309551101 郭育麟

ML07

A. Code with detailed explanations

(1) Kernel Eigenfaces

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

Eigenfaces : Use PCA to show the eigenfaces .

Step1: Dataloader

```
def dataloader(path,H,W):
    pictures = os.listdir(path)
    images = np.zeros((H*W,len(pictures)))
    labels = np.zeros(len(pictures))
    for i,picture in zip(np.arange(len(pictures)),pictures):
        labels[i] = int(picture.split('.')[0][7:9])-1
        image = np.asarray(Image.open(os.path.join(path,picture)).resize((W,H),Image.ANTIALIAS)).flatten()
        images[:,i] = image
    return images , labels
```

Given a path , and get every images which sizes is (H,W) and its corresponding label .

Step2 : Doing PCA

```
def pca(X,dim):
    #https://blog.csdn.net/hustqb/article/details/78394058
    #https://blog.csdn.net/zouxy09/article/details/45276053 can know by xT*x get xxT eigenvectors
    X_mean = np.mean(X,axis=1).reshape(-1,1)
    X_cor  = X-X_mean
    print("Raw data X is",X.shape)
    print("X_mean is ",X_mean.shape)
    print("X_cor is ",X_cor.shape)
    X_corvariance  = np.dot(X_cor.T,X_cor)
    print("X_corvariance is",X_corvariance.shape)
    eigenvalues , eigenvectors = np.linalg.eig(X_corvariance)
    sort_index = np.argsort(-eigenvalues)
    sort_index = sort_index[:dim]

    eigenvalues = eigenvalues[sort_index]
    eigenvectors = np.dot(X_cor,eigenvectors)
    eigenvectors = eigenvectors[:,sort_index]
    eigenvectors_normalize = np.linalg.norm(eigenvectors,axis=0)
    eigenvectors=eigenvectors/eigenvectors_normalize
    eigenfaces = eigenvectors
    print("eigenvectors is",eigenvectors.shape)  #45045*30

    pca_data = np.dot(eigenvectors.T,X_cor)
    print("pca_data",pca_data.shape)

    return eigenvalues,eigenfaces,X_mean , pca_data
```

In this step , I use the PCA method , the PCA is the method to data dimensionality reduction and it is unsupervised . First , given a initial whole data “X” , and calculate every dimension mean “X_mean” to get “X_cor” , then calculate its covariance matrix to get “X_corvariance” . We want the actual covariance matrix is “X*X.T” not “X.T*X” , but in practice , the actual covariance matrix is N*N , N represents the data dimension , it could be very large , so when wants to gets it corresponding eigenvalue and eigenvectors are very waste time , so we calculate

"X.T*X" covariance matrix eigenvalue and eigenvectors , this matrix size is m*m , m represent the number of data , it can significantly reduce the computational burden . Now I want to show the two matrix's eigenvalue "X*X.T" and "X.T*X" are same , and the eigenvectors can calculate by a matrix product .

$$C' = X^T * X$$

By the definition of eigenvalue and eigenvectors we can find the eigenvalue λ and its corresponding eigenvectors e_i s.t

$$C' * e_i = \lambda * e_i$$
$$X^T * X * e_i = \lambda * e_i$$
$$X * X^T * X * e_i = \lambda * X * e_i$$
$$\text{where } C = X * X^T$$
$$C * X * e_i = \lambda * X * e_i$$
$$\text{Now let } X * e_i = v_i$$
$$C * v_i = \lambda * v_i$$

By this proof , we can get C' and C has the same eigenvalue , and could get C eigenvectors by a matrix X * C' eigenvectors . So the "X_covariance" is "X.T*X" can help us reduce a lot of computational burden . After get its eigenvalue and eigenvectors , we want the top k eigenvalues and its corresponding eigenvectors , the reason why should choose top k eigenvalues is the covariance matrix can be diagonalize , it means that we could find eigenvalue and eigenvectors s.t $e * C * e^T = D$, e represent eigenvectors , D represents diagonal matrix and all the elements are eigenvalues . The eigenvalues can seen as variance , the more scattered the data in a certain feature dimension of the data, the more important the feature normally , so we choose top k eigenvalues and its corresponding eigenvectors to represent the new Principal Component . Finally , calculate the product of chosen eigenvectors and "X_cor" can get the processed data from PCA method .

step 3 : show eigenfaces

```
def show_eigenfaces(eigenfaces,num,H,W):
    n = int(num**0.5)
    for i in range(num):
        plt.subplot(n,n,i+1)
        plt.imshow(eigenfaces[:,i].reshape(H,W),cmap='gray')
    plt.show()
```

In this step , we want to show 25 eigenfaces , so we create 5*5 figure and use plt.imshow to plot every eigenface from the previous step's eigenvectors .

Step 4 : reconstruction

Recall that the data after PCA method are following equations

$$data_{pca} = eigenvectors_{topk}^T * (data_{origincal} - data_{mean})$$

So

$$\text{eigenvectors}_{\text{topk}} * data_{pca} = eigenvectors_{topk} * eigenvectors_{topk}^{T} * (H)$$

Because eigenvectors are orthonormal matrix

$$\text{eigenvectors}_{\text{topk}} * \text{data}_{\text{pca}} = \text{I} * (\text{data}_{\text{original}} - \text{data}_{\text{mean}})$$

$$\text{data}_{\text{original}} = eigenvectors_{topk} * data_{pca} + data_{mean}$$

After this computation , we could reconstruct the data and to visualize .

```
def reconstruct(X,X_recover,number,H,W):
    choose = np.random.choice(X.shape[1],number)
    for i in range(number):
        plt.subplot(2,number,i+1)
        plt.imshow(X[:,choose[i]].reshape(H,W),cmap='gray')
        plt.subplot(2,number,i+1+number)
        plt.imshow(X_recover[:,choose[i]].reshape(H,W),cmap='gray')
    plt.show()
```

There are two matrix , one is original matrix and the other is reconstruct matrix .

Then random choose 10 pictures to plot the original picture and reconstruct picture .

Main:

```
if __name__ == '__main__':
    filepath = "./Yale_Face_Database/Training"
    H=231
    W=195
    dim = 30
    X_train,y_train = dataloader(filepath,H,W)
    eigenvalues,eigenfaces,X_mean,pca_train = pca(X_train,dim)

    #show 25 eigenface
    show_eigenfaces(eigenfaces,25,H,W)

    #recover
    recover = np.dot(eigenfaces,pca_train)+X_mean
    print("recover shape",recover.shape)
    number = 10
    reconstruct(X_train,recover,number,H,W)
```

Fisherfaces:

Step1: Dataloader (same as previous)

Step2: Do LDA

The LDA method purpose is similar as PCA , but it is supervised learning .

I will divide LDA into several parts . The main idea in LDA is wishing the data points which belong to same class are projected the closer the better , and the data which not belong to the same class are projected the far the better .

(1) compute the data every class mean vector

(2) By class mean vector , compute Between class S_B *and within class* S_W .

S_B represent the degree of separation of different class projection data points

$$S_B = \sum_{i=1}^{K} Number_{C_k} (m_{C_k} - m)(m_{C_k} - m)^T$$

where $Number_{C_k}$ *is the number of* $k-$ *class data points*

m_{C_k} *is the* $k-$ *class mean vector*

m is the all data points mean vector

S_W represent the degree of separation of same class projection data points

$$S_W = \sum_{k=1}^{K} S_k$$

$$\text{where } S_k = \sum_{n \in C_k} (x_n - m_{C_k})(x_n - m_{C_k})^T$$

S_k *can seen as* $k-$ *class divergence matrix*

We want S_B the bigger the better and want S_W the smaller the better . So the objective function will be $J(w) = \frac{W^T S_B W}{W^T S_W W}$ w represent the projection matrix . And by Rayleigh quotient and differential with respect to w

$$\frac{\partial J(w)}{\partial w} = 0 \rightarrow S_B w(w^T S_W w) = (w^T S_B w) S_W w$$

$$S_B w = \frac{w^T S_B w}{w^T S_W w} S_W w = J(w) S_W w$$

$$S_W^{-1} S_B w = J(w) w$$

where $S_W^{-1} S_B$ *is an array* , $J(w)$ *is a scalar*

array $*$ vector $=$ sclar $*$ vecotr

can seen as the definition of eigenvalue and eigenvectors .

(3) The equation $S_W^{-1} S_B W = \lambda W$, compute the $S_W^{-1} S_B$ eigenvalue and eigenvectors

(4) Find top k eigenvalue and its corresponding eigenvectors to generate p projection matrix W

Because we want J(w) the bigger the better , so we wants $S_W^{-1} S_B$ the top k eigenvalue and its corresponding eigenvectors

(5) Y = WX

```
def lda(x,y,dim):
    #https://blog.csdn.net/kuweicai/article/details/79330524
    #https://blog.csdn.net/matrix_space/article/details/51375691
    #https://blog.csdn.net/SongGu1996/article/details/99560964 it has multiple situation
    N = x.shape[0]
    X_mean = np.mean(x,axis = 1).reshape(-1,1)

    classes_mean = np.zeros((N,15))
    for i in range(15):
        class_blong_tuple = np.where(y == i)
        class_blong=[]
        count = 0
        for j in class_blong_tuple:
            for k in j:
                k = int(k)
                class_blong.append(k)
        for j in range(len(class_blong)):
            classes_mean[:,i]  = classes_mean[:,i] + x[:,class_blong[j]]
            count = count +1
        classes_mean[:,i] = classes_mean[:,i] / count
```

```
    #S_W within-class
    S_W = np.zeros((N,N))
    for i in range(15):
        class_blong_tuple = np.where(y == i)
        class_blong=[]
        count = 0
        for j in class_blong_tuple:
            for k in j:
                k = int(k)
                class_blong.append(k)
        for j in range(len(class_blong)):
            temp = x[:,class_blong[j]].reshape(-1,1)-classes_mean[:,i].reshape(-1,1)
            S_W  = S_W + np.dot(temp,temp.T)
```

```
    #S_B between-class
    S_B = np.zeros((N,N))
    for i in range(15):
        temp = classes_mean[:,i].reshape(-1,1)-X_mean
        S_B =S_B + np.dot(temp,temp.T)
```

```
    S_W_inverse = np.linalg.inv(S_W)
    Array = np.dot(S_W_inverse,S_B)
    eigenvalues , eigenvectors = np.linalg.eig(Array)
    sort_index = np.argsort(-eigenvalues)
    sort_index = sort_index[:dim]

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

    return eigenvalues , eigenvectors
"
```

Step 3 show eigenfaces (same as PCA)

Step 4 reconstruction (same as PCA)

Main:

```python
if __name__ == '__main__':
    filepath = "./Yale_Face_Database/Training"
    H=75
    W=75
    dim = 25
    number = 10
    X_train,y_train = dataloader(filepath,H,W)
    print(X_train.shape)

    eigenvalues_lda , eigenvectors_lda , X_mean = lda(X_train,y_train,dim)
    eigenfaces_lda = eigenvectors_lda


    show_eigenfaces(eigenfaces_lda,25,H,W)

    lda_data = np.dot(eigenfaces_lda.T,X_train)

    X_recover = np.dot(eigenfaces_lda,lda_data)
    reconstruct(X_train,X_recover,number,H,W)
```

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

```python
def testing(X_test,y_test,pca_train,y_train,eigenvectors,X_mean,k):

    pca_test = np.dot(eigenvectors.T,(X_test-X_mean))
    print("pca_test shape",pca_test.shape)

    #k-nn
    test_predict = np.zeros(pca_test.shape[1])
    for i in range(pca_test.shape[1]):
        distance = np.zeros(pca_train.shape[1])
        for j in range(pca_train.shape[1]):
            distance[j] = np.sum(np.square(pca_test[:,i]-pca_train[:,j]))
        sort_index = np.argsort(distance)
        closet_neightbor = y_train[sort_index[:k]]
        unique , counts = np.unique(closet_neightbor,return_counts=True)

        nearest_neighbors=[k for k,v in sorted(dict(zip(unique, counts)).items(), key=lambda item: -item[1])]
        test_predict[i] = nearest_neighbors[0]

    accuracy = np.count_nonzero((y_test-test_predict)==0)/len(y_test)
    return accuracy
```

In this function , first compute embedding data from pca or lda by using eigenvectors.T * (data-data_mean) and use k-nn algorithm to compute the distance between training data embedding feature and testing data embedding feature to find where testing data embedding feature belongs which class and find which occurs the most times in K times . So we can finally get predict result on every testing embedding feature to calculate the accuracy on testing data .

Main:

```python
if __name__ == '__main__':
    #test and calculate accuracy
    filepath = "./Yale_Face_Database/Testing"
    k=3
    X_test , y_test = dataloader(filepath,H,W)
    accuracy = testing(X_test,y_test,pca_train,y_train,eigenfaces,X_mean,k)
    print("Accuracy is %.3f"%accuracy)
```

Part3: Use kernel PCA and kernel LDA to do face recognition, and compute the performance

In this part , we use kernel function to represent our covariance matrix before . By doing kernel trick , it can let datapoints projecting into infinite dimension , and use PCA or LDA method to classify in low dimension . The reason why use kernel function is that the kernel function could compute the similarity in high dimension between two vectors .

Eigenfaces : Following the steps

(1) Compute the similarities and centralization

$$\widetilde{\mathbf{K}} = \mathbf{K} - \mathbf{K} \cdot \mathbf{1_N} - \mathbf{1_N} \cdot \mathbf{K} + \mathbf{1_N} \cdot \mathbf{K} \cdot \mathbf{1_N}$$

```python
def kernel_trick(x1,gamma):
    print("Commputing RBF kernel")
    ###Compute RBF kernel
    K    = np.exp(-gamma*cdist(x1,x1,'sqeuclidean'))
    ###Compute Polynomial kernel
    K = np.dot(x1,x1.T)+gamma
    K = K**2
    ###Compute linear kernel
    K= np.dot(x1,x1.T)

    N    = K.shape[0]
    one_n = np.ones((N,N))/N
    K_hat = K- np.dot(one_n,K) - np.dot(K,one_n)+np.dot(np.dot(one_n,K),one_n)
    print("RBF kernel ready")
    print("RBF shape is ",K.shape)
    return K,K_hat
```

There are some kernel function to use and do the centralization .

(2) doing PCA method

```python
def kernel_pca(kernel,kernel_hat,dim):
    eigenvalues , eigenvectors = np.linalg.eig(kernel_hat)
    sort_index = np.argsort(-eigenvalues)
    sort_index = sort_index[:dim]
    eigenvectors = eigenvectors[:,sort_index]
    eigenvectors_normalize = np.linalg.norm(eigenvectors,axis=0)
    eigenvectors=eigenvectors/eigenvectors_normalize

    eigenvalues=np.asarray(eigenvalues.real,dtype='float')
    eigenvectors=np.asarray(eigenvectors.real,dtype='float')

    eigenfaces = eigenvectors
    print("kernel shape is ",kernel.shape)
    print("eigenvectors is",eigenvectors.shape)

    pca_data = np.dot(eigenvectors.T,kernel_hat)
    print("pca_data shape is ",pca_data.shape)
    return eigenvectors,pca_data
```

This steps is similar as previous

(3) Testing

```python
def kernel_testing(X_test,y_test,X_train,training_kernel,eigenvectors,k,gamma):
    test_kernel  = kernel_trick2(X_test.T,X_train.T,gamma)
    print("test_kernel shape is ",test_kernel.shape)  #30*135
    L,N   = test_kernel.shape
    one_N = np.ones((N,N))/N
    one_NL = np.ones((N,L))/N
    test_kernel_hat = test_kernel-np.dot(test_kernel,one_N)
    test_kernel_hat = test_kernel_hat-np.dot(one_NL.T,training_kernel)
    test_kernel_hat = test_kernel_hat+ np.dot(np.dot(one_NL.T,training_kernel),one_N)


    pca_test = np.dot(eigenvectors.T,test_kernel_hat.T)
    print("pca_test shape",pca_test.shape)

    #k-nn
    test_predict = np.zeros(pca_test.shape[1])
    for i in range(pca_test.shape[1]):
        distance = np.zeros(pca_train.shape[1])
        for j in range(pca_train.shape[1]):
            distance[j] = np.sum(np.square(pca_test[:,i]-pca_train[:,j]))
        sort_index = np.argsort(distance)
        closet_neightbor = y_train[sort_index[:k]]
        unique , counts = np.unique(closet_neightbor,return_counts=True)

        nearest_neighbors=[k for k,v in sorted(dict(zip(unique, counts)).items(), key=lambda item: -item[1])]
        test_predict[i] = nearest_neighbors[0]
    accuracy = np.count_nonzero((y_test-test_predict)==0)/len(y_test)
    return accuracy
```

```python
def kernel_trick2(x1,x2,gamma):
    ###Compute RBF kernel
    K    = np.exp(-gamma*cdist(x1,x2,'sqeuclidean'))
    ###Compute Polynomial kernel
    K = np.dot(x1,x2.T)+gamma
    K = K**2
    ###Compute linear kernel
    K= np.dot(x1,x2.T)
    return K
```

$$\widetilde{\mathbf{K}}_{test} = \widetilde{\phi}(\mathbf{X_{test}})^T \widetilde{\phi}(\mathbf{X})$$
$$= [\phi(\mathbf{X_{test}}) - \phi(\mathbf{X})\,\mathbf{1_{NL}}]^T [\phi(\mathbf{X}) - \phi(\mathbf{X})\,\mathbf{1_N}]$$
$$= \phi(\mathbf{X_{test}})^T \phi(\mathbf{X}) - \phi(\mathbf{X_{test}})^T \phi(\mathbf{X})\,\mathbf{1_N} - \mathbf{1_{NL}}^T \phi(\mathbf{X})^T \phi(\mathbf{X}) + \mathbf{1_{NL}}^T \phi(\mathbf{X})^T \phi(\mathbf{X})\,\mathbf{1_N}$$
$$= \mathbf{K_{test}} - \mathbf{K_{test}} \cdot \mathbf{1_N} - \mathbf{1_{NL}}^T \cdot \mathbf{K} + \mathbf{1_{NL}}^T \cdot \mathbf{K} \cdot \mathbf{1_N}$$

Where $K_{\text{test}}: L * N$, $1_N: N * N$, $1_{NL}: N * L$

First compute the similarity between training data and testing data by using kernel function , and also do centralization . Finally do KNN algorithm as previous steps to get the predict result .

Main:

```
if __name__ == '__main__':
    #https://blog.csdn.net/ChenVast/article/details/79236160
    #https://blog.csdn.net/pantingd/article/details/107300037
    #https://zhuanlan.zhihu.com/p/59775730
    filepath = "./Yale_Face_Database/Training"
    H=231
    W=195
    dim = 20
    gamma = 7e-10
    X_train,y_train = dataloader(filepath,H,W)
    kernel,kernel_hat = kernel_trick(X_train.T,gamma)
    eigenvectors,pca_train = kernel_pca(kernel,kernel_hat,dim)


    #test and calculate accuracy
    filepath = "./Yale_Face_Database/Testing"
    k=5
    X_test , y_test = dataloader(filepath,H,W)
    accuracy = kernel_testing(X_test,y_test,X_train,kernel,eigenvectors,k,gamma)
    print("Accuracy is %.3f"%accuracy)
```

Kernel LDA:

The main difference between LDA and LDA is that SB(between class) and SW(within class) matrix are different . Kernel LDA use kernel trick to represent SB and SW matrix . Others idea are same as LDA . I use RBF kernel , linear kernel , Polynomial kernel to compare .

```python
#S_B between-class
S_B = np.zeros((N,N))
for i in range(15):
    temp = kernel_trick2(classes_mean[:,i].reshape(-1,1),X_mean,gamma)
    S_B =S_B + temp * count_class[i]


#S_W within-class
S_W = np.zeros((N,N))
for i in range(15):
    class_blong_tuple = np.where(y == i)
    class_blong=[]
    count = 0
    for j in class_blong_tuple:
        for k in j:
            k = int(k)
            class_blong.append(k)
    for j in range(len(class_blong)):
        temp = kernel_trick2(x[:,class_blong[j]].reshape(-1,1),classes_mean[:,i].reshape(-1,1),gamma)
        S_W  = S_W + np.dot(temp,temp.T)
```

(2) t-SNE

Part1: Try to modify the code a little bit and make it back to symmetric SNE

First I want to talk about SNE , different from PCA , SNE is a non-linear data dimensionality reduction method , it could images data from high dimension to low dimension efficiently , and keep data local structure in high dimension , t-SNE and symmetric SNE are extend of the SNE .

SNE:

$$p_{j|i} = \frac{\exp(-\|x_i - x_j\|^2/(2\sigma_i^2))}{\sum_{k\neq i}\exp(-\|x_i - x_k\|^2/(2\sigma_i^2))}$$

$P_{j|i}$: $measure\ data_j be\ data_i\ neighbor's\ probability\ in\ high\ dimension$

$$q_{j|i} = \frac{\exp(-\|y_i - y_j\|^2)}{\sum_{k\neq i}\exp(-\|y_i - y_k\|^2)}$$

$$C = \sum_i KL(P_i\|Q_i) = \sum_i\sum_j p_{j|i}\log\frac{p_{j|i}}{q_{j|i}}$$

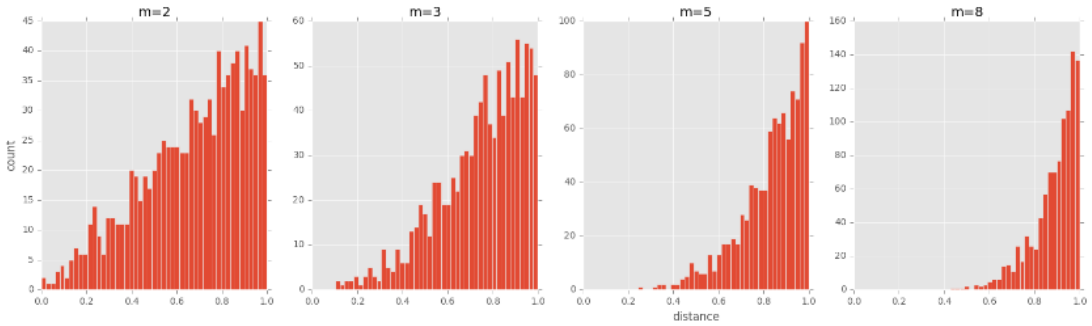
For setting variance σ_i , define Perplexity

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

$$H(P_i) = -\sum_i p_{j|i}\log_2 p_{j|i}$$

Symmetric SNE: use joint probability to replace conditional probability , it could simply the gradient formula . But it will suffer “crowding problem” .

“Crowding problem “ is indistinguishable when data gather together in low dimension . For example , when high dimension embedding into low dimension (under 10) , it has nice performance . However in 2 dimension , it could not get a credible embedding result . In the other words , suppose an m-dimension sphere with a data point as the center and a radius of r (3-dimension) , its volume is increasing , assuming that the data points are uniformly distributed in the m-dimensional sphere , let’s look at other data points and the change of the distance as the dimension increases .



It can be seen from the figure above , as the dimensionality increases , most of the data points are clustered near the surface of the m-dimensional sphere , and the distance distribution from the points is extremely uneven . if this distance relationship is directly retained to a low dimension , there will be a “crowding problem”

P_i: High dimension X joint probability

Q_i: *Low dimension Y joint probability*

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

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

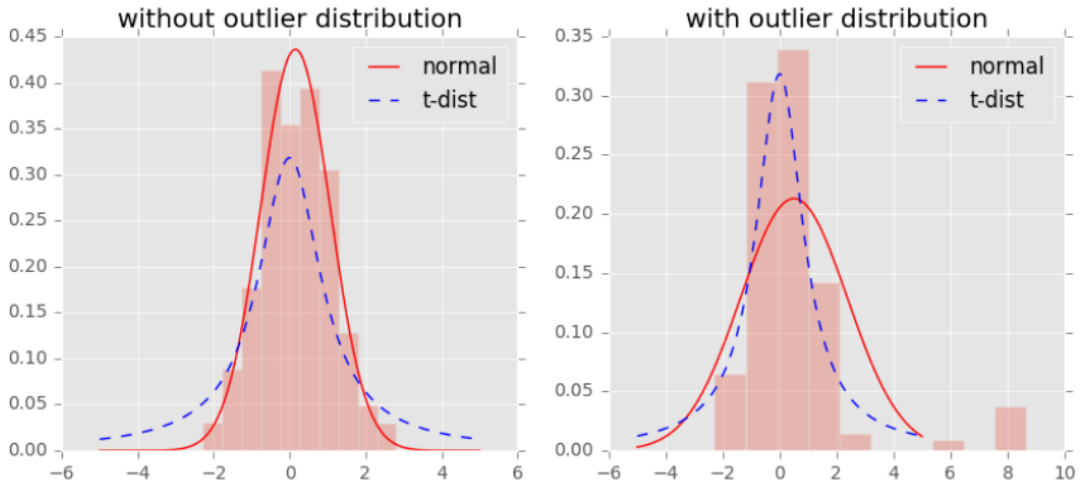
$$C = KL(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

Objective function C

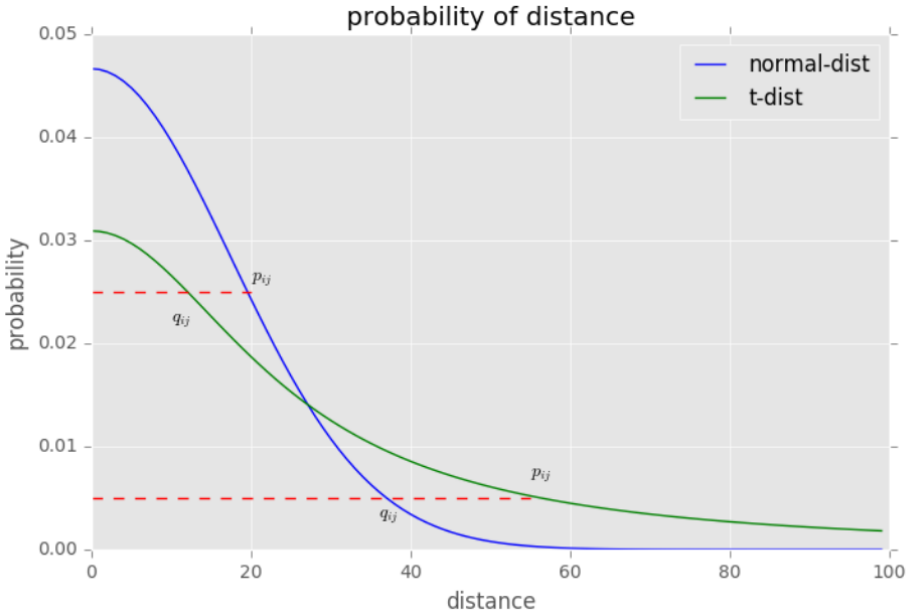
$p_{i|i} = q_{i|i} = 0 , p_{ij} = p_{ji}, q_{ij} = q_{ji}$

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

t-SNE use symmetric SNE to simply the gradient formula and use student-distribution to represent the similarity between any two datapoints , it can avoid crowding problem . Use student-distribution can more emphasis long tail distribution to transform distance into probability distribution , it can let small distance in high dimension could has a farther distance after images in low dimension .



This figure shows that student-distribution is less affected by outliers and more reasonable fitting result .



This figure shows that for high similarity data points , the distance will be little nearest , and for low similarity data points , the distance will be little farther than normal-distance . It means that the same class data points closer together and

different class data points could be far away .

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

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

Use student-distribution to represent distance probability in low dimension
So the difference between symmetric SNE and -SNE are data distance probability representation in low dimension and the final gradient result .

```
# 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

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

Symmetric SNE

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

```
pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
```

Use pylab.scatter to plot the result

Part3: Visualize the distribution of pairwise similarities in both high-dimensional space and low-dimensional space

```
def plot(P,Q):
    plt.clf()
    pylab.subplot(2,1,1)
    pylab.title('Symmetric_sne high-dim')
    pylab.hist(P.flatten(),bins=40,log=True)
    pylab.subplot(2,1,2)
    pylab.title("Symmetric_sne low-dim")
    pylab.hist(Q.flatten(),bins=40,log=True)
    plt.show()
```

P represent high dimension datapoints distance probability , Q represent low dimension datapoints distance probability .

Part4: Try to play with different perplexity values

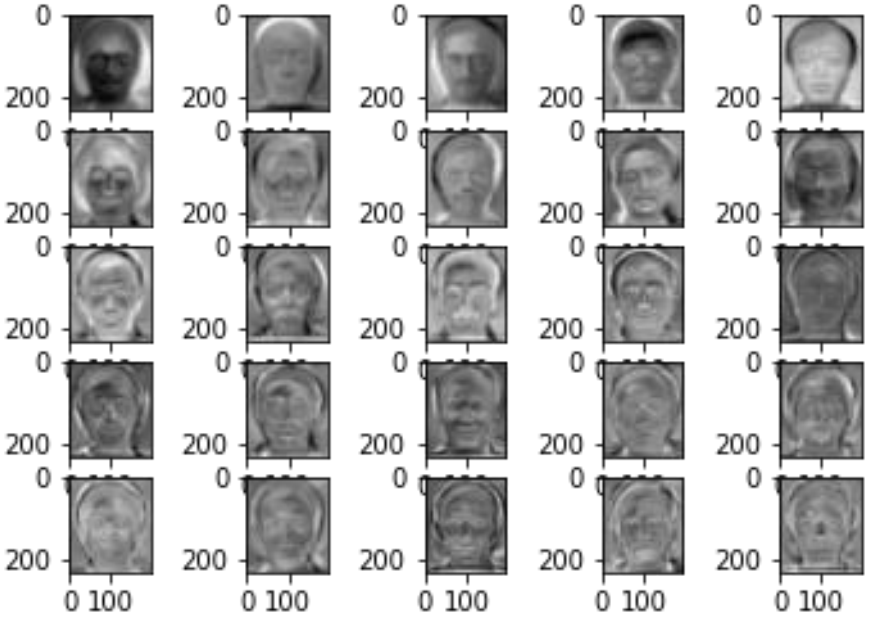
Perplexity can seen as how many neighboring points are considered in the optimization process .

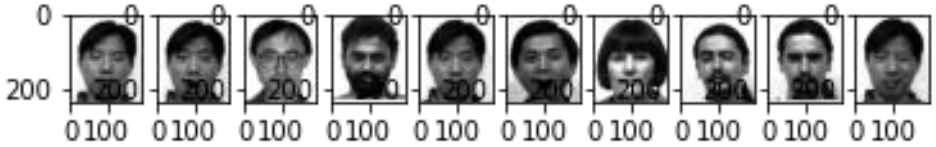
B. Experiments settings and results and discussions

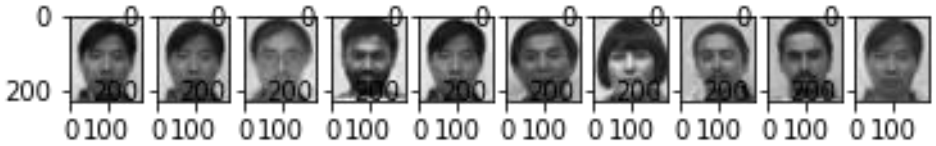
(1) Kernel Eigenfaces

Part1:

Eigenfaces and reconstruction



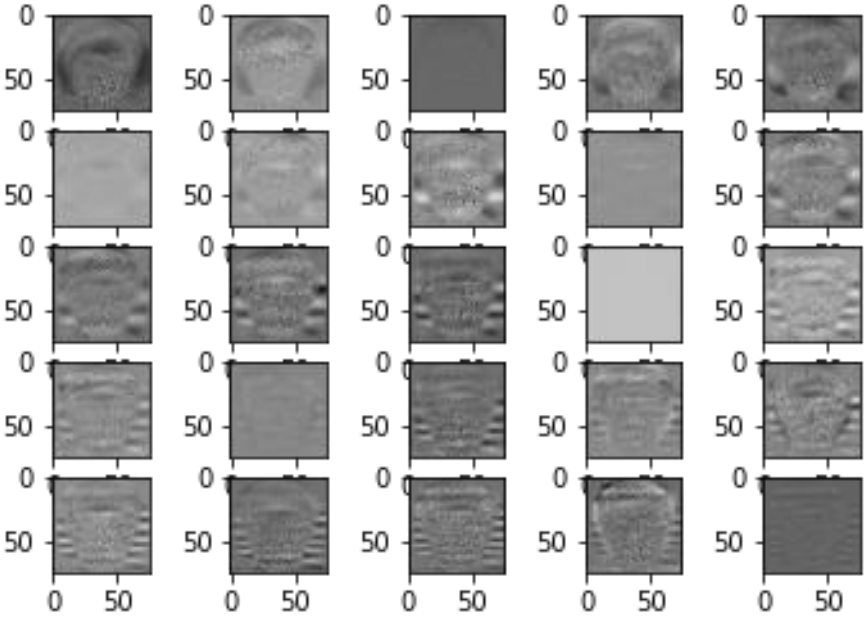


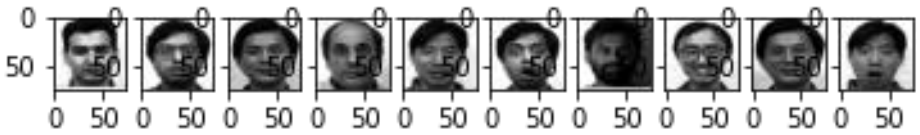


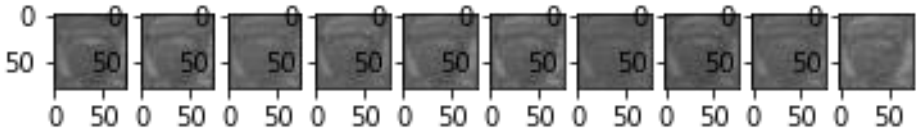
The figure in the upper are original images , and pictures in the bottom are reconstruction pictures . It can conclude that the reconstruction are very well , it can

directly see the face in the pictures .

Fisherfaces and reconstruction







In fisherfaces , i think that there are some problem in my code , because the reconstruction pictures can not directly see a human face , it just a contour .

Part2:

Eigenfaces

```
pca_test shape (30, 30)
Accuracy is 0.833
```

Fisherfaces:

```
pca_test shape (25, 30)
Accuracy is 0.867
```

Although the fisherfaces's reconstruction are not well . However , it could get higher performance in this dataset .

Part3:

Eigenface:

Linear kernel function

$$\mathrm{k}(\mathrm{x},\mathrm{y}) = x^T y + c$$

Polynomial kernel function

$$\mathrm{k}(\mathrm{x},\mathrm{y}) = (x^T y + c)^d$$

RBF kernel function

$$\mathrm{K}(\mathrm{x},\mathrm{x}') = \exp(-\gamma\left|\left|\mathrm{x}-\mathrm{x}'\right|\right|_2^2)$$

	Linear kernel	Polynomial	RBF
Parameters	None	C = 7e-10	Gamma =7e-10
Accuracy	83.3%	80%	80%

First , I thought use kernel in PCA will improve accuracy . However , my result could not get higher performance in this dataset . Maybe in other datasets , it could get a higher performance .

Fisherfaces

	Linear kernel	Polynomial	RBF
Parameters	None	C = 7e-10	Gamma =7e-10
Accuracy	23.3%	83.3%	83.3%

In this dataset , Linear kernel get a worst performance , it only has 23.3% accuracy .

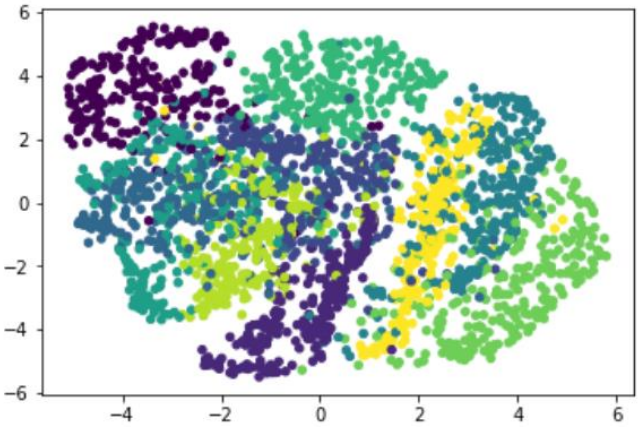
(2) t-SNE

Part1

Do not need to plot anything

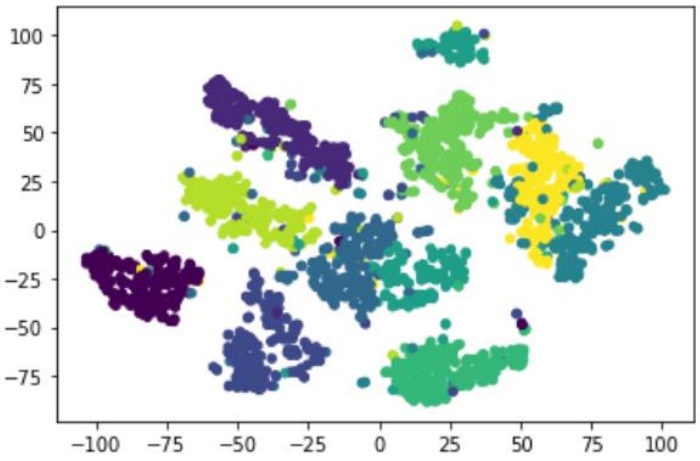
Part2

Symmetric-SNE final result



Perplexity=20

t-SNE final result

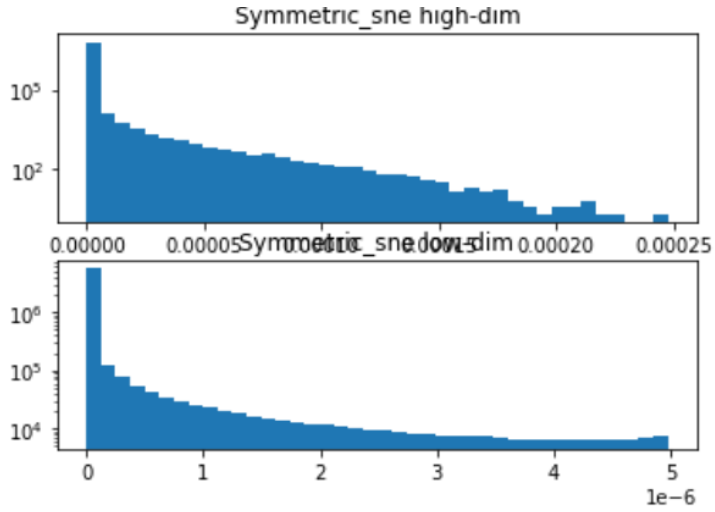


Perplexity = 20

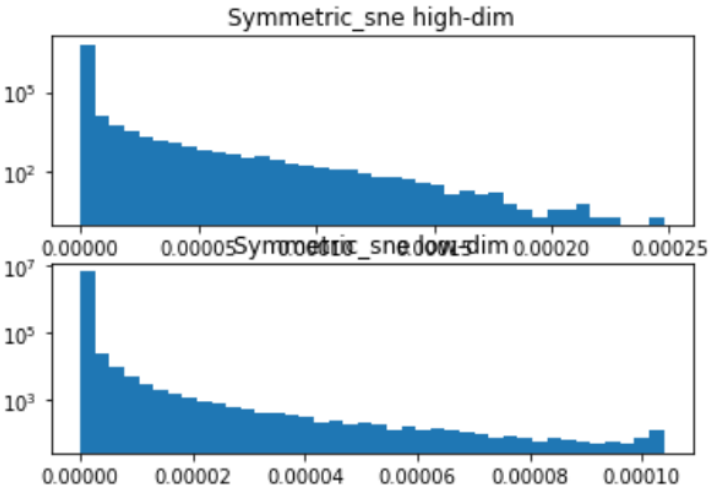
It can directly see that there has “crowding problem” in symmetric problem , the datapoints are get together , it shows that the datapoints are very crowded . However , the datapoints are separable in t-SNE , it shows that t-SNE can avoid “crowding problem “

Part3

Symmetric-SNE result



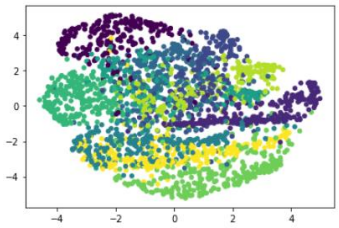
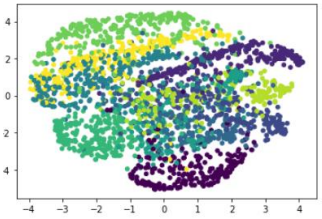
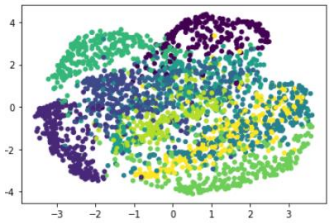
t-SNE result



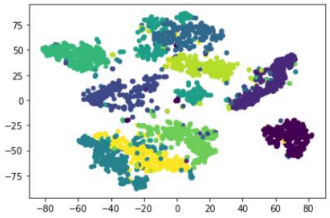
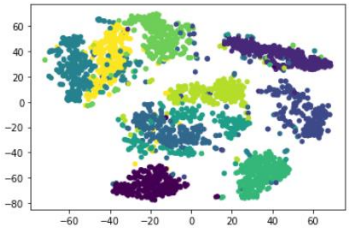
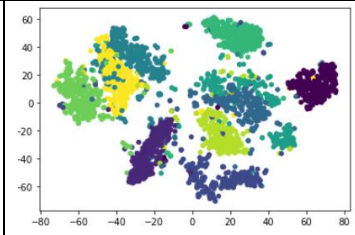
It shows that t-SNE can keep initial data structure in the low dimension . However , symmetric SNE can not keep initial data structure in low dimension , it can observed it has “crowding problem “ by this graph too .

Part4:

Symmetric SNE

Perplexity =30	Perplexity = 40	Perplexity = 50
		

t-SNE

Perplexity =30	Perplexity = 40	Perplexity = 50
		

It can see that when perplexity = 40 in Symmetric SNE , it has the better performance . However , it still occur “crowding problem” , it can not beat the t-SNE. And I think whether perplexity is , It still has the nice result in t-SNE .