

For TA's convenience , you can check the link here to see the animated gif online directly on report and have the outline to link quickly:

<https://docs.google.com/document/d/10YWXukt2DiIXozCEotiBqUptYRtdwtxfEualvNoYhhA/edit?usp=sharing>

## code

### # Kernel Eigenfaces

## part1

### PCA

```
def PCA(data,dim=None,is_kernel=False,gamma=15,mode=0):
    N = data.shape[0]
    # kernel PCA
    if is_kernel:
        cov = kernel(data,data,gamma=gamma,is_center=True,mode=mode)
    # PCA
    else:
        mean = np.mean(data,axis=0)
        # center the data
        data = data - mean
        cov = np.cov(data.T)
    # solve the eigenvalue problem of covariance matrix
    eigValues , eigVectors = np.linalg.eig(cov)
    index = np.argsort(eigValues)[::-1]
    if dim != None:
        index = index[:dim]
    W = eigVectors[:,index].real.astype(np.float64)
    return W
```

Data is a N by d matrix containing the flatten images. N is the number of training images , d is the total number of pixels of one image.

First I calculate the mean of every pixel and let data minus it to center the data. Then use the centered data to calculate the covariance matrix and solve the eigenvalue problem of this matrix by numpy.

In the end I sort the eigenvalue from big to small and according to parameter “dim” to determine the dimension of low dimensional space

and to select how many eigenvectors should be used to construct the orthogonal projection matrix  $W$ .

### ### LDA

```
def LDA(data,dim=None,stroke=9,class_num=15,is_kernel=False,gamma=15,mode=0):
    N , d = data.shape
    mean = np.mean(data,axis=0)
    Sw = np.zeros((d,d))
    Sb = np.zeros((d,d))

    if is_kernel:
        K = kernel(data,data,gamma,True,mode=mode)
        Z = np.zeros_like(K)
        diag_block = np.ones((stroke,stroke)) / stroke
        for i in range(class_num):
            Z[i*stroke:(i+1)*stroke,i*stroke:(i+1)*stroke] = diag_block
        Sw = K@K
        Sb = K@Z@K
    else:
        # get the mean of every image subjects
        classes_mean = np.zeros((class_num,d))
        for i in range(class_num):
            begin = i * stroke
            end = begin + stroke
            classes_mean[i,:] = np.mean(data[begin:end,:],axis=0)

        # the distance within class scatter
        for i in range(class_num):
            for j in range(stroke):
                d = data[i*stroke+j,:] - classes_mean[i,:]
                Sw += d.T @ d

        # the distance between class scatter
        for i in range(class_num):
            d = classes_mean[i,:] - mean
            Sb += stroke * (d.T @ d)

    # solve the eigenvalue problem of covariance matrix
    eigValues , eigVectors = np.linalg.eig(np.linalg.pinv(Sw)@Sb)
    index = np.argsort(eigValues)[::-1]
    if dim != None:
        index = index[:dim]
    W = eigVectors[:,index].real.astype(np.float64)
    return W
```

There are four steps of LDA. Because LDA is the supervised method and the image dataset is sorted by label , I can access the same class image sequentially by knowing the number of one class (parameter

“stride”) and how many classes the dataset contains (parameter “class\_num”). According to spec , the stride of the training dataset is 9 and the number of classes are 15.

1. Calculating the mean of every class.
2. Calculating the distance of within class scatter. For every image calculates the square of the image minus the mean of the class of this image and sums up those values.

$$S_W = \sum_{i=1}^c \sum_{x_j \in X_i} (x_j - \mu_i)(x_j - \mu_i)^T$$

3. Calculating the distance of between class scatter. For every class calculates the square of the mean of class minus the mean of whole training images and sums up those values.

$$S_B = \sum_{i=1}^c N_i (\mu_i - \mu)(\mu_i - \mu)^T$$

4. Solving the eigen value problem of below equation to get the eigenvalue and eigenvector.

$$\begin{aligned} S_B v_i &= \lambda_i S_W v_i \\ S_W^{-1} S_B v_i &= \lambda_i v_i \end{aligned}$$

In the end , as same as PCA , sorting the eigenvalue from big to small and according to parameter “dim” to construct the orthogonal projection matrix W.

When I implement the LDA there is some problem of inverse. the matrix Sw may be singular so I use the pseudo inverse to get the inverse and let LDA work properly.

## ## part2

```
def reconstruct(W,mean,img_data):
    y = W.T @ (img_data - mean)
    x = W @ y + mean
    # x = normalize(x) * 255
    img = x.reshape(img_h,img_w)
    return img
```

For image reconstruction , I first use W to convert the testing image(img\_data) to the low dimensional space(the image after

converting is named as y) and invert the formula to reconstruct it back to the original data space.

```
def knn(img_data, imgs, label, k):
    N = len(label)
    dist = np.zeros((N))
    for i in range(N):
        dist[i] = np.linalg.norm(img_data - imgs[i])
    index = np.argsort(dist)
    result = label[index[:k]]
    result_l, result_c = np.unique(result, return_counts=True)
    if len(result_l) != 1 and np.max(result_c) == np.min(result_c):
        ret = label[np.argmin(dist[index[:k]])]
    else:
        ret = result_l[np.argmax(result_c)]
    print(np.unique(result, return_counts=True), ret)
    return ret
```

And then I use KNN according to the L2 norm to compare with the training dataset to find the K most similar training images and base on the label of those images to vote the predictive label of the testing image.

### ## part3

The code of kernel version PCA and LDA is as same as origin PCA and LDA in the above. The different part is the kernel version will run the code conditioned by the “is\_kernel” flag.

```
# mode : 0 : RBF , 1 : linear kernel(cosine similarity)
def kernel(X, Y, gamma = 1, is_center = True, mode=0):
    if mode == 0:
        dist = cdist(X, Y, 'sqeuclidean')
        K = np.exp(-gamma * dist)
    elif mode == 1:
        K = gamma * (1 - cdist(X, Y, 'cosine'))
    # Center the kernel matrix.
    if is_center:
        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)
    return K
```

Here is the kernel function. I use two kinds of kernel. RBF & linear kernel(comparing with the cosine similarity. The “cdist” function

calculates the distance so I need to minused by 1 make it back to cosine similarity).

To match the requirement of centered data , I center the kernel matrix by the formula derived in the course material.

$$K^C = K - \mathbf{1}_N K - K \mathbf{1}_N + \mathbf{1}_N K \mathbf{1}_N$$

### ### Kernel PCA

```
# kernel PCA
if is_kernel:
    cov = kernel(data,data,gamma=gamma,is_center=True,mode=mode)
```

The different of kernel PCA is that originally PCA solves the eigenvalue problem of covariance matrix of centered data , but kernel PCA solves the eigenvalue problem of centered kernel matrix.

### ### Kernel LDA

```
if is_kernel:
    K = kernel(data,data,gamma,True,mode=mode)
    Z = np.zeros_like(K)
    diag_block = np.ones((stride,stride)) / stride
    for i in range(class_num):
        Z[i*stride:(i+1)*stride,i*stride:(i+1)*stride] = diag_block
    Sw = K@K
    Sb = K@Z@K
```

I implement the kernel LDA based on the reference paper in spec. Following this paper I prepare two matrix K and Z , K is the traditional kernel matrix (explaining the similarity between dataset). Z is a block diagonal matrix which can be divided into 15 by 15 blocks(according to the number of classes) and only has the value on the diagonal block. The value of diagonal blocks are all equal to 1 / 9(the sample number of a class.)

And then the eigenvalue problem is converted to this

$$\lambda K K \alpha = K Z K \alpha$$

. To align my code I use the same variable name Sw and Sb to represent this formula and I can directly use the same code of LDA to get the result of kernel LDA.

## # t-SNE

### ## part1

```
# Compute pairwise affinities
# sum_Y is some of square of every dimension of every data point in low dimensional space(N*1)
sum_Y = np.sum(np.square(Y), 1)
num = -2. * np.dot(Y, Y.T)

if is_symmetric:
    num = np.exp(-np.add(np.add(num, sum_Y).T, sum_Y))
else:
    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
if is_symmetric:
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
else:
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

The main difference between t-SNE and symmetric SNE is the distribution function used in low dimensional space.

t-SNE is student-t distribution.

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

Symmetric SNE is Gaussian distribution.

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

Because of using different distribution, the gradient is also different. The gradient of Symmetric SNE is in the picture below.

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

In the code after “if is\_symmetric” is the implementation of Symmetric SNE. As I mentioned in previous , I change the way to calculate the probability in low dimensional space and the gradient.

## ## part2

```
file_path = os.path.join("Image", "GIF_Image", "img{}.png".format(len(self.imgs)))
plt.figure()
plt.title(title)
plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
plt.savefig(file_path)
plt.close()
img = plt.imread(file_path)
self.imgs.append(img)
```

```
def save(self, filepath):
    imageio.mimsave(filepath, self.imgs, fps=self.fps)
```

I plot the 2D result data by matplotlib every certain iteration(5 or 10 ) and save the plot as the image to make those images to a gif image by python module imageio.

## ## part3

```
# visualize the pairwise similarity
def plot_similarity(P, Q, title):
    fig, ax = plt.subplots(nrows=2, ncols=1, figsize=(10, 8), dpi=120)
    plt.suptitle(title)
    ax[0].set_title("similarities in high dimensional space")
    ax[0].hist(P.flatten(), bins=100, log=True)
    ax[1].set_title("similarities in low dimensional space")
    ax[1].hist(Q.flatten(), bins=100, log=True)
```

After SNE algorithm I can obtain the pairwise probability in both low dimensional(matrix Q) space and high dimensional space(matrix P).

And these pairwise probability matrices also means the similarity between two data. So to see the distribution of pairwise similarity , I use matplotlib “hist” function to make statistics and draw the statistics result as 100 bins histogram.

## part4

```
def tsne(X=np.array([]), labels = np.array([]), no_dims=2, initial_dims=50, perplexity=30.0, is_symmetric=False,
```

The reference code has the parameter to control the perplexity. I simply use a different number of this parameter for experiment.

## Experiment

### # Kernel Eigenfaces

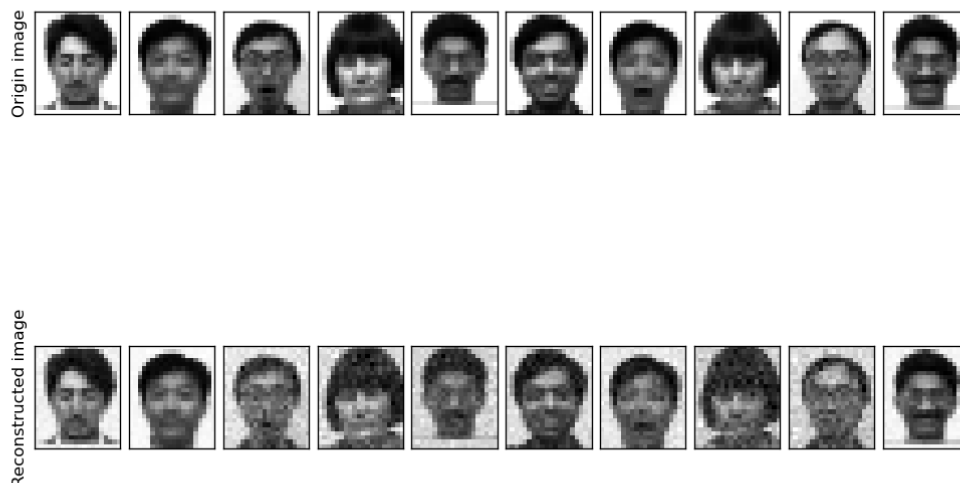
In this part I shrink the image size to  $24 * 20$  to speed up the running time. And in this resolution although the image is blurry but I think it still can distinguish the image.

## part1

### PCA

Reconstruction of 10 random image of testing image

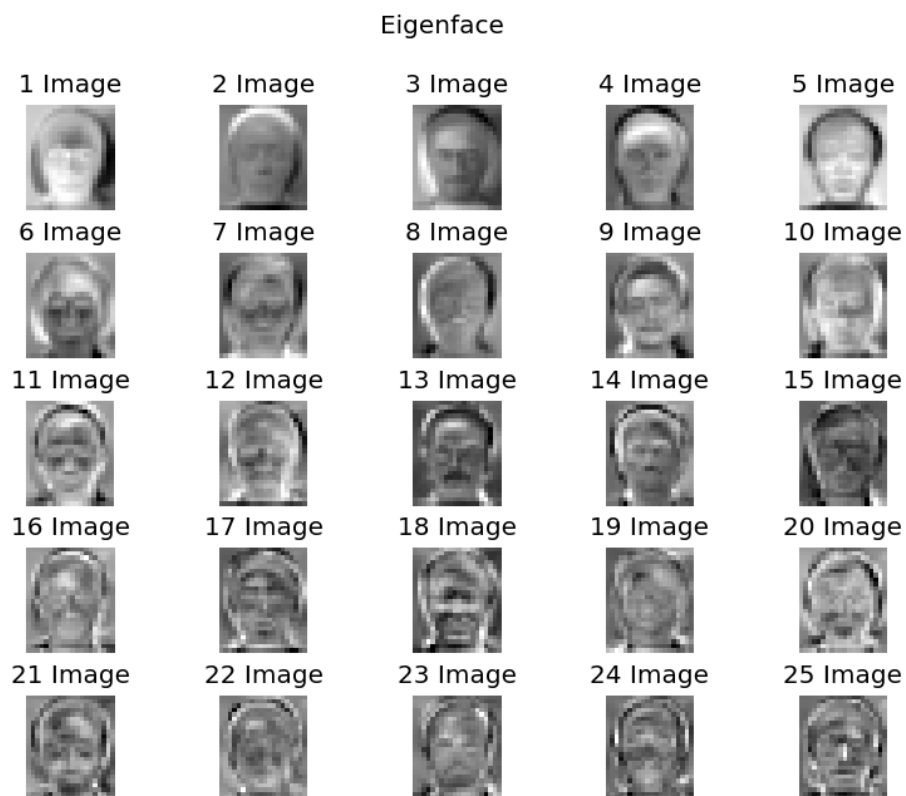
Face reconstruction



Although there are some reconstructing results have some noise like the Gaussian noise but except the noise the result is very close to original image. Including the hair style, expression and so on.



## First 25 eigenface

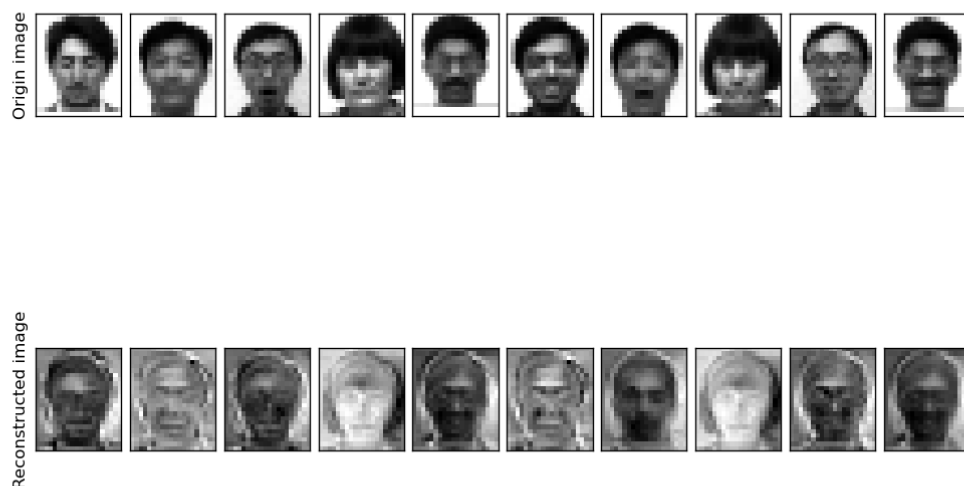


The results look like the person face but according to the white part of image I think the different eigenface may emphasize on different part of the face feature.

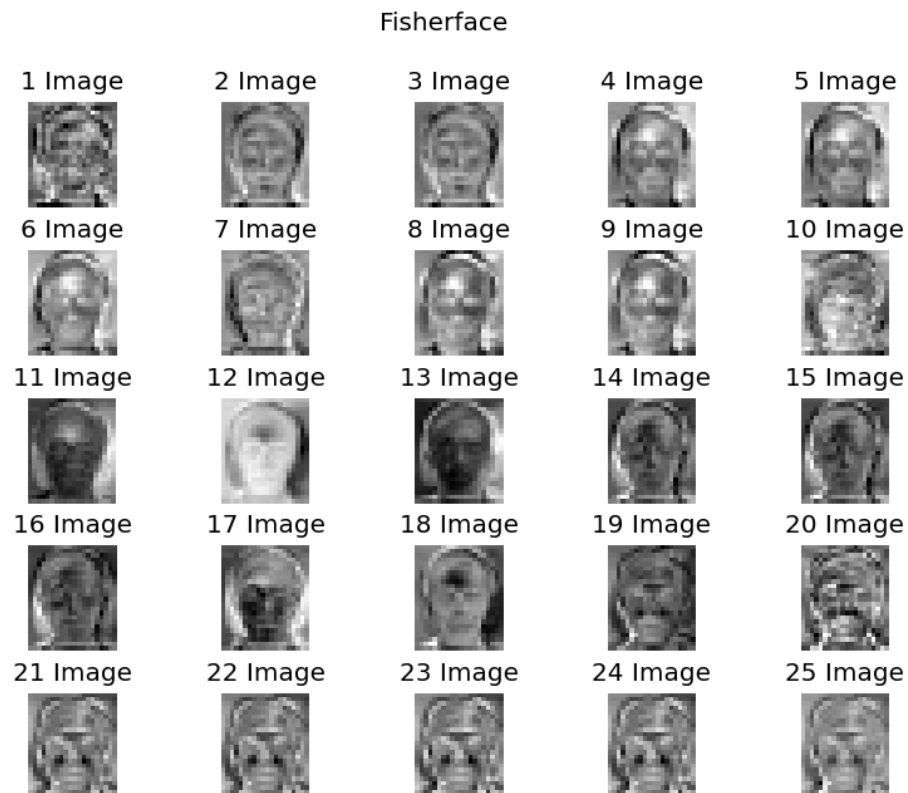
## ### LDA

Reconstruction of 10 random image of testing image

Face reconstruction



## First 25 fisherface



Comparing to PCA , the reconstruction of LDA is hard to distinguish by person. Based on this result , LDA's ability of reconstruction is worse than PCA but for the same person it can reconstruct to the similar result. I think the model still successfully extracts the face feature and performs the function of face recognition.

## ## part2

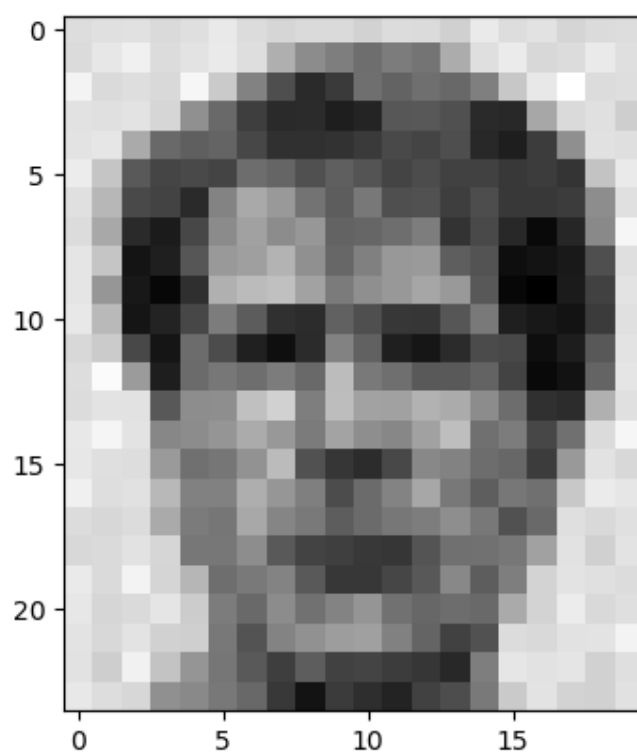
I use  $K = 5$  KNN to compare the result with training images and do the prediction.

## ###PCA

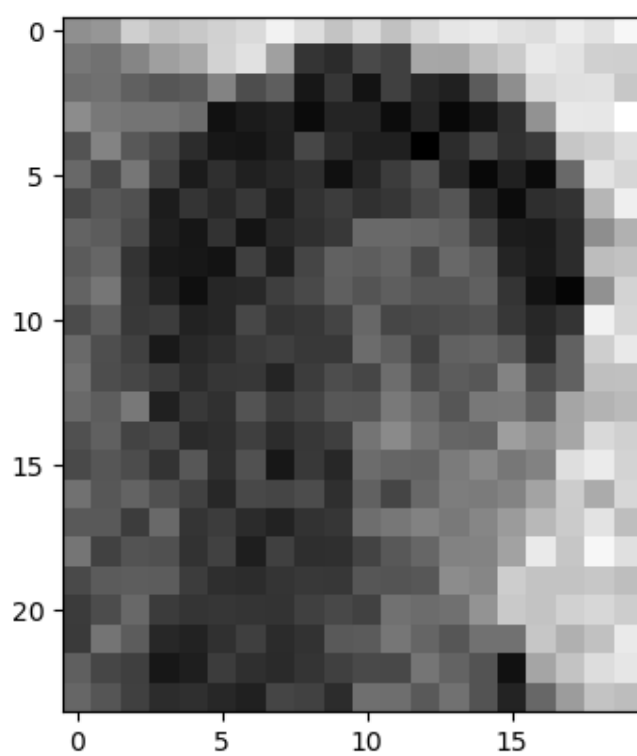
The result of PCA prediction : **Accuracy : 28 / 30 (93.33%)**

Below are the wrong prediction image and right prediction image. It can see that in the wrong prediction the reconstruction result is more blurry and the right prediction can see facial features more clearly.

Right Prediction :



Wrong Prediction :

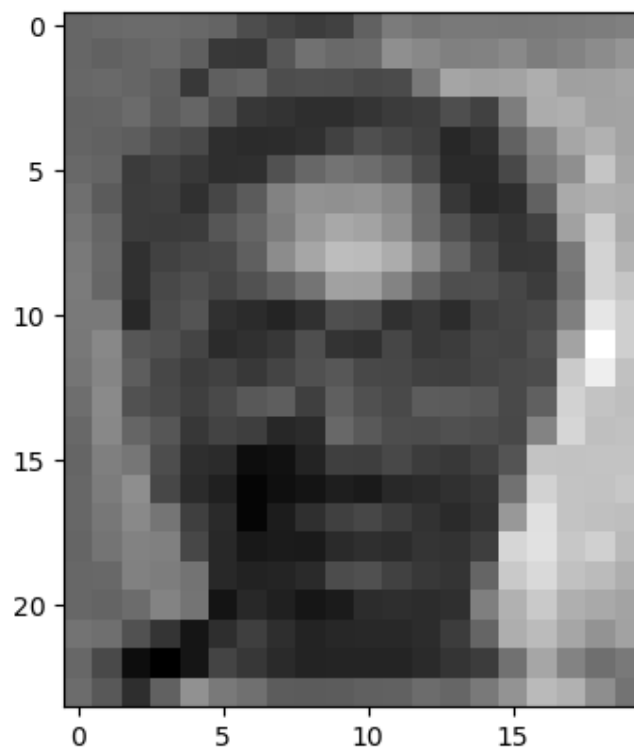


### ### LDA

The result of LDA prediction : `Accuracy : 10 / 30 (33.33%)`

LDA's ability of face recognition is worse than PCA but still has some accuracy. So even if the reconstruction image is hard to distinguish by person , LDA still extracts some useful features that let the result close to the original image to recognize the image.

Right Prediction :



### ## part3

I use the RBF kernel and linear kernel based on cosine similarity on both kernel PCA and kernel LDA. And the hyperparameter of the kernel function(both kernel functions only have 1 hyperparameter ) is simply decided by trying different setting few time and observing the best accuracy setting. In the end for linear kernel the hyperparameter is 1 and RBF is 0.000001.

$$k(x, y) = \exp(-\gamma \|x - y\|^2)$$

$$k(x, y) = \frac{xy^T}{\|x\| \|y\|}$$

For testing , I use training dataset and testing image to create a kernel vector describing the similarity between testing image and training dataset and then use  $W$  to project this kernel vector. Finally comparing the projected kernel vector with  $W$  (In kernel version the row of  $W$  is the coordinate of training image in projected feature space)

Result :

	Kernel PCA	Kernel LDA
Linear kernel	27/30 (90%)	3/10 (10%)
RBF kernel	28/30 (93.33%)	22/30 (73.33%)

In the beginning I use the whole eigenvector got from KPCA & KLDA to test but the result of Kernel LDA is miserable. So I reduce the number of eigenvector got from kernel LDA to 20 (project the data into 20 dimensional space instead) and surprisingly the RBF kernel result becomes much better.

The final result , Kernel PCA projects image into whole 135 dimensional feature space and kernel LDA only projects image into 20 dimensional feature space.

## # t-SNE

### ## part1

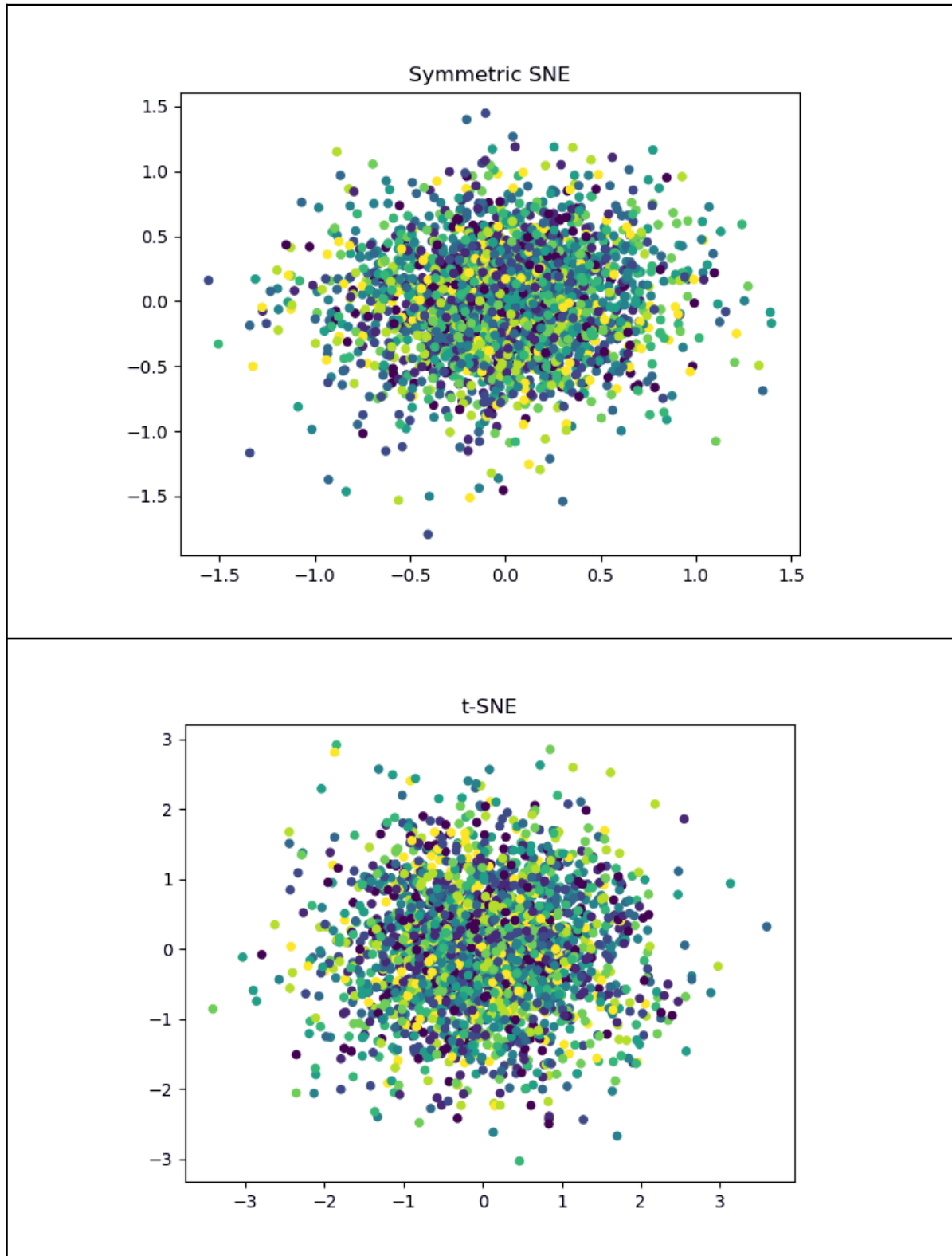
I have two observations in this part. First is that after the survey of SNE I know the probability in high dimensional space is same. So I can directly reuse it.

Second is that according to the printed log the error is converge faster when using symmetric SNE. The change of error may become very small ( $<0.000001$ ) after 450th iterations.

Instead the error of t-SNE is keep reducing so the final error symmetrisc SEN ( $2.0 \times \dots$ ) is bigger than t-SNE ( $1.0 \times \dots$ )

## ## part2

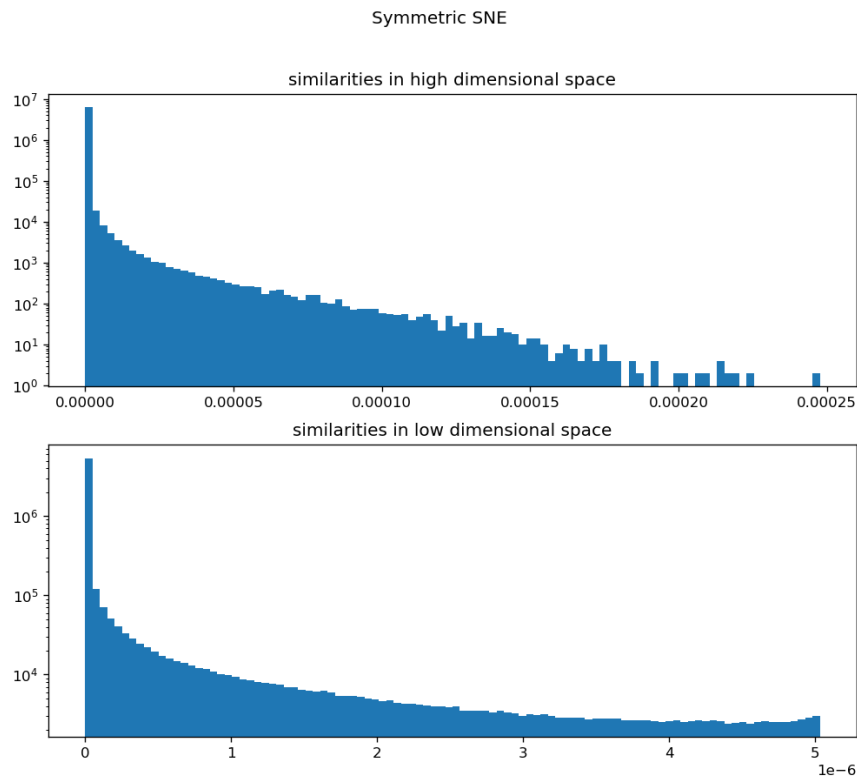
For automatically creating a gif picture , in my implementation I need to save the result as an image so for efficiency I save the result per 5 iterations.



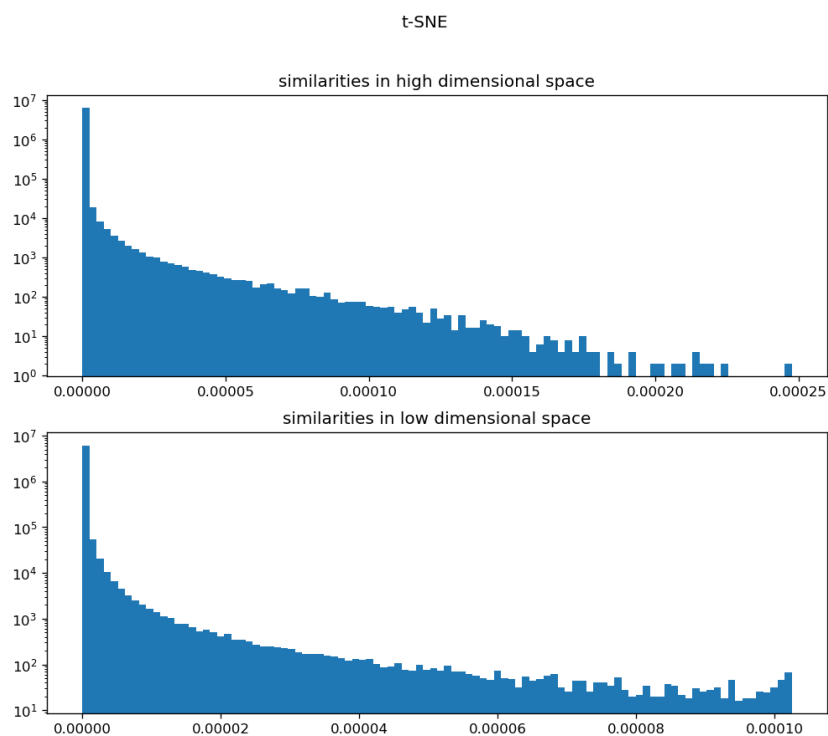
Comparing the two results , it's very obvious that symmetric SNE result is more crowded than t-SNE.

## ## part3

### Symmetric SNE



### t-SNE



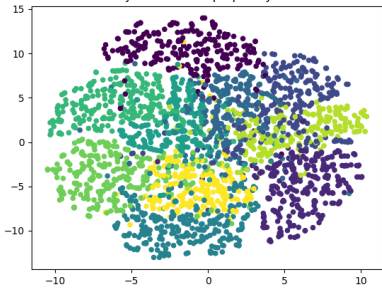
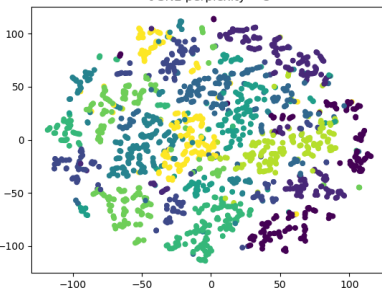
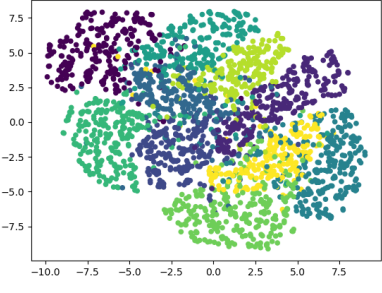
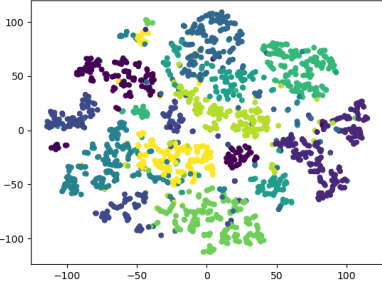
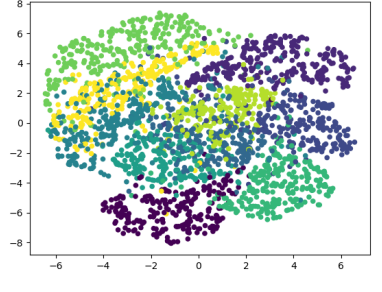
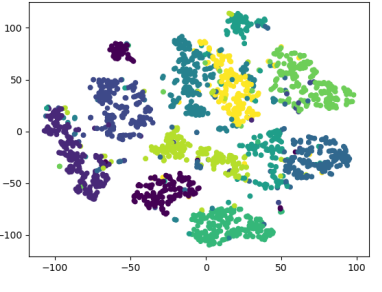
It can be observed that the “tail” of t-SNE low dimensional distribution is longer than symmetric SNE because of the t-student distribution.

## ## part4

The perplexity is a measure of the effective number of neighbors. It is a guess about the number of close neighbors each point has.

If perplexity is small , the local points will dominate the result ,which leads to the more crowded result.

## Symmetric SNE

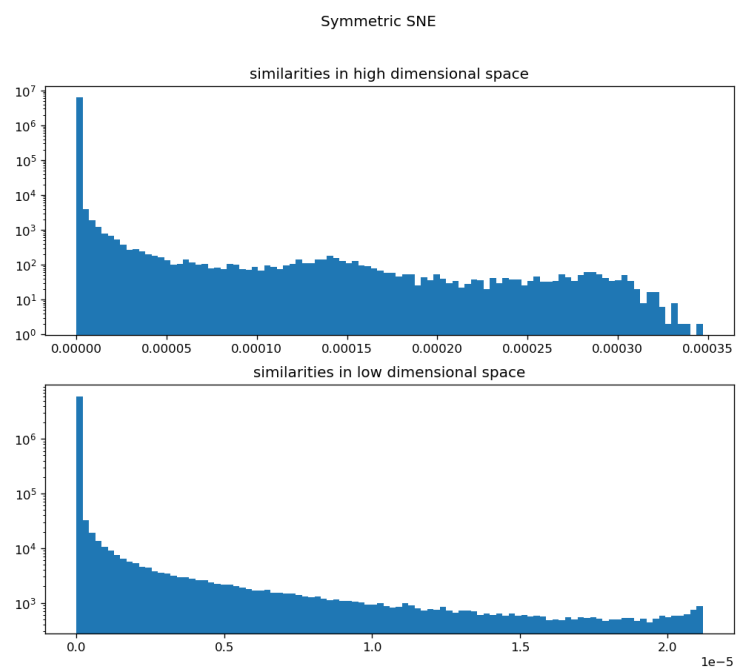
	Symmetric SNE	t-SNE
Perplexity = 3		
Perplexity = 5		
Perplexity = 10		



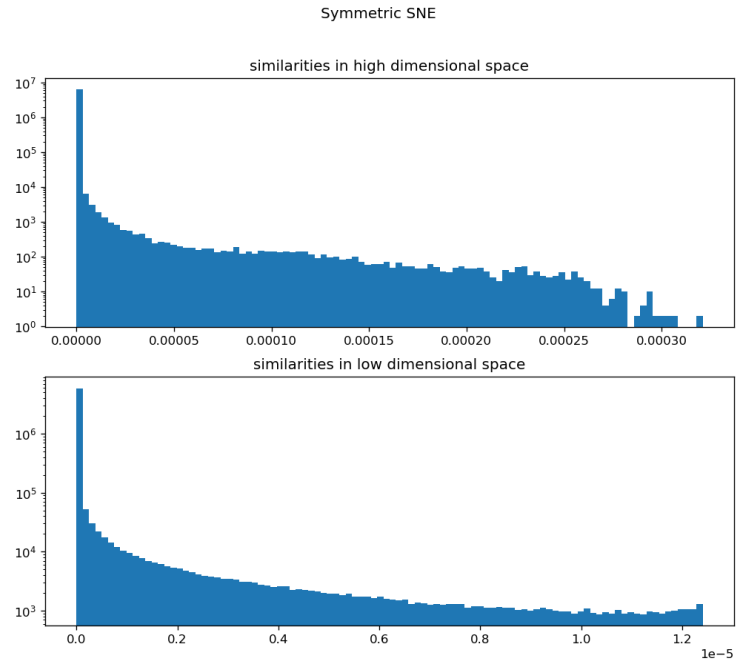
By observing the histogram of distribution of pairwise similarity. We can see the distribution is closer to the left hand side (the value in bottom right is smaller -> distribution is closer to origin) because the smaller perplexity may let the projection result separate incompletely. It may rely more on local variance so the similarity in low dimensional space may be lower.

### Symmetric SNE similarity distribution

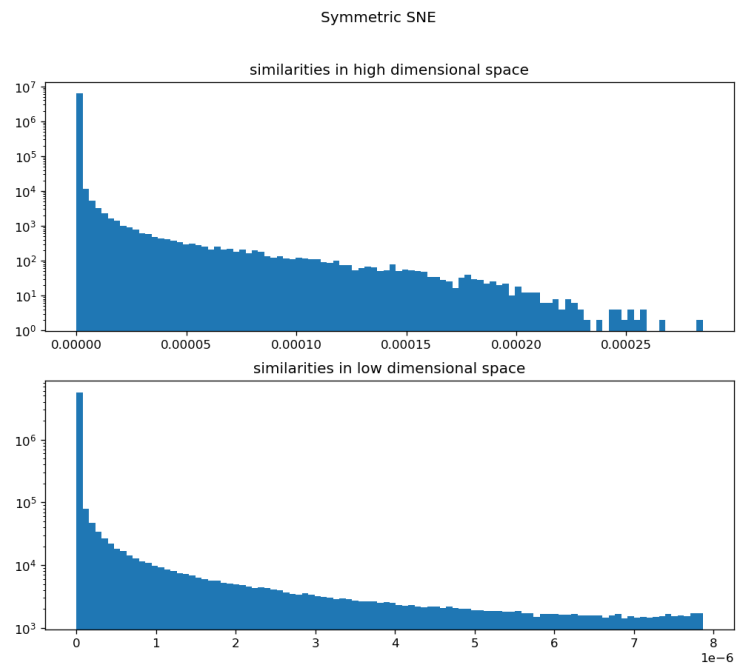
Perplexity = 3



Perplexity = 5

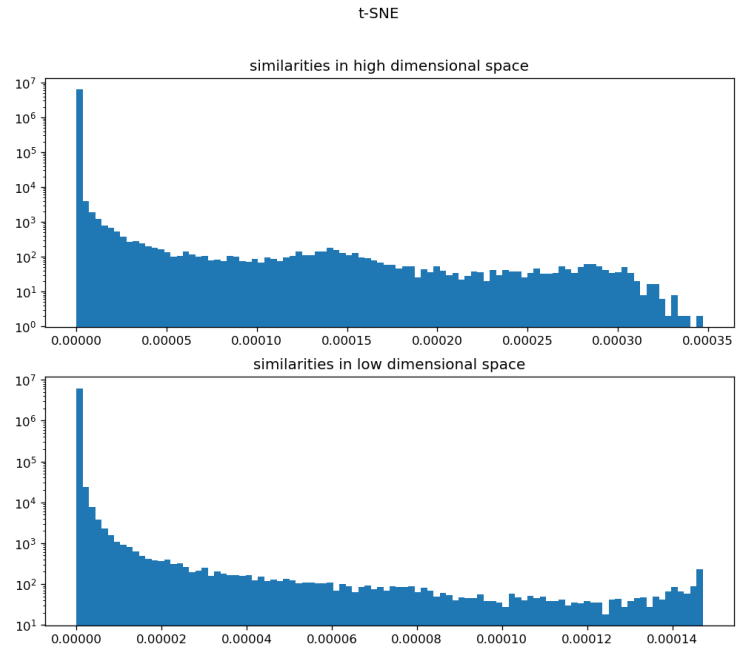


Perplexity = 10

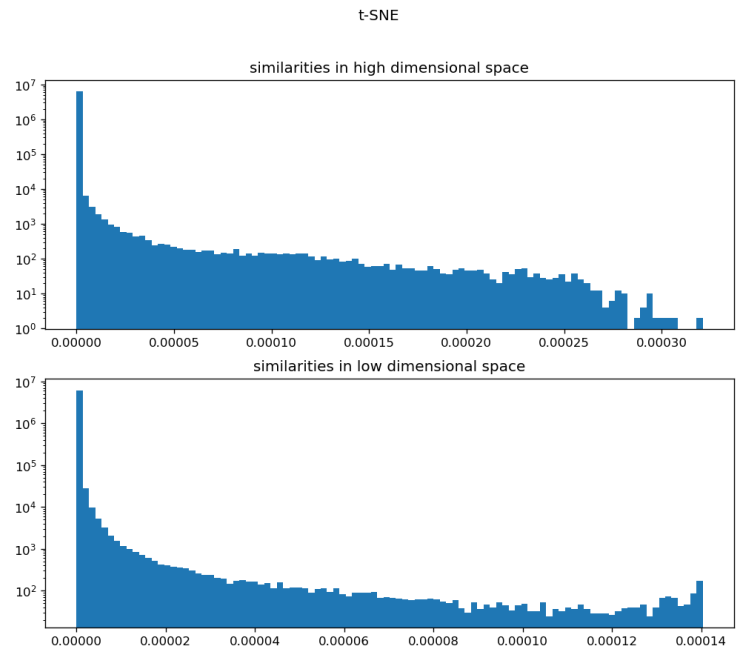


t-SNE similarity distribution

Perplexity = 3



Perplexity = 5



Perplexity = 10

