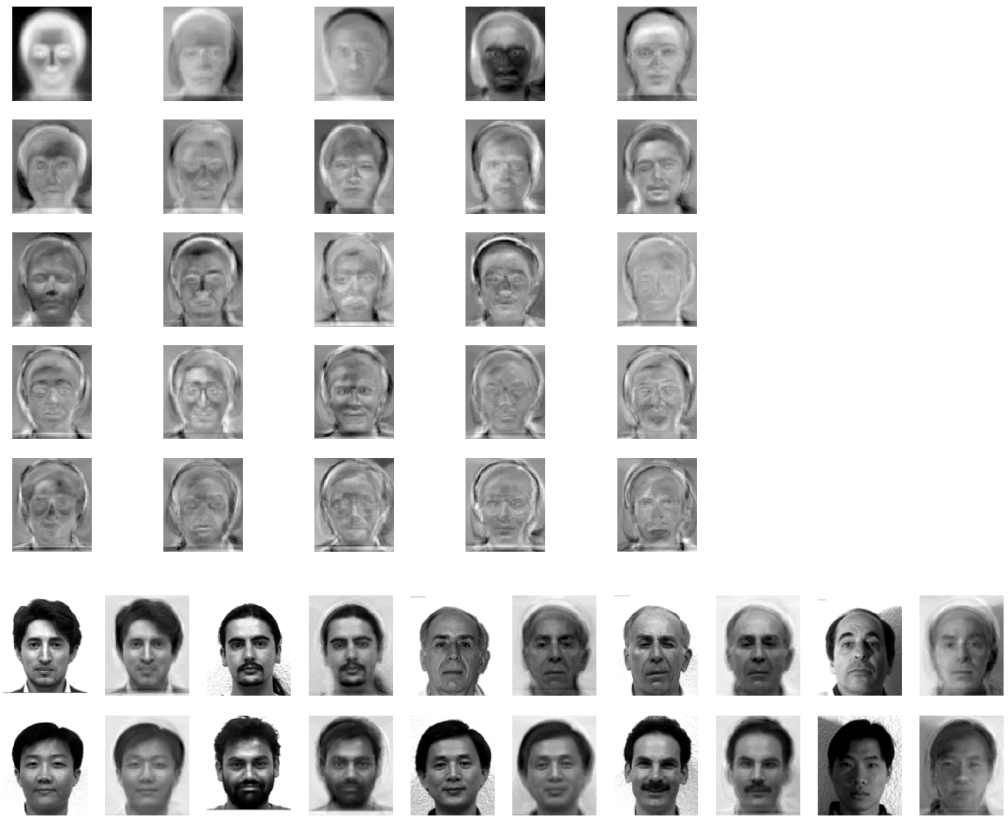
```
error rate: 0.1
```

3. Kernel PCA

- RBF



- Linear



4. Kernel LDA
   - RBF

   - linear

C. 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. LDA has a better performance than PCA by looking at reconstruction result.
3. It can be easily told from reconstruction faces that they are fake faces. Nevertheless, it still make good prediction on which subject they belong to. PCA and LDA reserve most important information, especially LDA. LDA knows labels in advance, and knowing what are images that would like to be project to be closed in low-dimensional space.
4. 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.
5. There is no huge difference between using linear or RBF. However, it is really crucial to choose RBF's gamma, or it would results in all gray.

II. t-SNE

A. Source Code

1. Symmetric SNE and t-SNE are really similar. Just need to modify $q_{i,j}$ and gradient.
2. Symmetric SNE

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

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

3. t-SNE

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

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

4. My modification

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

```
num = np.exp(-1.0 * np.add(np.add(num, sum_Y).T, sum_Y))
```

5. Visualize

   save distribution for each 10 iterations.

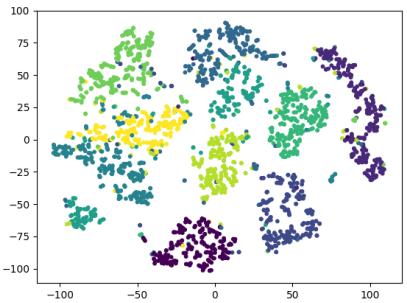
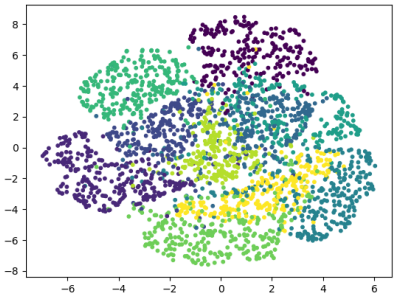
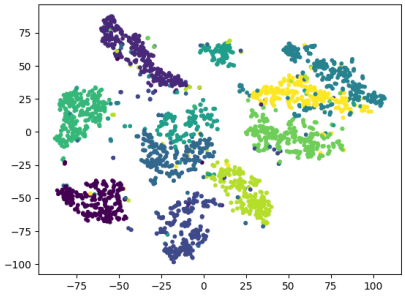
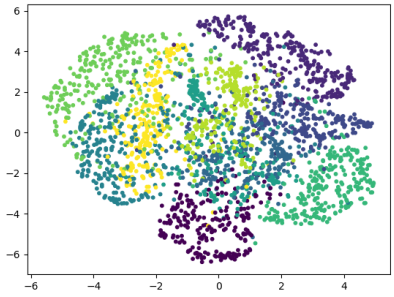
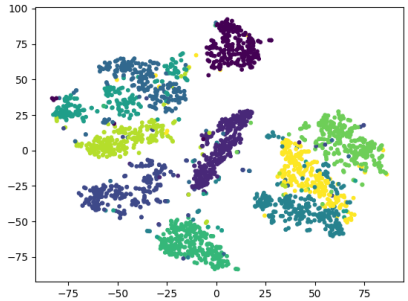
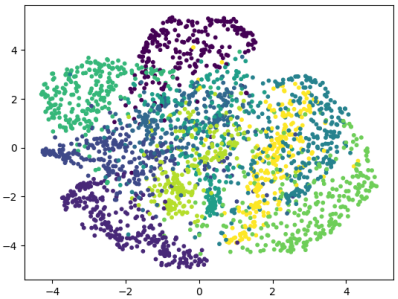
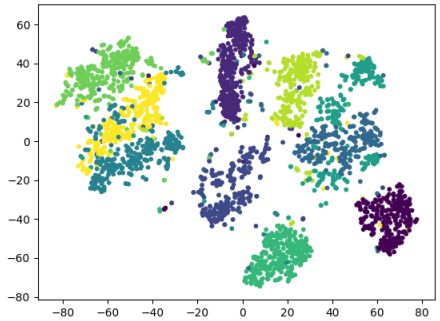
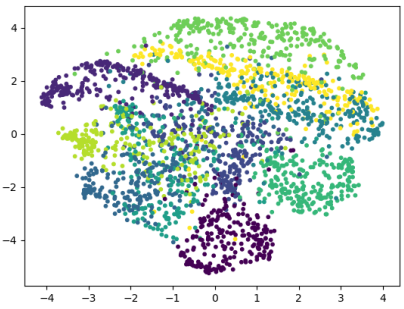
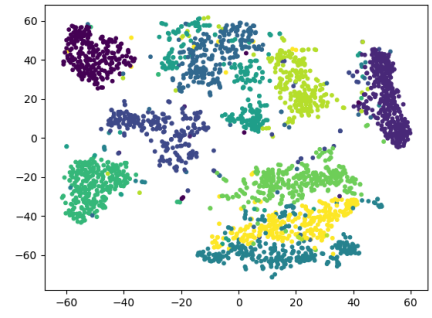
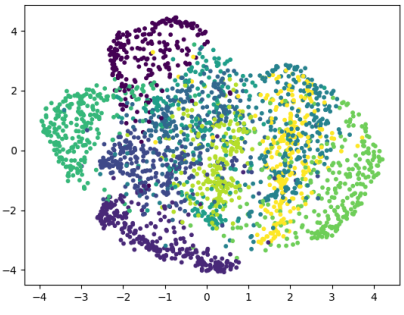
```
if iter%10==0 or iter==(max_iter-1):
    save_visualize(iter, Y)
```

```
def save_visualize(iter, Y):
    pylab.clf()
    pylab.scatter(Y[:, 0], Y[:, 1], 10, labels)
    pylab.savefig( os.path.join(output_dir, "%04d"%iter) )
```

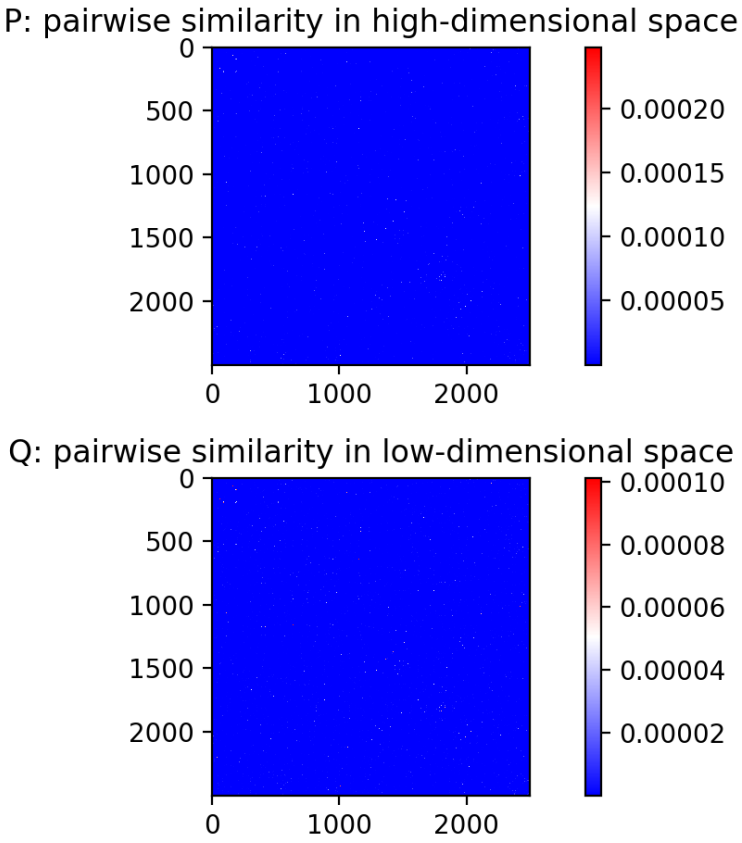
6. Pairwise similarity

```
plt.subplot(211)
plt.title('P: pairwise similarity in high-dimensional space')
plt.imshow(P, cmap='bwr')
plt.colorbar()

plt.subplot(212)
plt.title('Q: pairwise similarity in low-dimensional space')
plt.imshow(Q, cmap='bwr')
plt.colorbar()

plt.tight_layout()
plt.savefig( os.path.join(output_dir, "similarity") )
```

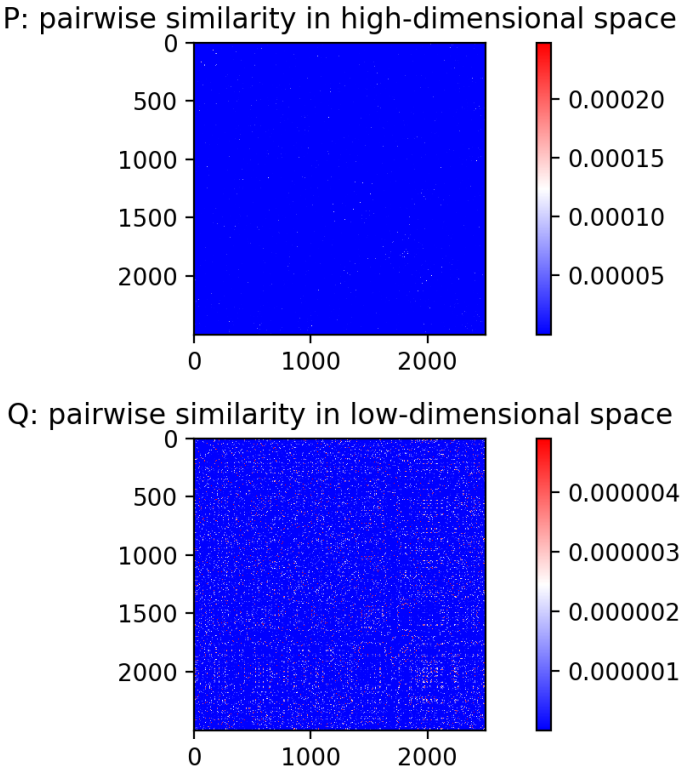
B. Result

perplexity	t-SNE	Symmetric SNE
10		
20		
30		
40		
50		

7. t-SNE



Symmetric SNE



C. Discussion

1. 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.
2. Via gif, we can see how data are being grouped through training. t-SNE separates different classes into different place in 2D really fast, then, modifies distribution to a more general way. As to symmetric SNE, it just move data slightly in low-dimensional space compares to SNE. Then, it modify slightly to makes the distribution better.
3. 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.
4. It can be seen from the result of both symmetric SNE and t-SNE. As perplexity go smaller, only close neighbor has influence on a single data. Opposite to above mentioned, as perplexity goes larger, the structure may become more clear.
5. Perplexity has greater effect on t_SNE than symmetric SNE, since symmetric SNE has crowded problem.