HW7 report

309552026
鄭偉丞

I. Kernel Eigenfaces

a. Code explanation

```
if __name__ == "__main__":
    train_data, train_label = load("./Yale_Face_Database/Training/", 9)
    test_data, test_label = load("./Yale_Face_Database/Testing/", 2)

    k = 3
    for kernel_mode in ["none", "linear", "poly", "RBF"]:
        print("kernel: " + kernel_mode)
        pca_transform, pca_z = PCA(train_data, train_label, test_data, test_label, k, kernel_mode)
        LDA(pca_transform, pca_z, train_data, train_label, test_data, test_label, k, kernel_mode)
```

The main procedure of this program is 1. Loading data. 2. Do PCA and LDA with different kernels. 3. In both PCA and LDA, it has testing part to test the accuracy.

```
def PCA(train_data, train_label, test_data, test_label, k, mode):
    mean = np.mean(train_data, axis=0)
    center = train_data - mean

    S = kernel(mode, train_data)

    eigenValue, eigenVector = np.linalg.eig(S)
    index = np.argsort(-eigenValue) # inverse -> max sort
    eigenValue = eigenValue[index]
    eigenVector = eigenVector[:,index]

    # remove negtive eigenValue
    for i in range(len(eigenValue)):
        if (eigenValue[i] <= 0):
            eigenValue = eigenValue[:i].real
            eigenVector = eigenVector[:, :i].real
            break

    transform = center.T@eigenVector
    show_transform(transform)

    z = transform.T @ center.T
    reconstruct = transform @ z + mean.reshape(-1, 1)
    index = np.random.choice(135, 10, replace=False)
    show_reconstruct(train_data[index], reconstruct[:, index])

    # test
    test("PCA", transform, z, train_data, train_label, test_data, test_label, mean)

    return transform, z
```

This is the main function for doing PCA. In kernel function at 4th line in the figure performs the kernel function show later. 1st, if mode=”none”, it means it’s not kernel PCA that is S=cov(data) in kernel function.

2nd, calculate the eigenvalues and eigenvectors and sort then from max to min. Then remove negative elements in them.

3rd, calculate the transform matrix by multiply different image data center and eigenvectors to get eigenfaces. Then use show_transform to show eigenfaces.

4th, for reconstruct data, we need to mix the eigenface matrix and the Image data center and add Image mean in origin feature space by reshape it. Then random choice 10 reconstruct face and show their origin face by show_reconstruct function.

5th, test the accuracy. The detail will show later. Since LDA will use some outputs from PCA, the function return transform matrix and z.

```
def LDA(pca_transform, pca_z, train_data, train_label, test_data, test_label, k, mode):
    mean = np.mean(pca_z, axis=1)
    N = pca_z.shape[0] # (134, 135)

    S_within = np.zeros((N, N))
    for i in range(15):
        S_within += np.cov(pca_z[:, i*9:i*9+9], bias=True)

    S_between = np.zeros((N, N))
    for i in range(15):
        class_mean = np.mean(pca_z[:, i*9:i*9+9], axis=1).T
        S_between += 9 * (class_mean - mean) @ (class_mean - mean).T

    S = np.linalg.inv(S_within) @ S_between
    eigenValue, eigenVector = np.linalg.eig(S)
    index = np.argsort(-eigenValue) # inverse -> max sort
    eigenValue = eigenValue[index]
    eigenVector = eigenVector[:, index]

    # remove negtive eigenValue
    for i in range(len(eigenValue)):
        if (eigenValue[i] <= 0):
            eigenValue = eigenValue[:i].real
            eigenVector = eigenVector[:, :i].real
            break

    transform = pca_transform @ eigenVector
    show_transform(transform)

    mean = np.mean(train_data, axis=0)
    center = train_data - mean
    z = transform.T @ center.T
    reconstruct = transform @ z + mean.reshape(-1, 1)
    show_reconstruct(train_data, reconstruct)

    # test
    test("LDA", transform, z, train_data, train_label, test_data, test_label, mean)
```

LDA function is similar to PCA. They are different at the beginning part. In LDA, we

want to consider $\mathrm{J(w)} = \frac{\mathrm{w^T} S_B w}{\mathrm{w^T} S_W w}$. So I calculate S_between_class and S_within_class first and let $\mathrm{S} = \mathrm{S_w^{-1}} S_B$. The rest parts are all the same as PCA.

```
def kernel(mode, data):
    if mode == "none":
        S = np.cov(data, bias=True)
    else:
        if mode == "linear":
            S = data @ data.T
        elif mode == "poly":
            S = (0.01 * data @ data.T)**3
        elif mode == "RBF":
            S = np.exp(-0.01*cdist(data, data, 'sqeuclidean'))
        N = data.shape[0]
        one_N = np.ones((N, N))/N
        S = S - one_N @ S - S @ one_N + one_N @ S @ one_N
    return S
```

The kernel function checks whether mode == none first. If there's not kernel PCA and kernel LDA, it will return covariance. In kernel mode, I choice three common kernels, linear, polynomial and RBF kernel. And then make $\mathrm{K^C} = K - 1_N K - K 1_N + 1_N K 1_N$.

```
def test(testItem, transform, z, train_data, train_label, test_data, test_label, mean):
    test_z = transform.T @ (test_data - mean).T
    dist = np.zeros(train_data.shape[0])
    acc = 0
    for i in range(test_data.shape[0]):
        for j in range(train_data.shape[0]):
            dist[j] = cdist(test_z[:, i].reshape(1, -1), z[:, j].reshape(1, -1), 'sqeuclidean')
        knn = train_label[np.argsort(dist)[:k]]
        uniq_knn, uniq_knn_count = np.unique(knn, return_counts=True)
        predict = uniq_knn[np.argmax(uniq_knn_count)]

        # print(f"{predict}, {test_label[i]}")
        if predict == test_label[i]:
            acc += 1

    print(testItem+f" acc: {100*acc/test_data.shape[0]:.2f}%")
```

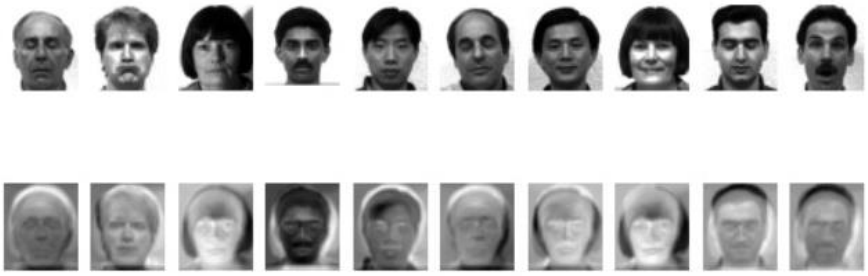
The testing part are used in both PCA and LDA part. It computes z for test data and calculates the distance between training z and testing z. Find the kth nearest neighborhood of the test data to determine which class it is. Finally get the accuracy.

b. <u>Results</u>

<u>PCA:</u>

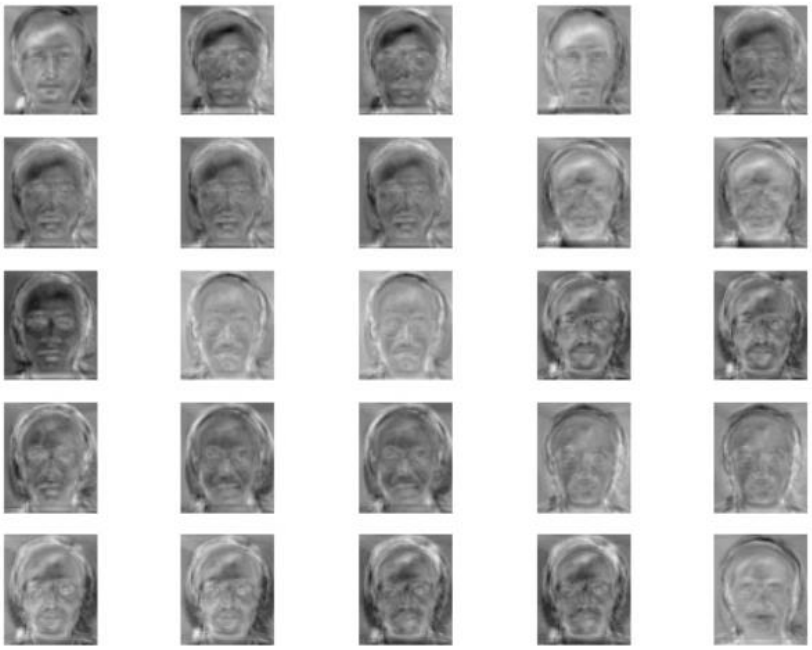
<u>Eigenface:</u>

Reconstruct (10 origin image and 10 reconstruct image):

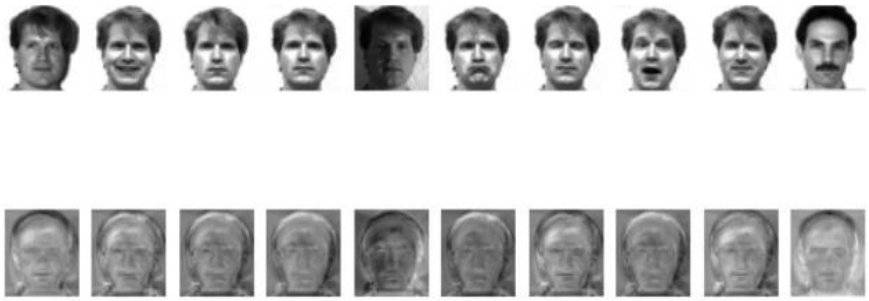


LDA:

Fisherface:



Reconstruct (10 origin image and 10 reconstruct image):



Testing accuracy using different kernel:

```
kernel: none
PCA acc: 83.33%
LDA acc: 73.33%
kernel: linear
PCA acc: 80.00%
LDA acc: 60.00%
kernel: poly
PCA acc: 83.33%
LDA acc: 60.00%
kernel: RBF
PCA acc: 76.67%
LDA acc: 83.33%
```

c. Observations

1. It shows in whatever kernel we use, PCA acc are all not bad. However, LDA acc are not so good in linear and polynomial kernel.
2. The eigenface with higher eigenvalue looks 'clearer' for human eyes.
3. Reconstruct faces are all fuzzy-looking.

II. t-SNE

a. **Code explanation:**

```
if mode == "tsne":
    num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y)) # t-SNE
elif mode == "ssne":
    num = np.exp(-1. * np.add(np.add(num, sum_Y).T, sum_Y)) # s-SNE
```

```
for i in range(n):
    if mode == "tsne":
        dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0) # t-SNE
    elif mode == "ssne":
        dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0) # s-SNE
```

The different code between s-SNE and t-SNE. The difference between them is the way to calculate q and the gradient. Since when we mapping high-dim data to low-

dim, not all features will separate so well. Some feature will be overlapping in low-dim, which is crowding problem. s-SNE uses union probability and optimize the function to get gradient. However, it doesn't deal with the problem. T-SNE uses t-distribution in the low-dim to solve crowding problem.

```
# 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))

    if mode == "tsne":
        saveImage(Y, labels, [-120, 120], f"./tsne_output/{int(perplexity)}/", iter+1)
    elif mode == "ssne":
        saveImage(Y, labels, [-10, 10], f"./ssne_output/{int(perplexity)}/", iter+1)
```

```
    X = np.loadtxt("./tsne_python/mnist2500_X.txt")
    labels = np.loadtxt("./tsne_python/mnist2500_labels.txt")
    for mode in ["ssne", "tsne"]:
        for perplexity in [10., 20., 30., 40., 50.]:
            Y = sne(mode, X, 2, 50, perplexity)
            output_dir = "./" + mode + f"_output/{int(perplexity)}/"
            showgif(output_dir)
        # pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
        # pylab.show()
```

```
def saveImage(Y, labels, lim, output_dir, iter):
    pylab.clf()
    pylab.xlim(lim)
    pylab.ylim(lim)
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    pylab.savefig(output_dir+f"{iter}.png")

def showgif(output_dir):
    gif = []
    for i in range(100):
        gif.append(Image.open(output_dir+f"{i+1}0.png"))
    gif[0].save(output_dir+f"result.gif",
                format='GIF',
                save_all=True,
                append_images=gif[1:],
                duration=400, loop=0)
```

Save the result Image for every 10 iterations. After all finish, read all result images to generate gif file and save the file. The code in second figure with red line is the for loop to try different perplexity.

```
    saveSimilarities(P, Q, mode, perplexity)

    # Return solution
    return Y
```

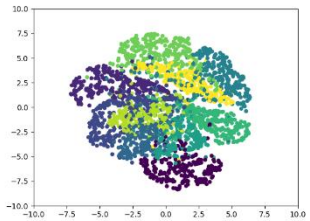
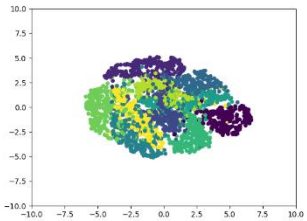
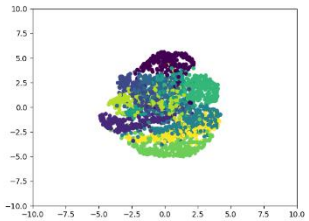
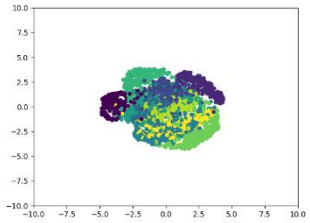
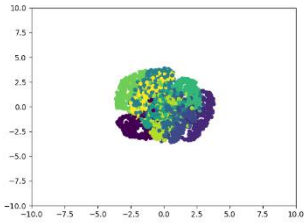
```
def saveSimilarities(P, Q, mode, perplexity):
    pylab.subplot(2,1,1)
    pylab.title(mode+" high-dim")
    pylab.hist(P.flatten(),bins=40,log=True)
    pylab.subplot(2,1,2)
    pylab.title(mode+" low-dim")
    pylab.hist(Q.flatten(),bins=40,log=True)
    pylab.savefig("./"+mode+f"_output/similarities_{int(perplexity)}.png")
```

Before SNE function return, use P and Q to show similarities. Input mode and perplexity are for output_dir.

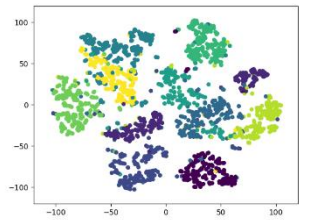
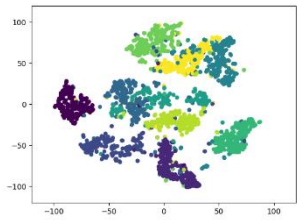
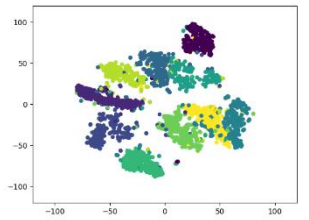
b. Results：

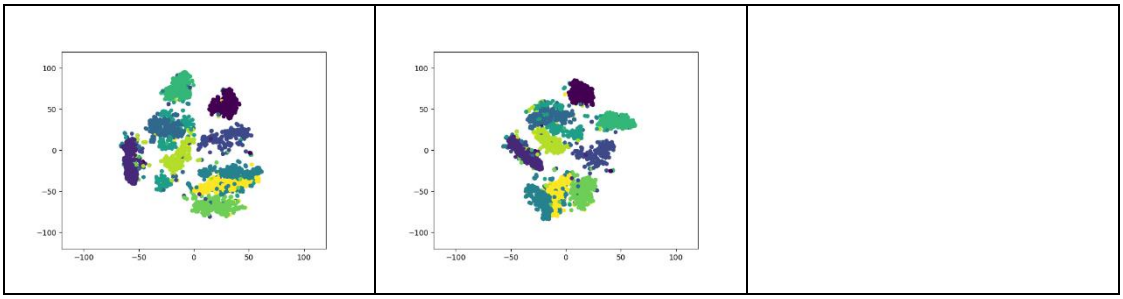
Result after 1000 iterations with different perplexity:

S-SNE:

10	20	30
		
40	50	
		

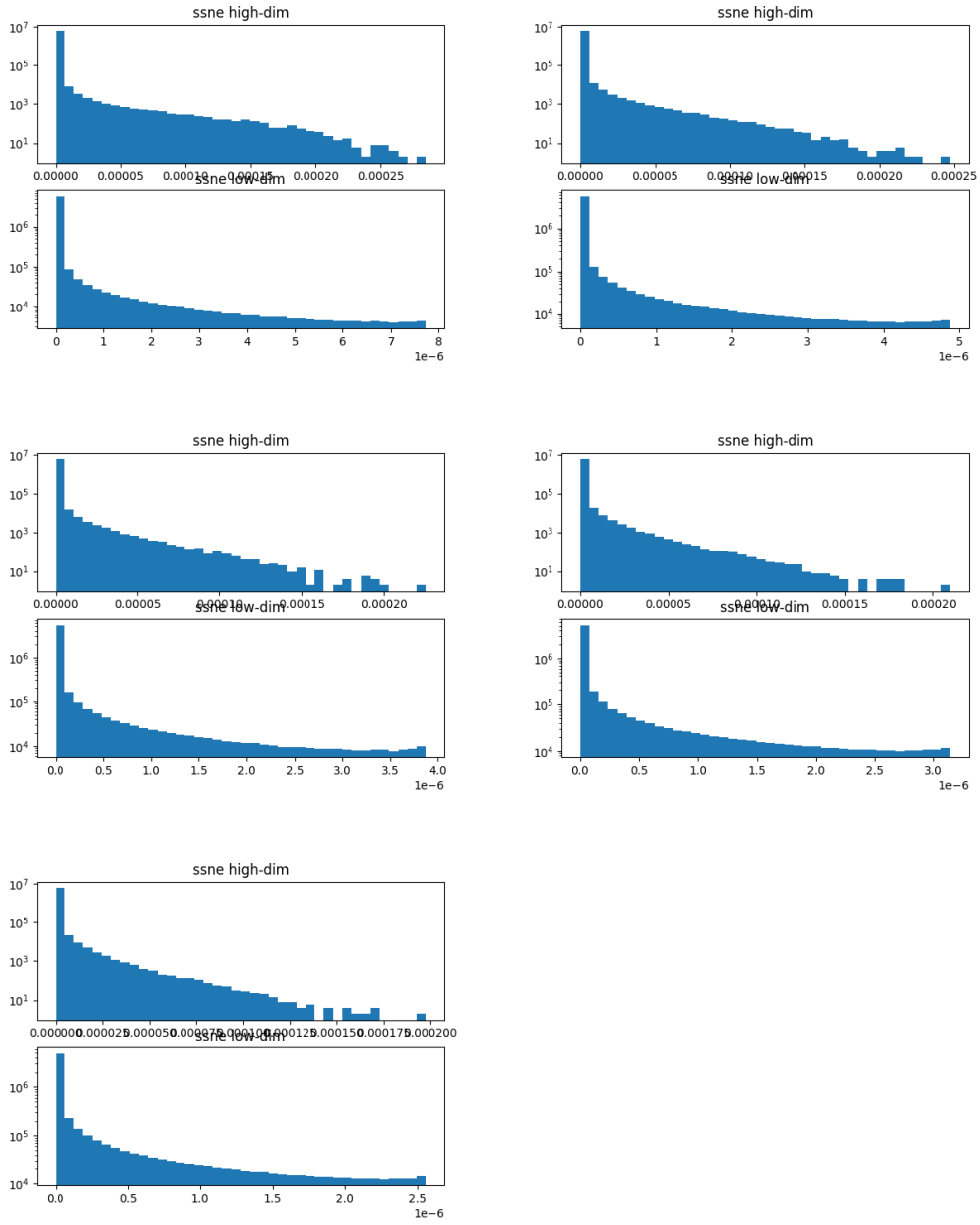
t-SNE:

10	20	30
		
40	50	

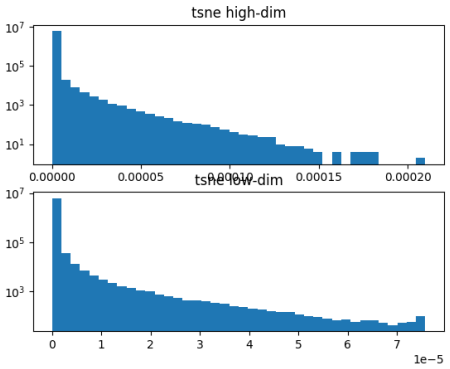


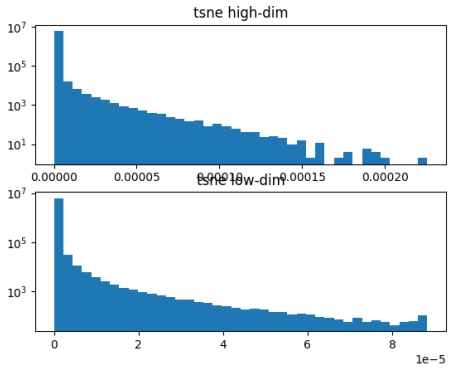
The pairwise similarity in different perplexity (10 to 50 for each):

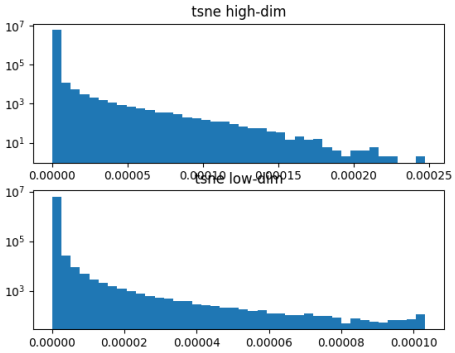
S-SNE:

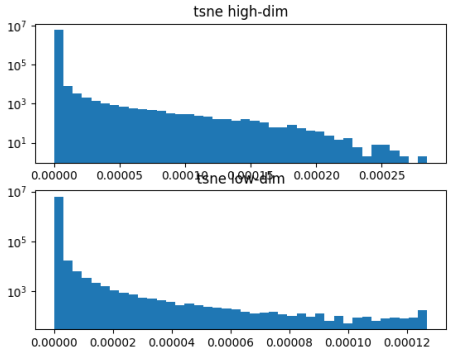


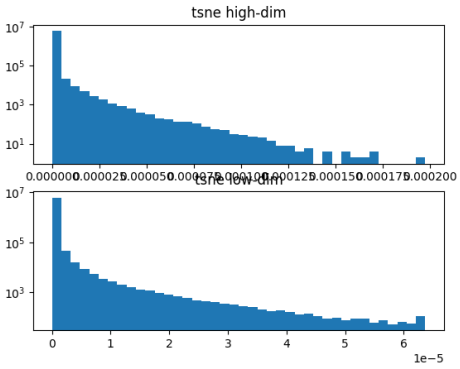
t-SNE:











c. **Observation:**

For result:

1. Results of s-SNE are more crowded than t-SNE looks like t-SNE can classify the data better.
2. Both s-SNE and t-SNE results are more crowded with high-perplexity than low-perplexity.

For pairwise similarity:

1. The pairwise similarity is independent in shape with perplexity in both s-SNE and t-SNE.

2. The range of result in s-SNE are larger than t-SNE. So that t-SNE has less crowd problem.