ML-HW7

資工所博士班
周芝妤 0886004

■ a. code with detailed explanations (40%)

- Kernel Eigenfaces

Part1 (10%) Also, simply explain how you do PCA & LDA

✓ PCA

The original size of pictures is 231*195, so the X@X.T will be a large matrix. We need to find the eigenvalues and eigenvectors of X@X.T first. By ruling out those eigenvectors which corresponding eigenvalues smaller than 0, we get the partial eigenvectors of X@X.T.

```
# PCA function
def pca(x,num_dim=None):
    x_mean = np.mean(x, axis=1).reshape(-1, 1)
    x_center = x - x_mean

    # PCA
    eigenvalues, eigenvectors = np.linalg.eig(x_center.T @ x_center)
    sort_index = np.argsort(-eigenvalues)
    if num_dim is None:
        for eigenvalue, i in zip(eigenvalues[sort_index], np.arange(len(eigenvalues))):
            if eigenvalue <= 0:
                sort_index = sort_index[:i]
                break
    else:
        sort_index=sort_index[:num_dim]

    eigenvalues=eigenvalues[sort_index]
    # from X.T@X eigenvector --> X@X.T eigenvector
    eigenvectors=x_center@eigenvectors[:, sort_index]
    eigenvectors_norm=np.linalg.norm(eigenvectors,axis=0)
    eigenvectors= eigenvectors/ eigenvectors_norm

    return eigenvalues,eigenvectors,x_mean
```

By using the “pca” function to reduce dimension from 231*195 to 134, we can get the partial eigenvectors us transform matrix.

```
# PCA

# The size of the picture is 231*195
H=231
W=195
filepath=os.path.join('Yale_Face_Database','Training')
X,y=read_data(filepath,H,W)

eigenvalues,eigenvectors,X_mean=pca(X)

# Calculate transform matrix
U=eigenvectors.copy()
print('U shape: {}'.format(U.shape))

# Show the top 25 eigenface
plot_eigenface(U,25,H,W)
```

```
U shape: (45045, 134)
```

```
# reduce the dimension (projection)
Z=U.T@(X-X_mean)

# recover
X_recover=U@Z+X_mean
show_reconstruction(X,X_recover,10,H,W)

# accuracy
filepath=os.path.join('Yale_Face_Database','Testing')
X_test,y_test=read_data(filepath,H,W)
acc = accuracy(X_test,y_test,Z,y,U,X_mean,3)
print('acc: {:.2f}%'.format(acc*100))
```

```
acc: 83.33%
```

✓ LDA

The basic code of LDA is to calculate within- and between-class scattering matrix according to the formula bellow, then calculate the eigenvalues and eigenvectors by using Sw and Sb.

$$\boldsymbol{S}^{(\mathrm{w})} = \frac{1}{2}\sum_{i,i'=1}^{n} Q_{i,i'}^{(\mathrm{w})}\left(\boldsymbol{x}_i - \boldsymbol{x}_{i'}\right)\left(\boldsymbol{x}_i - \boldsymbol{x}_{i'}\right)^{\top}$$
$$\boldsymbol{S}^{(\mathrm{b})} = \frac{1}{2}\sum_{i,i'=1}^{n} Q_{i,i'}^{(\mathrm{b})}\left(\boldsymbol{x}_i - \boldsymbol{x}_{i'}\right)\left(\boldsymbol{x}_i - \boldsymbol{x}_{i'}\right)^{\top}$$

```python
# LDA function

def lda(x,y,num_dim=None):
    N=x.shape[0]
    x_mean = np.mean(x, axis=1).reshape(-1, 1)

    C_mean = np.zeros((N, 15))  # 15 classes's means
    for i in range(x.shape[1]):
        C_mean[:, y[i]] += x[:, y[i]]
    C_mean = C_mean / 9

    # within-class scatter Sw
    S_within = np.zeros((N, N))
    for i in range(x.shape[1]):
        d = x[:, y[i]].reshape(-1,1) - C_mean[:, y[i]].reshape(-1,1)
        S_within += d @ d.T

    # between-class scatter Sb
    S_between = np.zeros((N, N))
    for i in range(15):
        d = C_mean[:, i].reshape(-1,1) - x_mean
        S_between += 9 * d @ d.T

    eigenvalues,eigenvectors=np.linalg.eig(np.linalg.inv(S_within)@S_between)
    sort_index=np.argsort(-eigenvalues)
    if num_dim is None:
        sort_index=sort_index[:-1]  # reduce 1 dim
    else:
        sort_index=sort_index[:num_dim]

    eigenvalues=np.asarray(eigenvalues[sort_index].real,dtype='float')
    eigenvectors=np.asarray(eigenvectors[:,sort_index].real,dtype='float')

    return eigenvalues,eigenvectors
```

I did the PCA first to reduce the dimension 231*195 to a lower dimension to avoid Sw to be the singular matrix, then I did LDA to get result.

```
# LDA
H=231
W=195
filepath=os.path.join('Yale_Face_Database','Training')
X,y=read_data(filepath,H,W)

# try
eigenvalues_pca,eigenvectors_pca,X_mean=pca(X,num_dim=28)
X_pca=eigenvectors_pca.T@(X-X_mean)
eigenvalues_lda,eigenvectors_lda=lda(X_pca,y)

# Transform matrix
U=eigenvectors_pca@eigenvectors_lda
print('U shape: {}'.format(U.shape))

# show top 25 eigenface
plot_eigenface(U,25,H,W)
```

```
U shape: (45045, 27)
```

```
# reduce dim (projection)
Z=U.T@X

# recover
X_recover=U@Z+X_mean
show_reconstruction(X,X_recover,10,H,W)

# accuracy
filepath = os.path.join('Yale_Face_Database', 'Testing')
X_test, y_test = read_data(filepath, H, W)
acc =accuracy(X_test, y_test, Z, y, U, X_mean, 5)
print('acc: {:.2f}%'.format(acc * 100))
```

```
acc: 76.67%
```

I tried different “num_dim” to reduce the dimension by PCA from range 26-70, and I got the best performance by setting the “num_dim” as 28 (76.67%)

Both PCA and LDA used the same function to plot the eigenfaces and fisherfaces by using the "matplotlib.pyplot" library. The code shown as following:

```
# plot all fiures
def plot_eigenface(X,number,H,W):
    # plot the eigenface
    n=int(number**0.5)
    matplotlib.use("Qt5Agg")
    for i in range(number):
        plt.subplot(n,n,i+1)
        plt.imshow(X[:,i].reshape(H,W),cmap='gray')
        plt.xticks(fontsize=5)
        plt.yticks(fontsize=5)

    plt.subplots_adjust(left=0.125,
                        bottom=0.1,
                        right=1.5,
                        top=1.5,
                        wspace=0.2,
                        hspace=0.35)
    plt.get_current_fig_manager().window.showMaximized()
    plt.show()

def show_reconstruction(X,X_recover,num,H,W):
    rand_init=np.random.choice(X.shape[1],num)
    for i in range(num):
        plt.subplot(2,num,i+1)
        plt.imshow(X[:,rand_init[i]].reshape(H,W),cmap='gray')
        plt.xticks(fontsize=8)
        plt.yticks(fontsize=8)
        plt.subplot(2,num,i+1+num)
        plt.imshow(X_recover[:,rand_init[i]].reshape(H,W),cmap='gray')
        plt.xticks(fontsize=8)
        plt.yticks(fontsize=8)

    plt.subplots_adjust(left=0.3,
                        bottom=0.1,
                        right=3.0,
                        top=0.9,
                        wspace=0.2,
                        hspace=0.35)
    plt.get_current_fig_manager().window.showMaximized()
    plt.show()
```

Part2: Use PCA and LDA to do face recognition, and compute the performance.

Both PCA and LDA used the same function to calculate accuracy, and I used k nearest neighbor to classify the testing images. I tried k=5 and get acc: 83.33% (PCA) and acc: 76.67% (LDA with reducing the dimension to 28).

```
def accuracy(x_test,y_test,z_train,y_train,U,x_mean=None,k=3):
    '''
    using k-nn to predict x_test's label
    x_test:  (H*W, # pics) array
    y_test:   (# pics) array
    z_train:  (low-dim, #pics) array
    y_train:  (# pics) array
    U: Transform matrix
    x_mean:  using when estimate eigenface
    k: k of k-nn

    '''
    if x_mean is None:
        x_mean=np.zeros((X_test.shape[0],1))

    # reduce dim (projection)
    z_test=U.T@(x_test-x_mean)

    # k-nn
    predicted_y=np.zeros(z_test.shape[1])
    for i in range(z_test.shape[1]):
        distance=np.zeros(z_train.shape[1])
        for j in range(z_train.shape[1]):
            distance[j]=np.sum(np.square(z_test[:,i]-z_train[:,j]))
        sort_index=np.argsort(distance)
        nearest_neighbors=y_train[sort_index[:k]]
        unique, counts = np.unique(nearest_neighbors, return_counts=True)
        nearest_neighbors=[k for k,v in sorted(dict(zip(unique, counts)).items(), key=lambda item: -item[1])]
        predicted_y[i]=nearest_neighbors[0]

    acc=np.count_nonzero((y_test-predicted_y)==0)/len(y_test)
    return acc
```

Part3: Kernel PCA and LDA

I implemented 3 kernel in both PCA and LDA, there are: 'polynomial', 'linear', 'sigmoid'. I didn't used rgf kernel because it raised MemoryError. Here I only show the different codes between kernel functions and original functions here, the rest of the codes are basically same:

✓ Kernel PCA

```
def kernel_pca(x, num_dim=50, kernel = 'polynomial', gamma = 2, C=5, alpha=2, d_poly=2):
    x_mean = np.mean(x, axis=1).reshape(-1, 1)
    x_center = x - x_mean

    if kernel == 'polynomial':
        """
        k(x, y) = (alpha * <x, y> + c)^d
        Hiperparámetros: alpha, c, d_poly
        """
        K = (alpha * (x_center.T@x_center) + C)**d_poly

        # Centering the symmetric NxN kernel matrix.
        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)

    elif kernel == 'linear':
        """
        k(x, y) = <x, y> + c
        Hiperparámetros: c
        """
        # Computing the MxM kernel matrix.
        K = x_center.T@x_center + C

        # Centering the symmetric NxN kernel matrix.
        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)

    elif kernel == 'sigmoid':
        """
        k(x, y) = tanh( alpha * <x, y> + c)
        Hiperparámetros: alpha, c
        """
        K = np.tanh(alpha * (x_center.T@x_center) + C)

        # Centering the symmetric NxN kernel matrix.
        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)
```

The equations of 'polynomial', 'linear', 'sigmoid' kernels are simply shown in the red frame.

✓ Kernel LDA

I followed the formula to implement the kernels, I first calculated the kernel of data then used “K” to calculate Sw and Sb.

$$J(\mathbf{w}) = \frac{\mathbf{w}^{\mathrm{T}}\mathbf{S}_B^{\phi}\mathbf{w}}{\mathbf{w}^{\mathrm{T}}\mathbf{S}_W^{\phi}\mathbf{w}},$$

where

$$\mathbf{S}_B^{\phi} = \left(\mathbf{m}_2^{\phi} - \mathbf{m}_1^{\phi}\right)\left(\mathbf{m}_2^{\phi} - \mathbf{m}_1^{\phi}\right)^{\mathrm{T}}$$

$$\mathbf{S}_W^{\phi} = \sum_{i=1,2}\sum_{n=1}^{l_i}\left(\phi(\mathbf{x}_n^i) - \mathbf{m}_i^{\phi}\right)\left(\phi(\mathbf{x}_n^i) - \mathbf{m}_i^{\phi}\right)^{\mathrm{T}},$$

and

$$\mathbf{m}_i^{\phi} = \frac{1}{l_i}\sum_{j=1}^{l_i}\phi(\mathbf{x}_j^i).$$

```
def kernel_lda(x,y,num_dim=None,kernel ='linear',gamma = 2, C=5, alpha=2, d_poly=3):
    if kernel == 'polynomial':
        """
        k(x, y) = (alpha * <x, y> + c)^d
        Hiperparámetros: alpha, c, d_poly
        """
        K = (alpha * (x.T@x) + C)**d_poly

        # Centering the symmetric NxN kernel matrix.
        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)

    elif kernel == 'linear':
        """
        k(x, y) = <x, y> + c
        Hiperparámetros: c
        """
        # Computing the MxM kernel matrix.
        K = x.T@x + C

        # Centering the symmetric NxN kernel matrix.
        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)

    elif kernel == 'sigmoid':
        """
        k(x, y) = tanh( alpha * <x, y> + c)
        Hiperparámetros: alpha, c
        """
        K = np.tanh(alpha * (x.T@x) + C)

        # Centering the symmetric NxN kernel matrix.
        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)
```

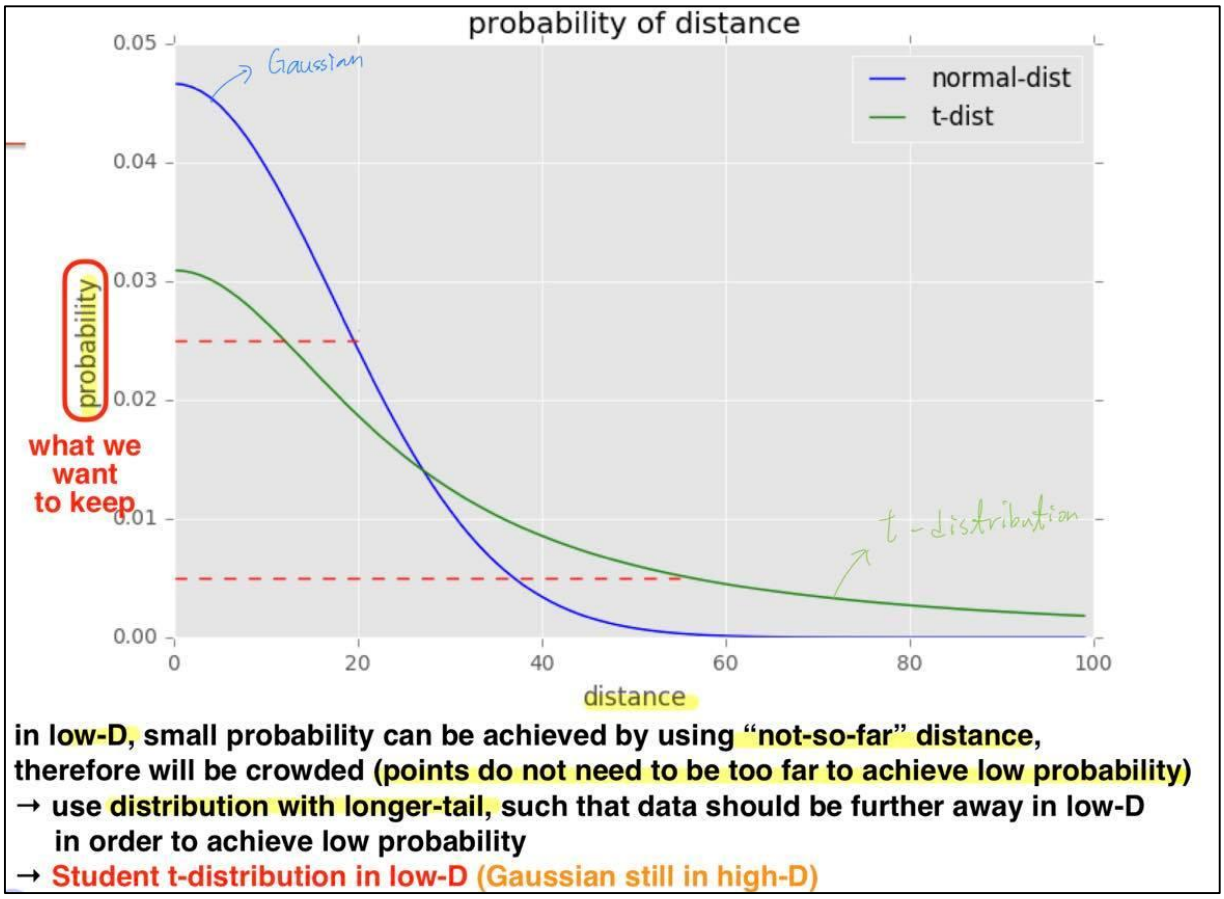
Same as kernel PCA, the equations of 'polynomial', 'linear', 'sigmoid' kernels are simply shown in the red frame.

Reference: https://en.wikipedia.org/wiki/Kernel_method

- t-SNE

Part 1: Try to modify the code a little bit and make it back to symmetric (show the formula of tsne & ssne)

The biggest difference between t-SNE and symmetric SNE is that the symmetric SNE calculate the pairwise Pij compare to the original SNE algorithm, and t-SNE use the t-distribution instead of Gaussian distribution in the low-dimension. t-SNE could solve the “crowed problem” in the low-d space because it is a long-tail distribution so the probability for far data appearance is higher than Gaussian distribution.



The different equations are showed below.

t-SNE:

$$q_{ij} = \frac{(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|\mathbf{y}_k - \mathbf{y}_l\|^2)^{-1}}$$

(A) t-distribution of qij (low-d)

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

(B) calculating the gradient

Symmetric SNE:

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|\mathbf{x}_i - \mathbf{x}_k\|^2 / 2\sigma_i^2)} \qquad p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

(C) pairwise similarity

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

(D) calculating the gradient

✓ t-SNE

The original code from reference code shown the calculation of t-distribution (A) and the gradient (B).

```
# Run iterations
for iter in range(max_iter):

    # 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)
```

✓ Symmetric SNE

To modify the t-SNE code to Symmetric SNE, we need to calculate the pairwise similarity by the exponential of 2-norm (C), and we also need to change the code of the calculating of gradient (D).

```
# Run iterations
for iter in range(max_iter):

    # Compute pairwise affinities
    sum_Y = np.sum(np.square(Y), 1)
    num = -2. * np.dot(Y, Y.T)
    # 2-norm
    num=np.exp(-(sum_Y+ sum_Y.reshape(-1,1) + num))
    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)
        dY[i, :] = np.dot(PQ[i,:],Y[i,:]-Y)
```

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

I used the same way to visualize the clustering performance in both t-SNE and symmetric SNE. I used "Y_list" to record the output after 10 iterations.

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

I used the "pylab" library to plot the scatter figure of each outcome in "Y_list" and save them as png files, then used "imageio" library to make the gif files from these png files.

```
import imageio
import os

filenames = []

for i in  range(len(Y_list)):
    # plot the line chart
    pylab.figure(figsize=(15,10))
    pylab.scatter(Y_list[i][:, 0], Y_list[i][:, 1], 20, labels)

    # create file name and append it to a list
    filename = f'{i}.png'
    filenames.append(filename)

    # save frame
    pylab.savefig(filename)
    pylab.close()
# build gif
with imageio.get_writer('tsne_p50_gif.gif', mode='I') as writer:
    for filename in filenames:
        image = imageio.imread(filename)
        writer.append_data(image)

# Remove files
for filename in set(filenames):
    os.remove(filename)
```

Part3: Visualize the distribution of pairwise similarities:

I also used the same code to visualize the pairwise similarities in both t-SNE and symmetric SNE. Just use "pylab.subplot" to visualize P and Q similarities (high-d and low-d).

```
def plot_similarity(P,Q):
    pylab.subplot(2,1,1)
    pylab.title('Symmetric SNE high-dimension')
    pylab.hist(P.flatten(),bins=50,log=True)
    pylab.subplot(2,1,2)
    pylab.title('Symmetric SNE low-dimension')
    pylab.hist(Q.flatten(),bins=50,log=True)
    #pylab.subplot_tool()
    pylab.subplots_adjust(wspace = 0.5,hspace = 0.5)
    pylab.show()
```

```
def plot_similarity(P,Q):
    pylab.subplot(2,1,1)
    pylab.title('tSNE high-dimension')
    pylab.hist(P.flatten(),bins=50,log=True)
    pylab.subplot(2,1,2)
    pylab.title('tSNE low-dimension')
    pylab.hist(Q.flatten(),bins=50,log=True)
    #pylab.subplot_tool()
    pylab.subplots_adjust(wspace = 0.5,hspace = 0.5)
    pylab.show()
```

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

I tried perplexity: [5, 20, 30, 50] in both t-SNE and symmetric SNE cause the 5 ~ 50 perplexity is usually implemented in the papers. When the lower perplexity is used, the sensitivity to the small area is also lower, it means only few neighbors have impact, a big cluster might become numerous small clusters. On the other hand, we usually use larger perplexity setting when the data is large, and it showed better global structure of clusters but may cause the ambiguity between clusters. The formula of perplexity is shown below:

$$Perp(P_i) = 2^{H(P_i)}$$

Reference: https://mropengate.blogspot.com/2019/06/t-sne.html?fbclid=IwAR1KCPvXqIkMl3tcH_6ydslXHBy8Ba9lBu37H7L9M0Ci3SYvqefTI1S6hSs

■ b. experiments settings and results (35%) & discussion (15%)

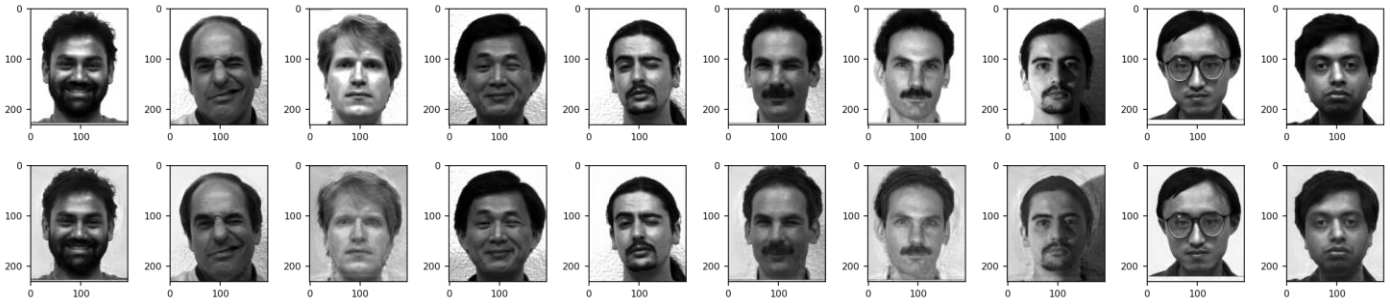
- Kernel Eigenfaces

Part 1 & Part 2:

✓ PCA

Here shows the eigenfaces of the columns in transform matrix U.

Here is the reconstruction result of PCA:

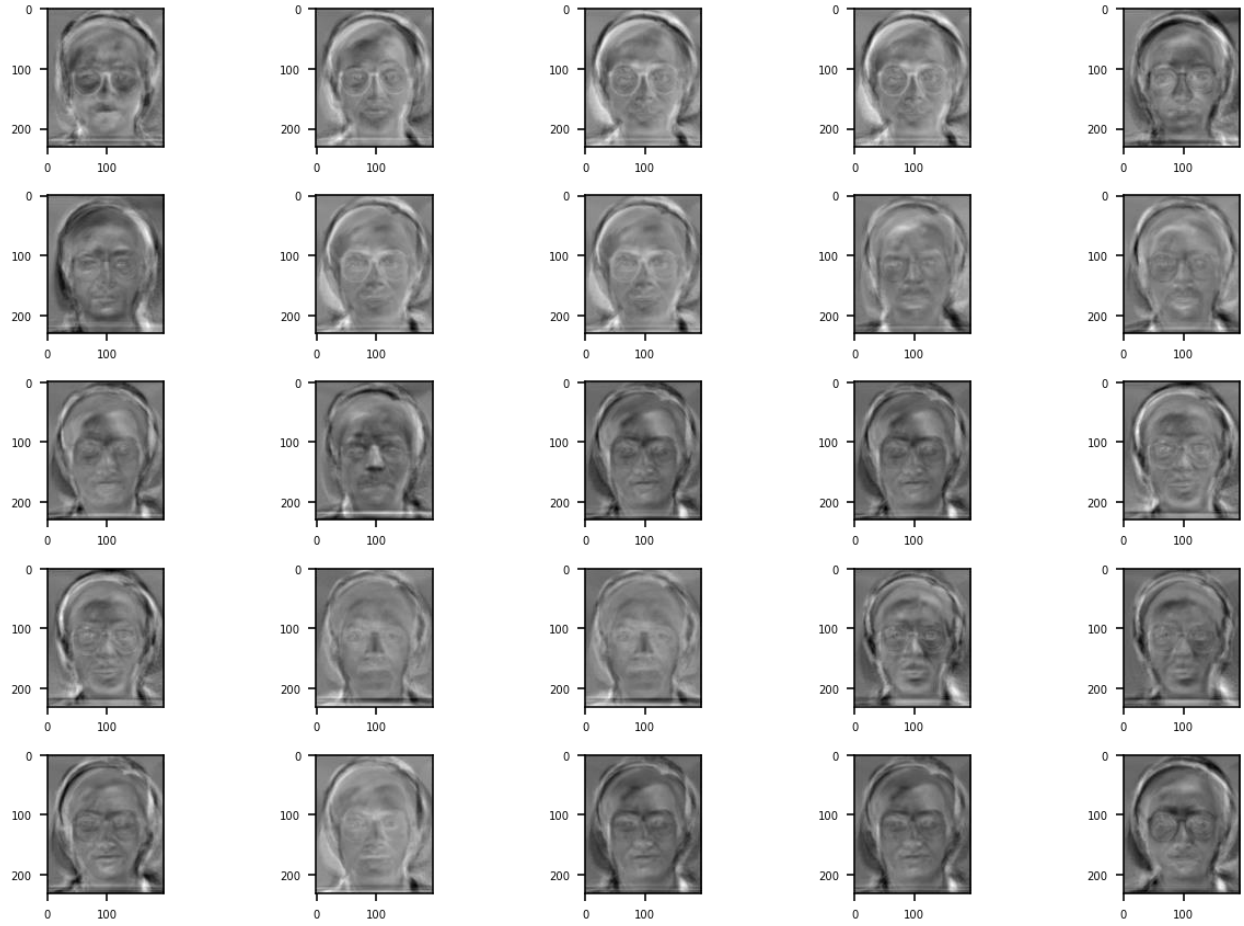


The top row are the original images; the second row are the reconstructed faces. The main idea of doing this task is to project original data X into a lower dimension by U and then project back into the original dimension by U.T, then add the X_mean back cause we centerize X in the beginning.

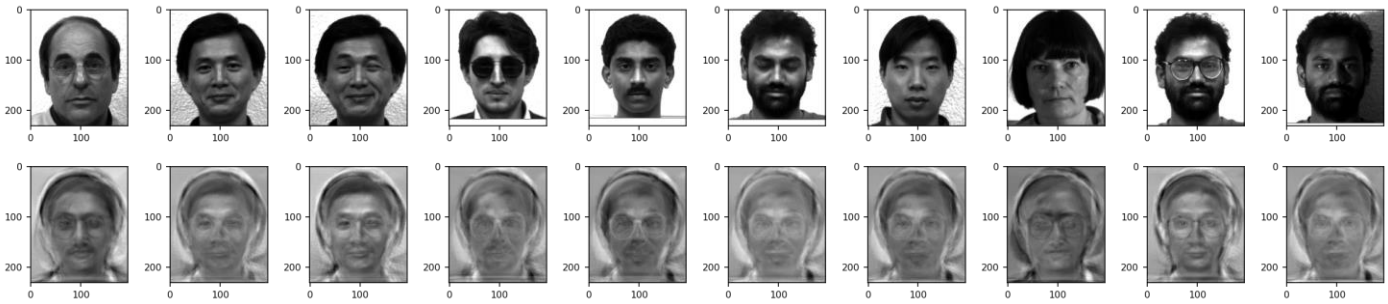
I get 83.33% accuracy by using K=3 (kNN).

✓ LDA

Here shows the fisherfaces of the columns in transform matrix U.



Here is the reconstruction result of LDA:



Same as PCA case, the top row are the original images; the second row are the reconstructed faces.

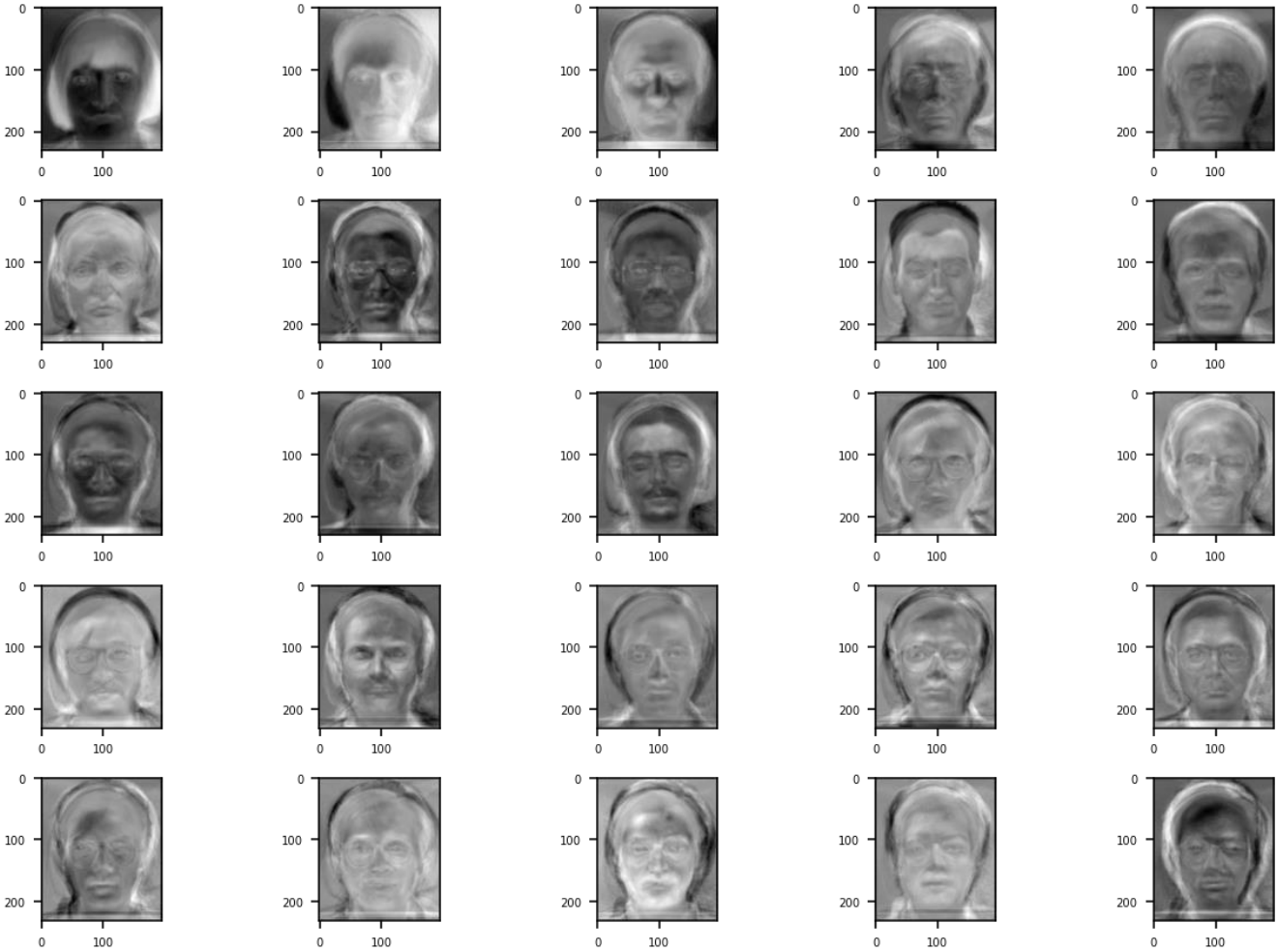
I get 76.67% accuracy by using K=5 (kNN) and setting the reduced dimension as 28.

(I tried different conditions like reduced dimension = 70, 35, 30, 29, 28, 26 and I got acc = 43.33, 33.33, 53.33, 46.67, 76.67, 16.67)

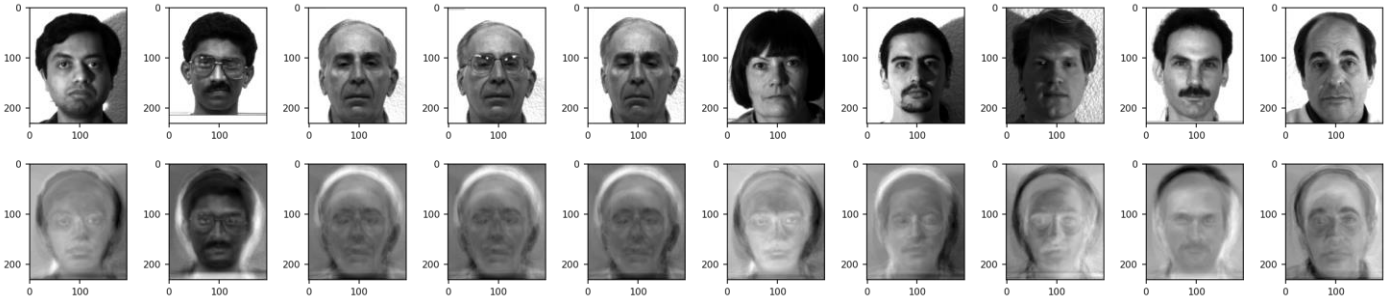
Part3 :

- ✓ Kernel PCA
- ➢ Polynomial case:

Here shows the eigenfaces of the columns in transform matrix U.



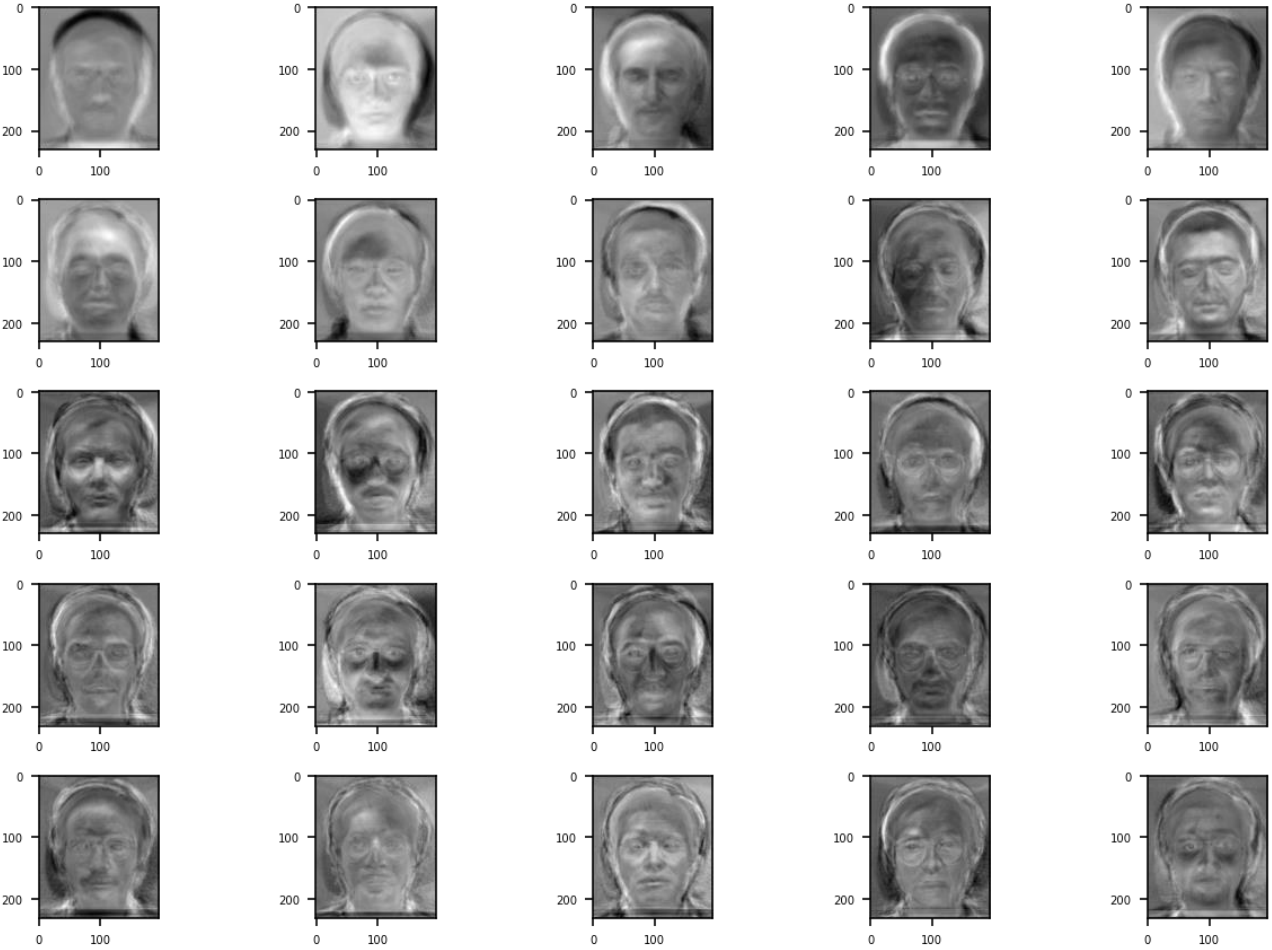
Here is the reconstruction result:



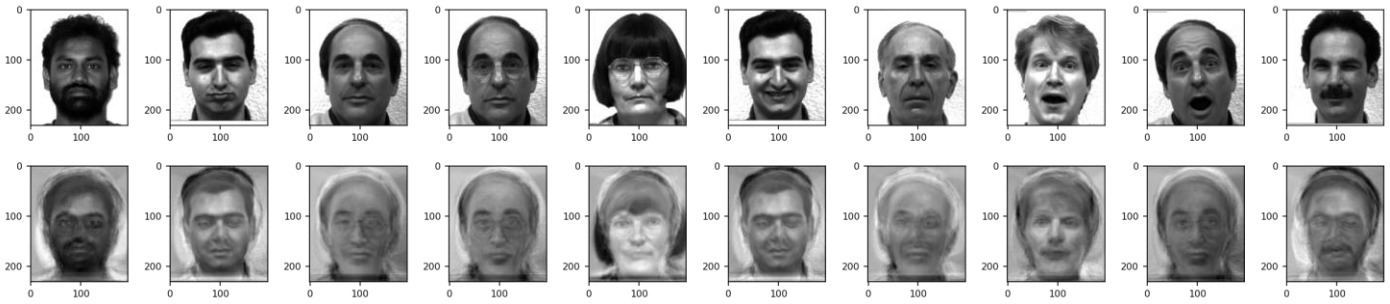
I get 80.00% accuracy by using K=3 (kNN) and setting the reduced dimension as 50, alpha=2, d_poly=2.

- Sigmoid case:

Here shows the eigenfaces of the columns in transform matrix U.



Here is the reconstruction result:

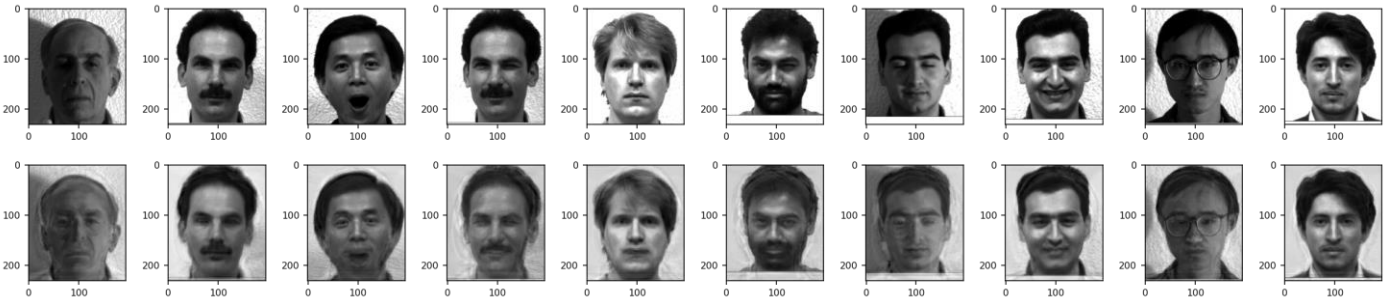


I get 86.67% accuracy by using K=3 (kNN) and setting the reduced dimension as 50, alpha=2, C=5.

➢ Linear case:

Here shows the eigenfaces of the columns in transform matrix U.

Here is the reconstruction result:

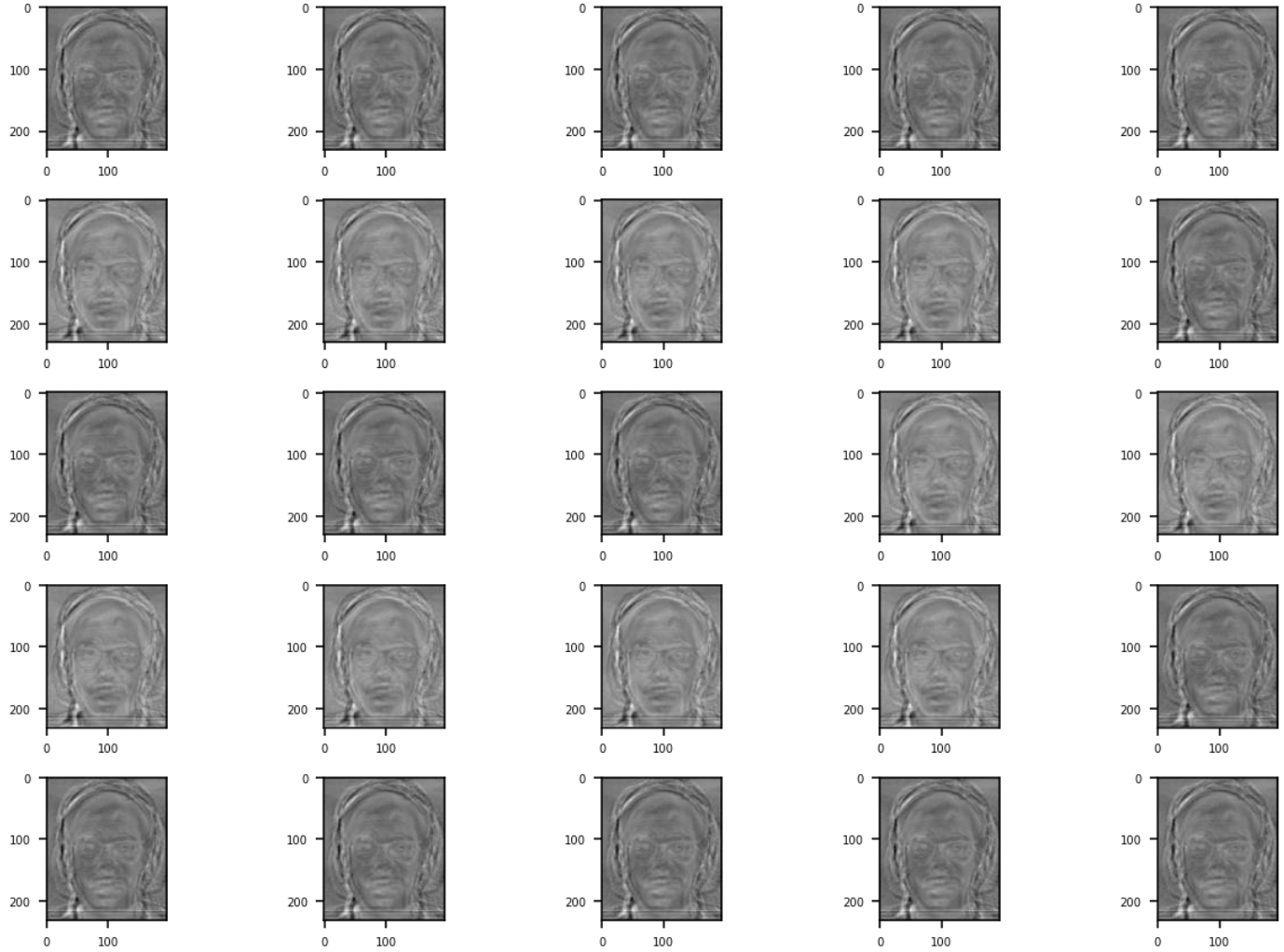


I get 86.67% accuracy by using K=3 (kNN) and setting the reduced dimension as 50, C=5.

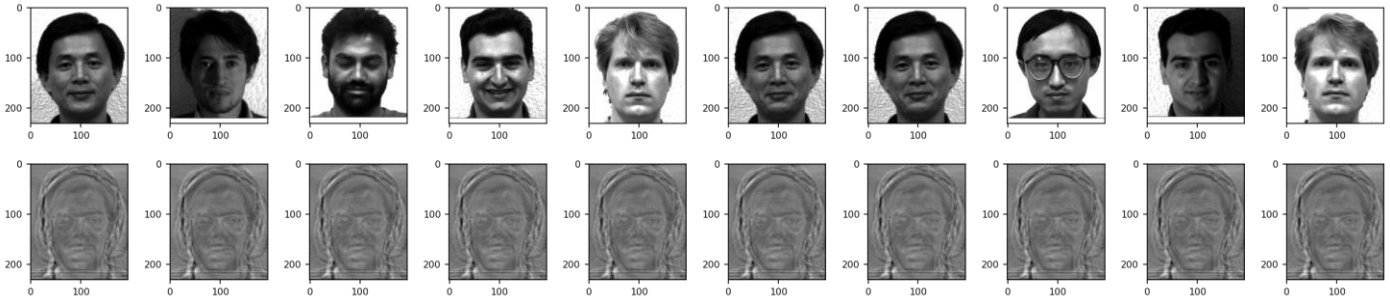
✓ Kernel LDA

➢ Polynomial case:

Here shows the fisherfaces of the columns in transform matrix U



Here is the reconstruction result:



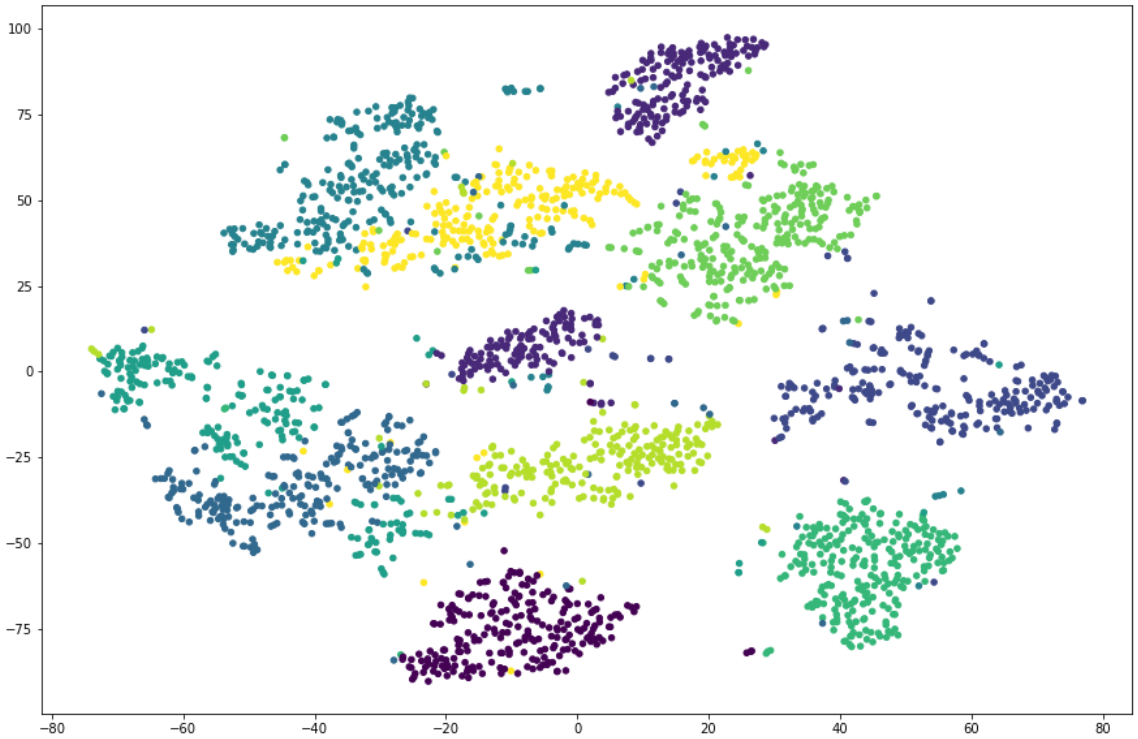
I get only 6.67% accuracy by using K=3 (kNN) and setting the reduced dimension as 50, alpha=2, d_poly=3.

- t-SNE

Part1 & Part2

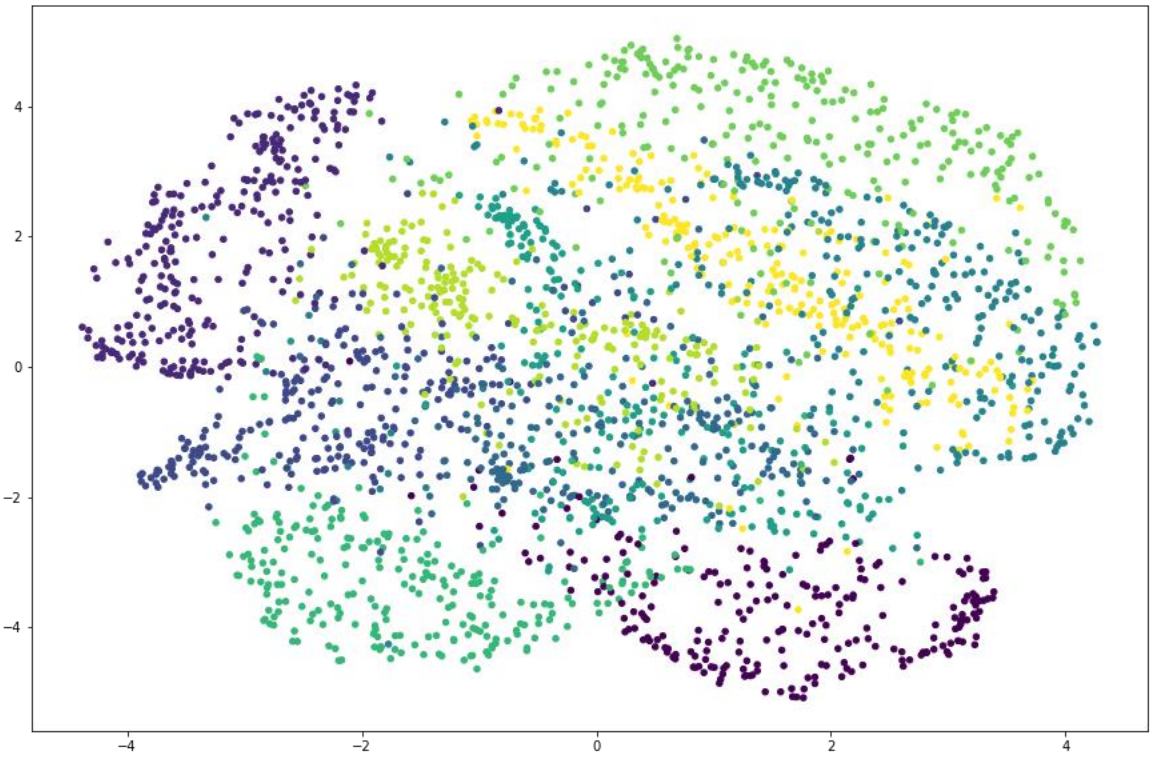
✓ t-SNE

The parameters I used are the default settings : (X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0) and 1000 iterations, the result is shown below:



✓ symmetric SNE

Same as t-SNE, I used the default settings : (X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0) and 1000 iterations, the result is shown below:

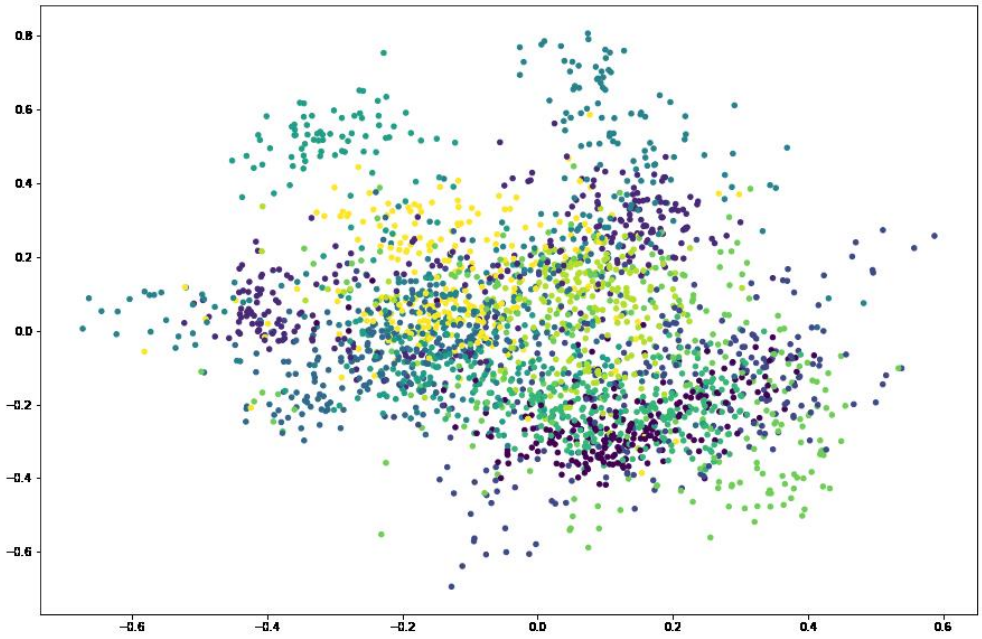


Observation: We can observe that the result of symmetric SNE suffers from “crowed problem”, and the result of t-SNE gets clear clusters after 1000 iterations.

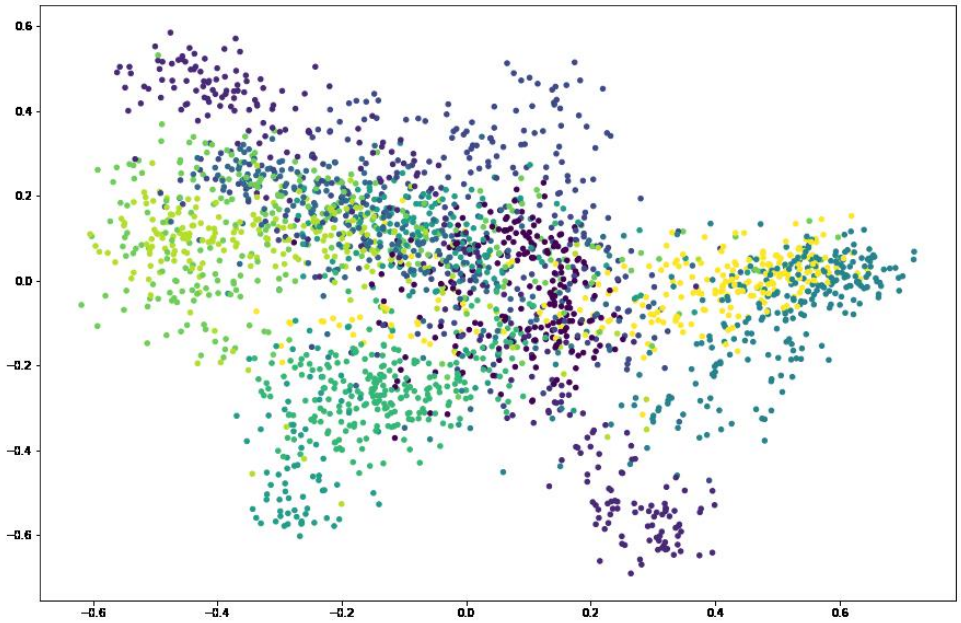
I also made gif files during clustering (1 picture/10 iterations):

The .gif files are organized in the files (the bellow gif files are frozen at the 10th iteration).

- ✓ t-SNE



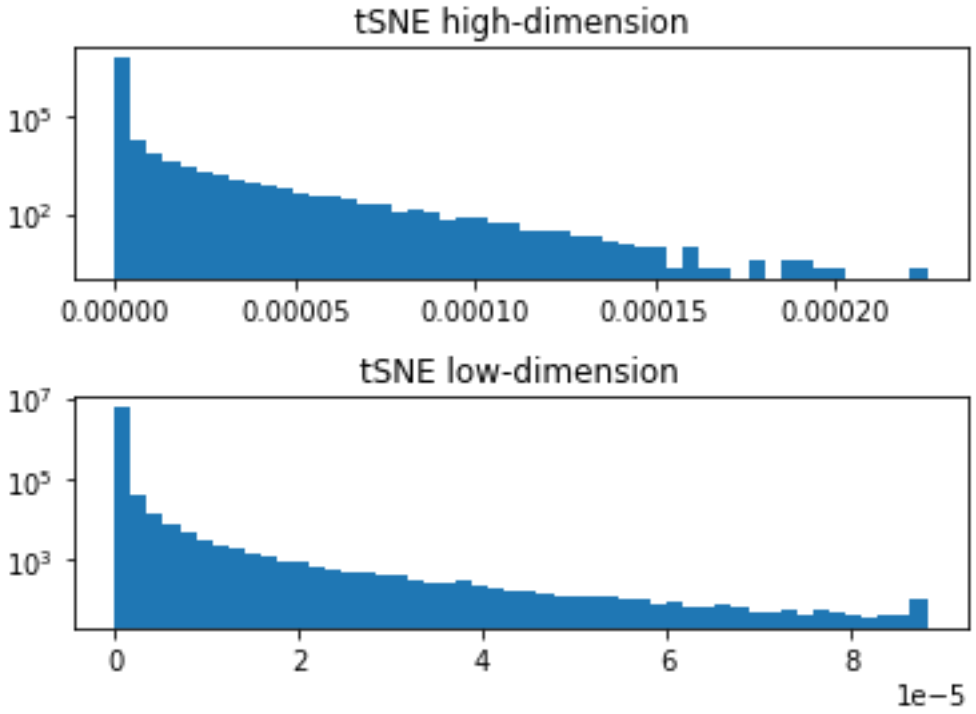
- ✓ symmetric SNE



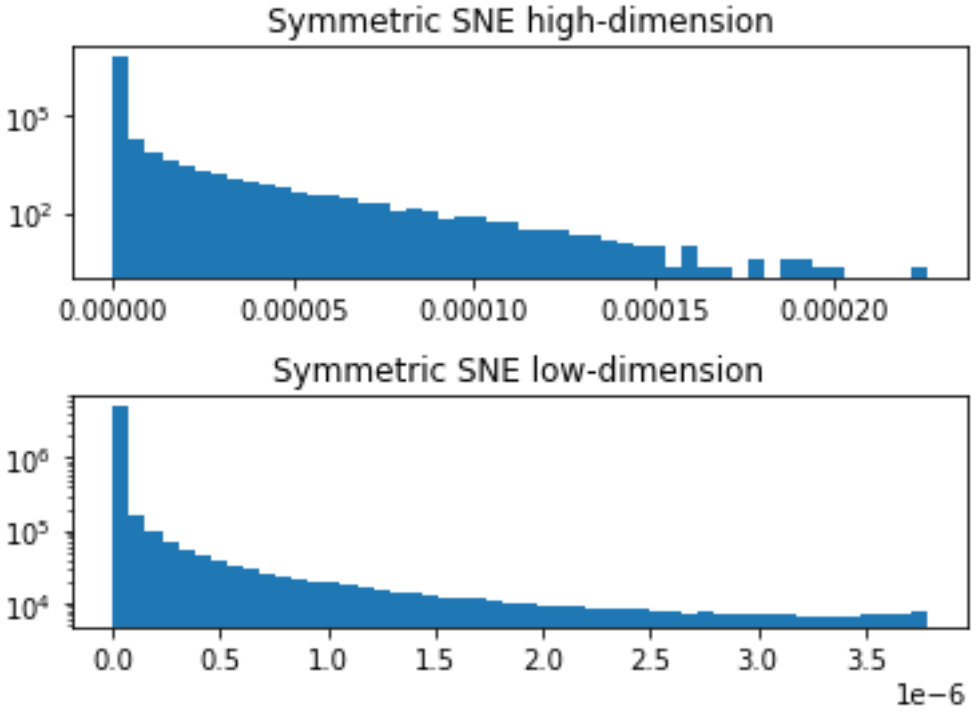
Part3: Visualize the distribution of pairwise similarities in both high-dimensional space and low-dimensional space, based on both t-SNE and symmetric SNE

Both methods are implemented with the default settings : (X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0) and 1000 iterations

✓ t-SNE



✓ symmetric SNE



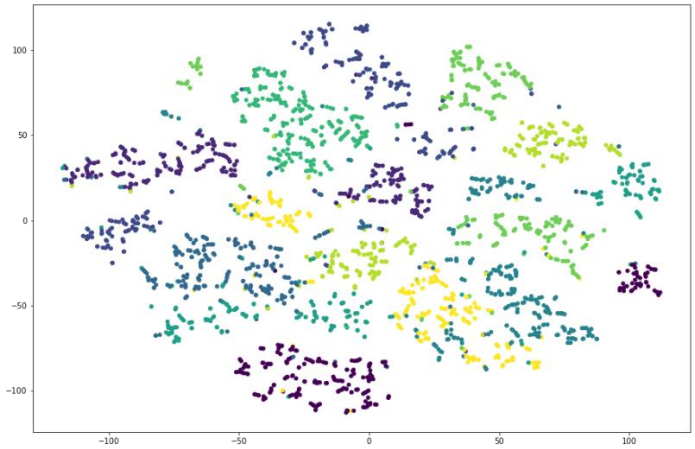
Observation: We can found that both t-SNE and symmetric SNE examples were converge after 1000 iterations. The pairwise similarities in low-d of t-SNE is in the range: 0 - $8*10^{-5}$, and of symmetric SNE: 0 – $3.5*10^{-6}$ and $3.5*10^{-6}$ is much smaller than $8*10^{-5}$. This is the reason of the crowed problem.

Part4 Try to play with different perplexity values:

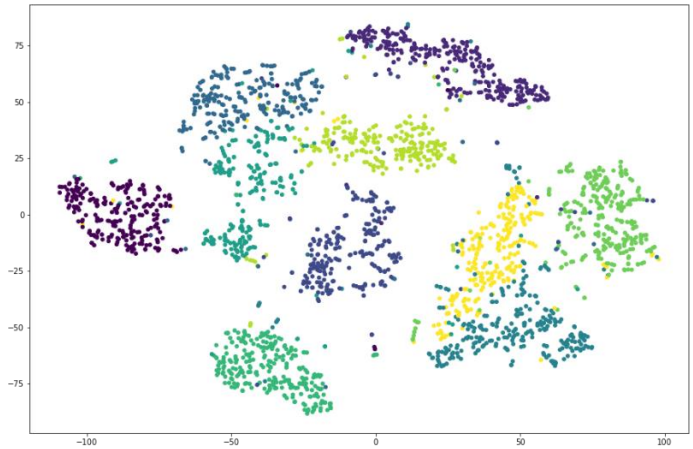
I tried perplexity: [5, 20, 30, 50] in both t-SNE and symmetric SNE

✓ t-SNE

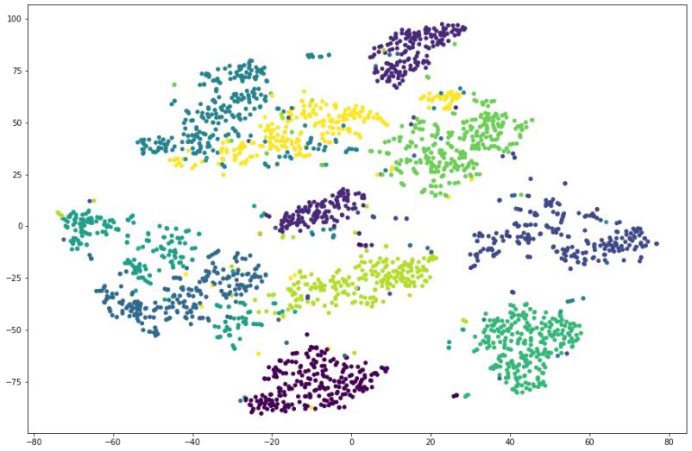
perplexity: 5



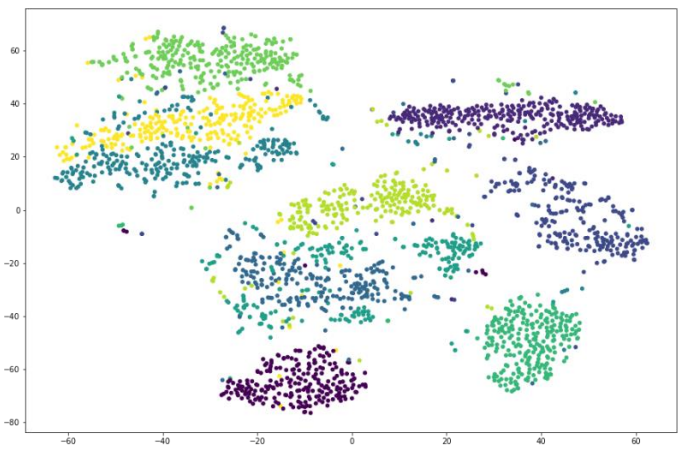
perplexity: 20



perplexity: 30

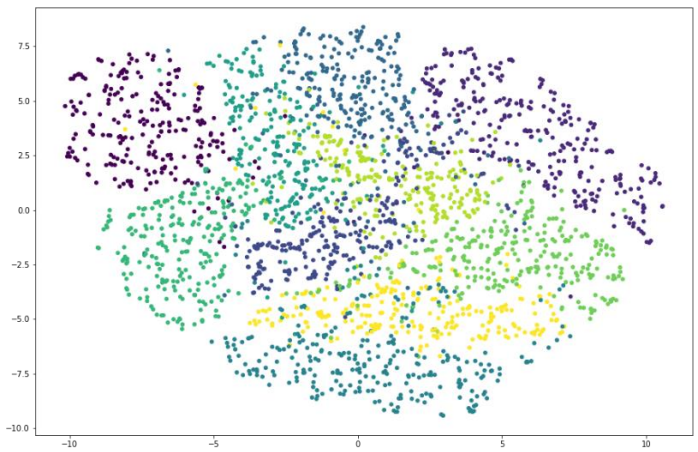


perplexity: 50

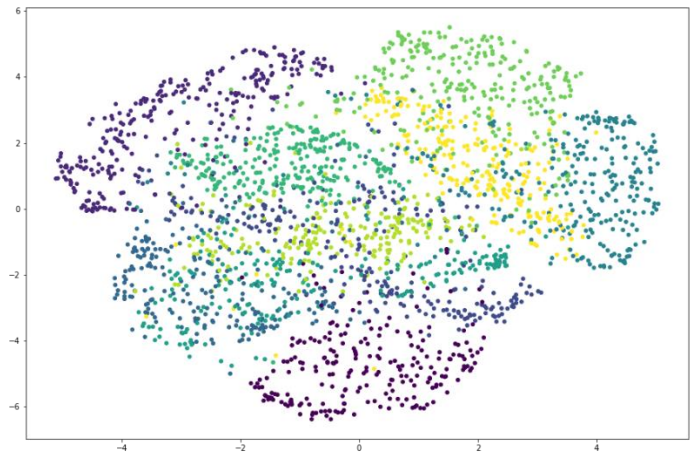


✓ symmetric SNE

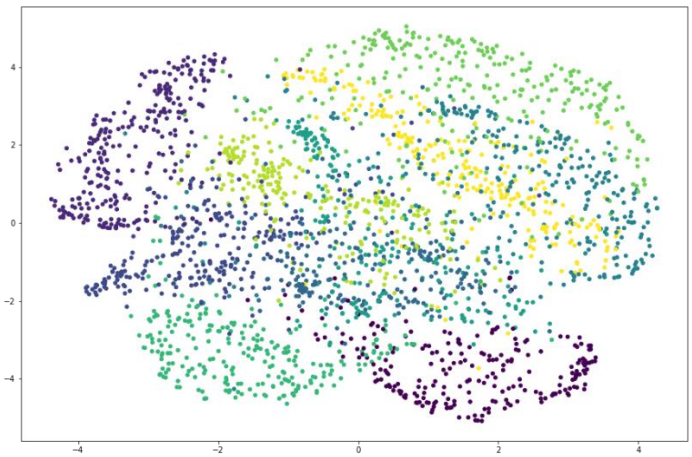
perplexity: 5



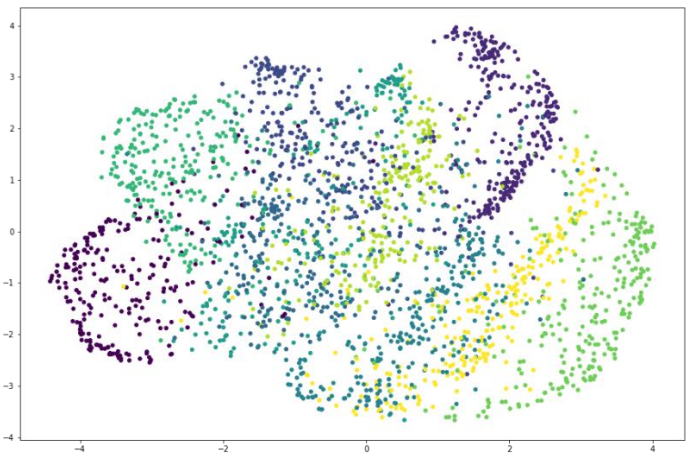
perplexity: 20



perplexity: 30



perplexity: 50



Observation: Like I described in the part a, we can observe these property on t-SNE cases very clearly: When the lower perplexity is used, only few neighbors have impact and the big cluster might become numerous small clusters. It showed better global structure of clusters but may cause the ambiguity between clusters with larger perplexity.

The symmetric SNE cases show the crowed problem in a clear way.

■ c. observations and discussion (10%)

In the Kernel Eigenfaces task, I got good performance after implementing the kernels into PCA cases, both ‘sigmoid’ and ‘linear’ kernels got >86% accuracy, slightly higher than the original PCA performance (around 83%). The “polynomial” PCA got 80% accuracy which is lower than the original PCA, if I did grid search, I’ll get better hyperparameters I think. In the Kernel LDA case, I got a really poor performance, maybe something went wrong when I implemented the kernel in to the original LDA code.
About t-SNE task, I think it is amazing that I can observe the “crowded problem” between symmetric SNE and t-SNE obviously. It’s also nice that we don’t need to handcraft the t-SNE code by ourselves.