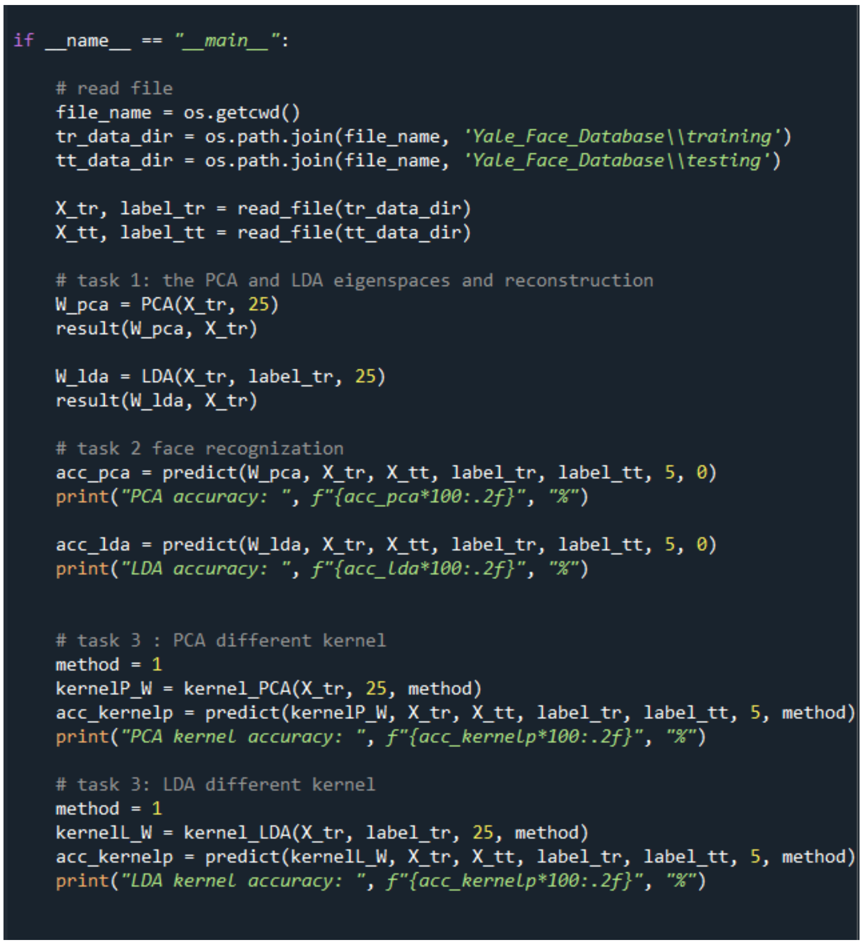
312554006 Machine Learning 羅名志 HW7

1. Code with detailed explanations

A. Kernel Eigenfaces

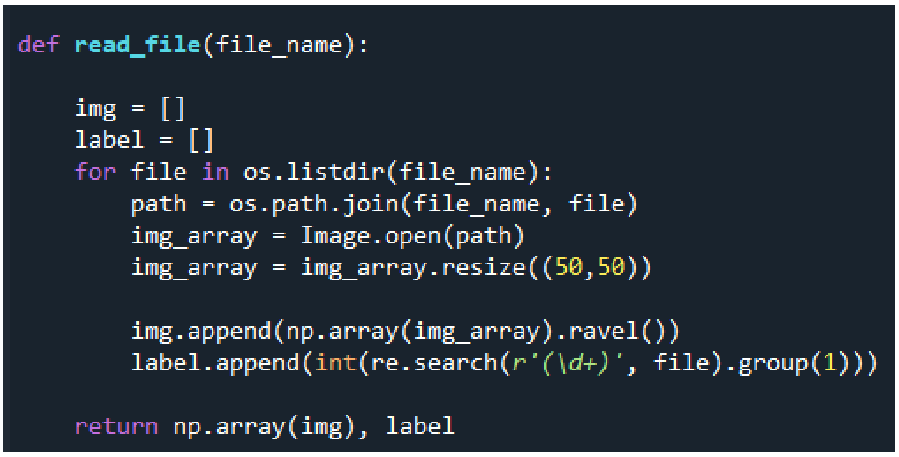
(1) Part 1:

a. The overall code structure, divided into 3 tasks



```python
if __name__ == "__main__":

    # read file
    file_name = os.getcwd()
    tr_data_dir = os.path.join(file_name, 'Yale_Face_Database\\training')
    tt_data_dir = os.path.join(file_name, 'Yale_Face_Database\\testing')

    X_tr, label_tr = read_file(tr_data_dir)
    X_tt, label_tt = read_file(tt_data_dir)

    # task 1: the PCA and LDA eigenspaces and reconstruction
    W_pca = PCA(X_tr, 25)
    result(W_pca, X_tr)

    W_lda = LDA(X_tr, label_tr, 25)
    result(W_lda, X_tr)

    # task 2 face recognization
    acc_pca = predict(W_pca, X_tr, X_tt, label_tr, label_tt, 5, 0)
    print("PCA accuracy: ", f"{acc_pca*100:.2f}", "%")

    acc_lda = predict(W_lda, X_tr, X_tt, label_tr, label_tt, 5, 0)
    print("LDA accuracy: ", f"{acc_lda*100:.2f}", "%")


    # task 3 : PCA different kernel
    method = 1
    kernelP_W = kernel_PCA(X_tr, 25, method)
    acc_kernelp = predict(kernelP_W, X_tr, X_tt, label_tr, label_tt, 5, method)
    print("PCA kernel accuracy: ", f"{acc_kernelp*100:.2f}", "%")

    # task 3: LDA different kernel
    method = 1
    kernelL_W = kernel_LDA(X_tr, label_tr, 25, method)
    acc_kernelp = predict(kernelL_W, X_tr, X_tt, label_tr, label_tt, 5, method)
    print("LDA kernel accuracy: ", f"{acc_kernelp*100:.2f}", "%")
```

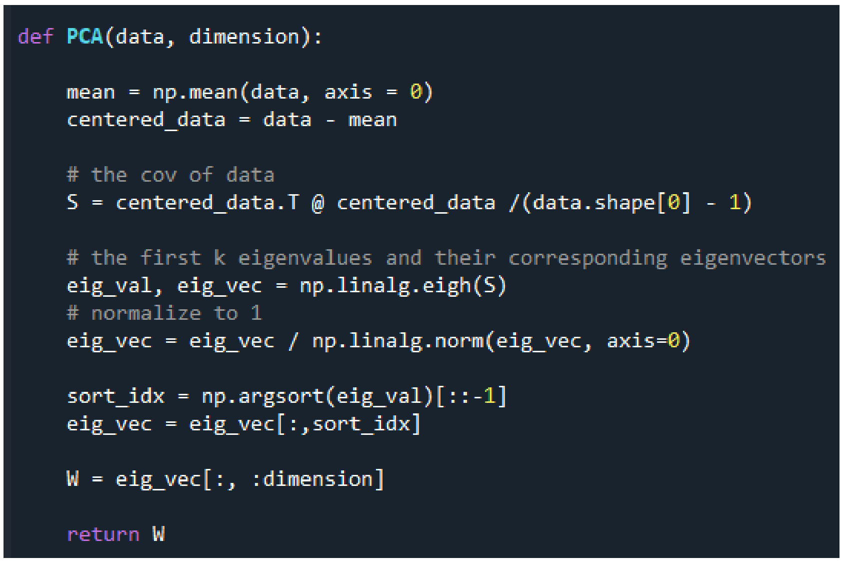
b. Read the pgm file and turn into specified dimension



```python
def read_file(file_name):

    img = []
    label = []
    for file in os.listdir(file_name):
        path = os.path.join(file_name, file)
        img_array = Image.open(path)
        img_array = img_array.resize((50,50))

        img.append(np.array(img_array).ravel())
        label.append(int(re.search(r'(\d+)', file).group(1)))

    return np.array(img), label
```

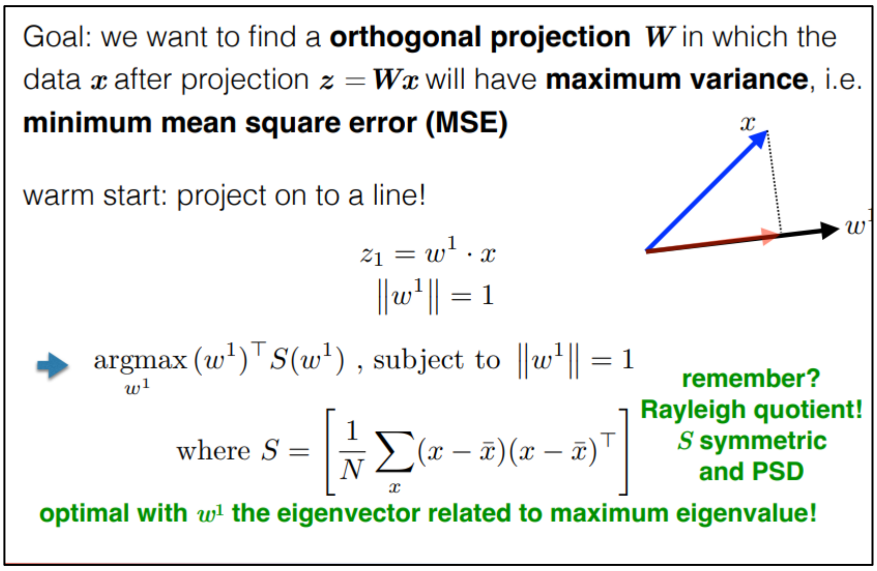
Turn the image shape into 50*5o and ravel it, so each image we have 2500 features. (the imgae file size is num of images* 2500) Also, we record their corressponding labels to use in prediction and LDA.

c. The PCA



```python
def PCA(data, dimension):

    mean = np.mean(data, axis = 0)
    centered_data = data - mean

    # the cov of data
    S = centered_data.T @ centered_data /(data.shape[0] - 1)

    # the first k eigenvalues and their corresponding eigenvectors
    eig_val, eig_vec = np.linalg.eigh(S)
    # normalize to 1
    eig_vec = eig_vec / np.linalg.norm(eig_vec, axis=0)

    sort_idx = np.argsort(eig_val)[::-1]
    eig_vec = eig_vec[:,sort_idx]

    W = eig_vec[:, :dimension]

    return W
```

The structure of the PCA function is followed by the lecture slides:



Goal: we want to find a **orthogonal projection** $\boldsymbol{W}$ in which the data $\boldsymbol{x}$ after projection $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$ will have **maximum variance**, i.e. **minimum mean square error (MSE)**

warm start: project on to a line!

$$z_1 = w^1 \cdot x$$
$$\|w^1\| = 1$$

$$\underset{w^1}{\operatorname{argmax}}\,(w^1)^\top S(w^1)\ ,\ \text{subject to}\ \|w^1\| = 1$$

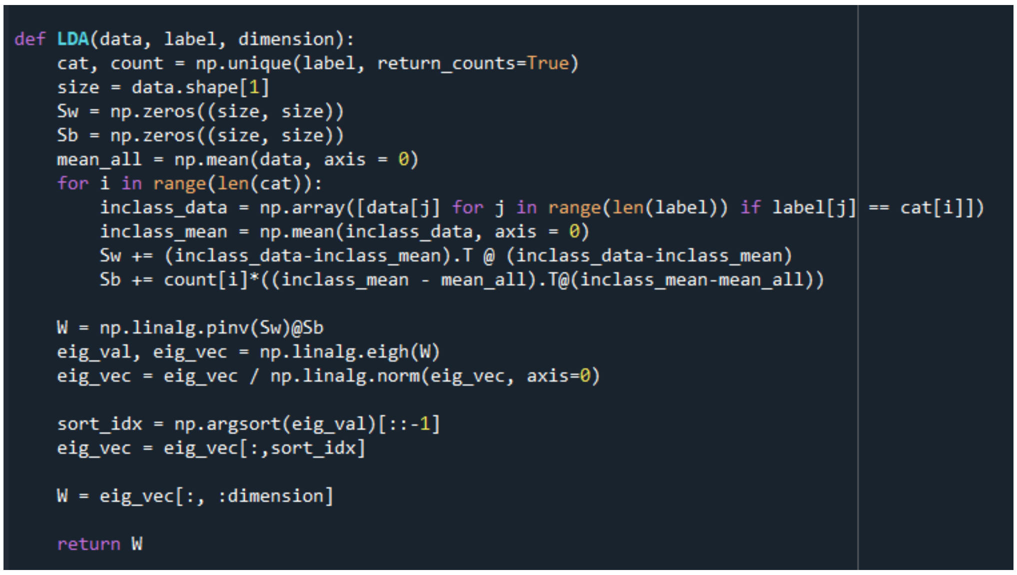
$$\text{where } S = \left[\frac{1}{N}\sum_x (x - \bar{x})(x - \bar{x})^\top\right]$$

remember? Rayleigh quotient! S symmetric and PSD

optimal with w^1 the eigenvector related to maximum eigenvalue!

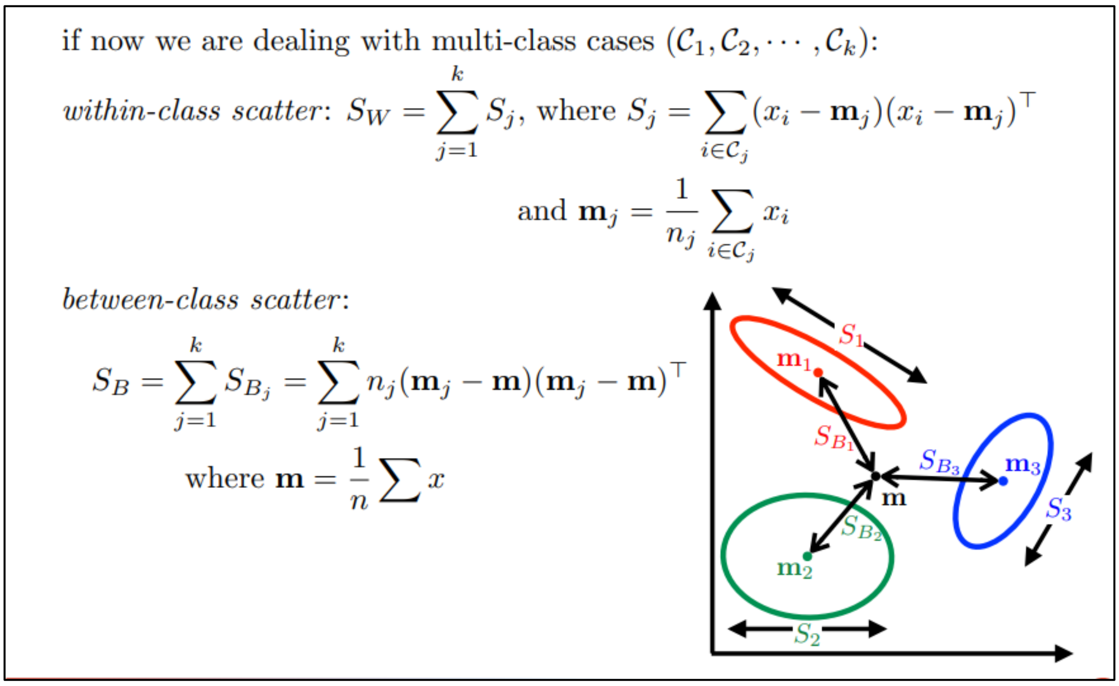
So, we first get the covariance matrix and find its eigenvalue and eigenvector. Then, using the sorted eigenvector, w construct the W(wirght) matrix.

d.LDA



```
def LDA(data, label, dimension):
    cat, count = np.unique(label, return_counts=True)
    size = data.shape[1]
    Sw = np.zeros((size, size))
    Sb = np.zeros((size, size))
    mean_all = np.mean(data, axis = 0)
    for i in range(len(cat)):
        inclass_data = np.array([data[j] for j in range(len(label)) if label[j] == cat[i]])
        inclass_mean = np.mean(inclass_data, axis = 0)
        Sw += (inclass_data-inclass_mean).T @ (inclass_data-inclass_mean)
        Sb += count[i]*((inclass_mean - mean_all).T@(inclass_mean-mean_all))

    W = np.linalg.pinv(Sw)@Sb
    eig_val, eig_vec = np.linalg.eigh(W)
    eig_vec = eig_vec / np.linalg.norm(eig_vec, axis=0)

    sort_idx = np.argsort(eig_val)[::-1]
    eig_vec = eig_vec[:,sort_idx]

    W = eig_vec[:, :dimension]

    return W
```

Also follow the lecture slides about LDA:



if now we are dealing with multi-class cases $(\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_k)$:

within-class scatter: $S_W = \sum_{j=1}^{k} S_j$, where $S_j = \sum_{i \in \mathcal{C}_j} (x_i - \mathbf{m}_j)(x_i - \mathbf{m}_j)^\top$

and $\mathbf{m}_j = \frac{1}{n_j} \sum_{i \in \mathcal{C}_j} x_i$

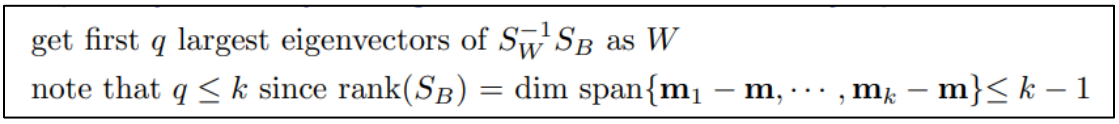
between-class scatter:

$$S_B = \sum_{j=1}^{k} S_{B_j} = \sum_{j=1}^{k} n_j (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^\top$$

where $\mathbf{m} = \frac{1}{n} \sum x$

We first calculate the Sw and Sb matrix by the definition. For Sb, we deduct each in-class mean by overall mean, and multiply with each group nums(9 in training sets) And for Sw, we sum each group’s in-class distance.

Lastly, refer to the following instruction, we calculate the W(weight) matrix



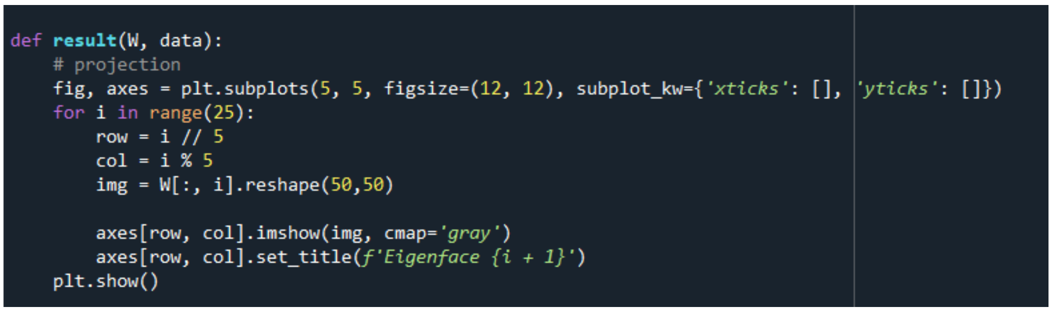
get first q largest eigenvectors of $S_W^{-1}S_B$ as W

note that $q \leq k$ since $\text{rank}(S_B) = \dim \text{span}\{\mathbf{m}_1 - \mathbf{m}, \cdots, \mathbf{m}_k - \mathbf{m}\} \leq k-1$

Same as what we do in the PCA, first sort it and return back the first k eigenvector.

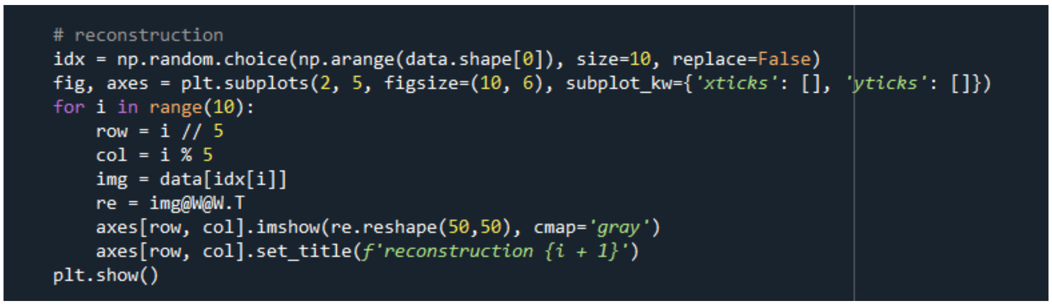
E. the result

After getting PCA and LDA weight, we use them to get the eigenfaces and fisherfaces.



```python
def result(W, data):
    # projection
    fig, axes = plt.subplots(5, 5, figsize=(12, 12), subplot_kw={'xticks': [], 'yticks': []})
    for i in range(25):
        row = i // 5
        col = i % 5
        img = W[:, i].reshape(50,50)

        axes[row, col].imshow(img, cmap='gray')
        axes[row, col].set_title(f'Eigenface {i + 1}')
    plt.show()
```

First, we reshape our W matrix to shape (50,50) to see the different faces respectively.

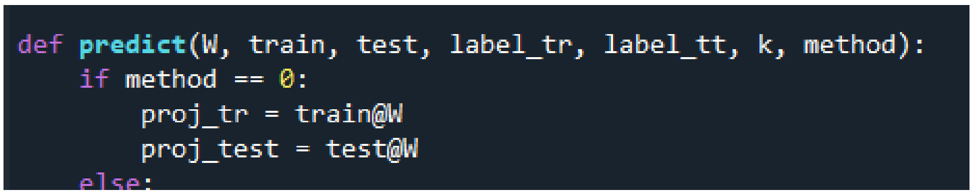


```python
    # reconstruction
    idx = np.random.choice(np.arange(data.shape[0]), size=10, replace=False)
    fig, axes = plt.subplots(2, 5, figsize=(10, 6), subplot_kw={'xticks': [], 'yticks': []})
    for i in range(10):
        row = i // 5
        col = i % 5
        img = data[idx[i]]
        re = img@W@W.T
        axes[row, col].imshow(re.reshape(50,50), cmap='gray')
        axes[row, col].set_title(f'reconstruction {i + 1}')
    plt.show()
```

And we reconstruct the image by using xWW_transpose, to show the reconstruction images by randomly chosen images.

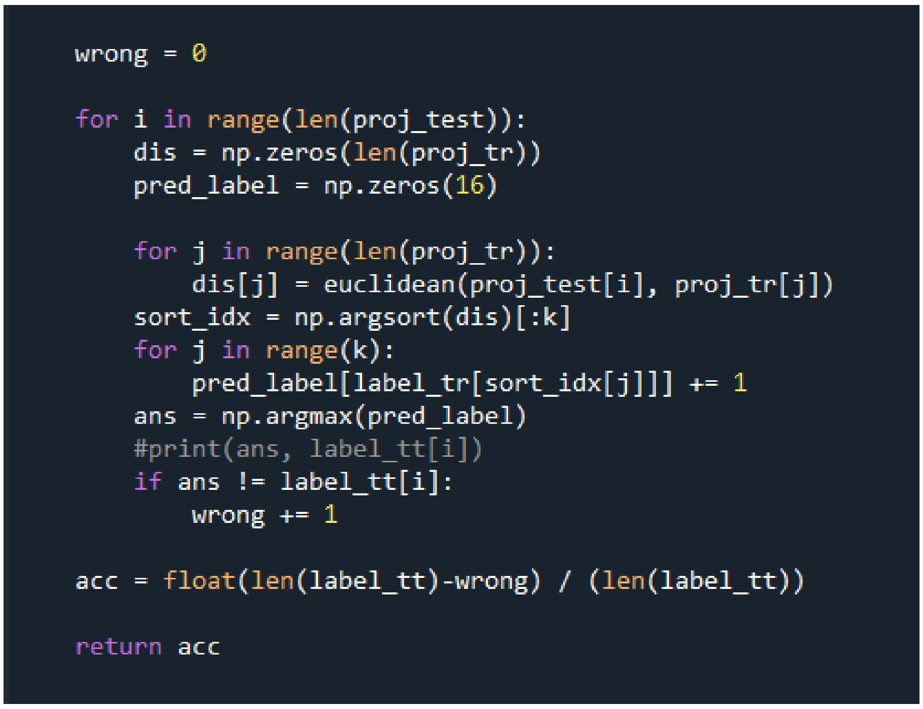
(2) Part 2:

Predict the performance, and using k-nn to specify each category that



```
def predict(W, train, test, label_tr, label_tt, k, method):
    if method == 0:
        proj_tr = train@W
        proj_test = test@W
    else:
```

Method == 0, meaning that here we don’y use the kernel function

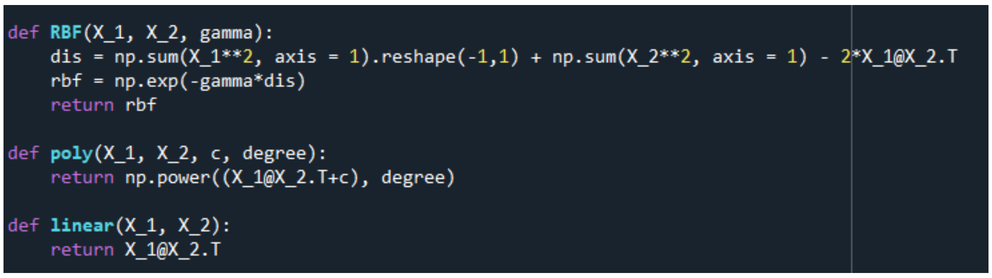


```
    wrong = 0

    for i in range(len(proj_test)):
        dis = np.zeros(len(proj_tr))
        pred_label = np.zeros(16)

        for j in range(len(proj_tr)):
            dis[j] = euclidean(proj_test[i], proj_tr[j])
        sort_idx = np.argsort(dis)[:k]
        for j in range(k):
            pred_label[label_tr[sort_idx[j]]] += 1
        ans = np.argmax(pred_label)
        #print(ans, label_tt[i])
        if ans != label_tt[i]:
            wrong += 1

    acc = float(len(label_tt)-wrong) / (len(label_tt))

    return acc
```

After project the data by using the Weight we calculated, we can find the closet k neighbors (inner loop) in training sets for each testeing instance(outer loop). Here the k we use is 5.

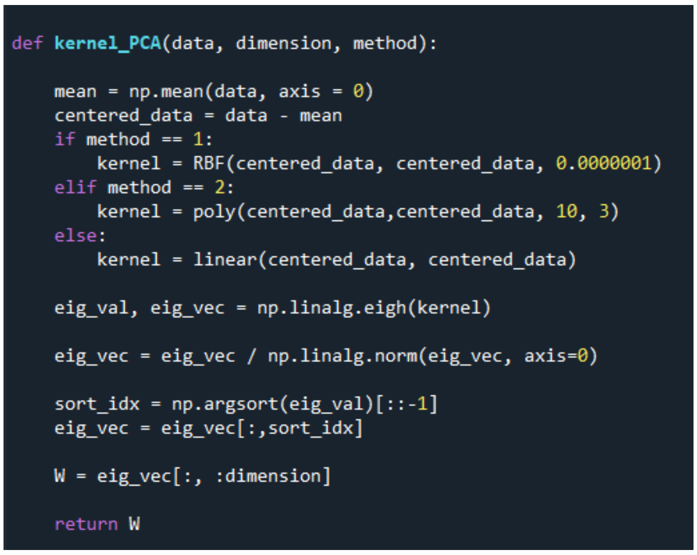
(3) Part 3:

a. The kernel functio I use in this hw: RBF, linear, ploynomial



```
def RBF(X_1, X_2, gamma):
    dis = np.sum(X_1**2, axis = 1).reshape(-1,1) + np.sum(X_2**2, axis = 1) - 2*X_1@X_2.T
    rbf = np.exp(-gamma*dis)
    return rbf

def poly(X_1, X_2, c, degree):
    return np.power((X_1@X_2.T+c), degree)

def linear(X_1, X_2):
    return X_1@X_2.T
```

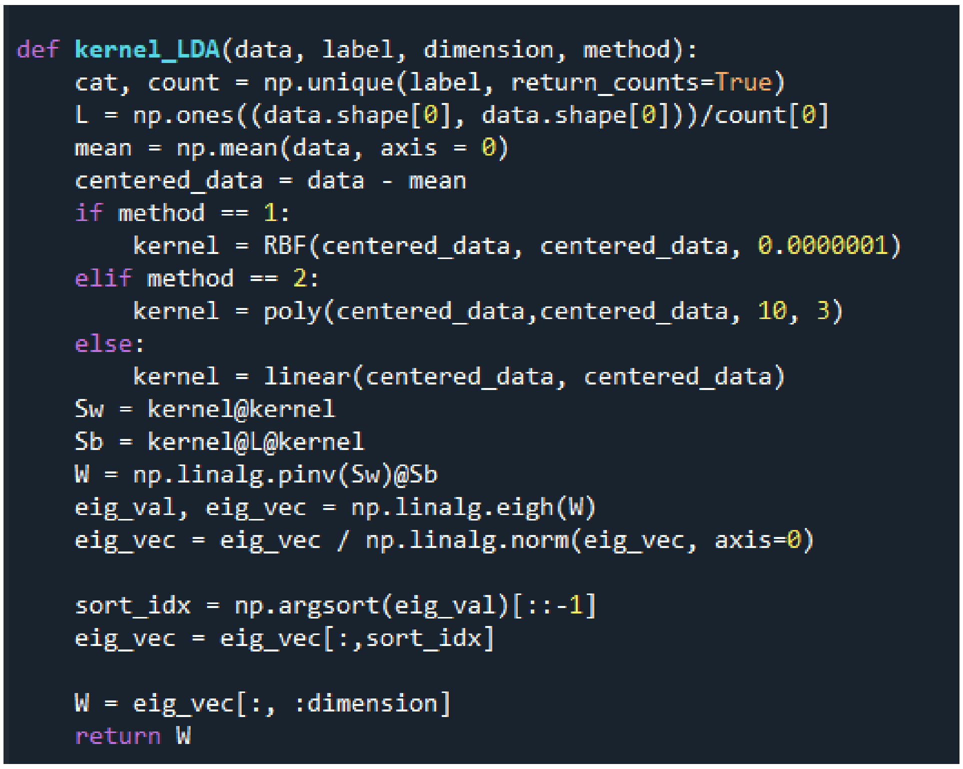
b. Kernel PCA



```python
def kernel_PCA(data, dimension, method):

    mean = np.mean(data, axis = 0)
    centered_data = data - mean
    if method == 1:
        kernel = RBF(centered_data, centered_data, 0.0000001)
    elif method == 2:
        kernel = poly(centered_data,centered_data, 10, 3)
    else:
        kernel = linear(centered_data, centered_data)

    eig_val, eig_vec = np.linalg.eigh(kernel)

    eig_vec = eig_vec / np.linalg.norm(eig_vec, axis=0)

    sort_idx = np.argsort(eig_val)[::-1]
    eig_vec = eig_vec[:,sort_idx]

    W = eig_vec[:, :dimension]

    return W
```

Again, we filrst centered the data and calculate the kernel. After getting the similarity matrix by using each kernel function, we follow the same structure to get our final weight W.

c. Kernel LDA



```python
def kernel_LDA(data, label, dimension, method):
    cat, count = np.unique(label, return_counts=True)
    L = np.ones((data.shape[0], data.shape[0]))/count[0]
    mean = np.mean(data, axis = 0)
    centered_data = data - mean
    if method == 1:
        kernel = RBF(centered_data, centered_data, 0.0000001)
    elif method == 2:
        kernel = poly(centered_data,centered_data, 10, 3)
    else:
        kernel = linear(centered_data, centered_data)
    Sw = kernel@kernel
    Sb = kernel@L@kernel
    W = np.linalg.pinv(Sw)@Sb
    eig_val, eig_vec = np.linalg.eigh(W)
    eig_vec = eig_vec / np.linalg.norm(eig_vec, axis=0)

    sort_idx = np.argsort(eig_val)[::-1]
    eig_vec = eig_vec[:,sort_idx]

    W = eig_vec[:, :dimension]
    return W
```

We first centered the data, and calculate each kernel or similarity matrix by using different kernel function. Next, I use the following formulas to simplify the calculation of kernel LDA. Again, after getting the eigenspaces, we use the same way to find our final weight matrix.

$$S_B^{\Phi} = \sum_{i=1}^{c} l_i \mu_i^{\Phi} (\mu_i^{\Phi})^T, \quad S_W^{\Phi} = \sum_{i=1}^{c} \sum_{j=1}^{l_i} \Phi(\mathbf{x}_{ij}) \Phi(\mathbf{x}_{ij})^T \qquad (12)$$

B. t-SNE

(1) Part 1

The difference between t-SNE and SSNE:

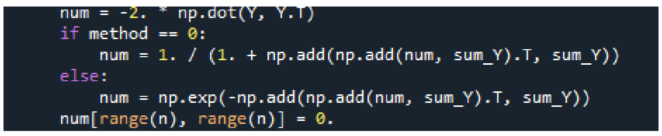
t-SNE q matrix in low-dimension compared to SSNE:

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

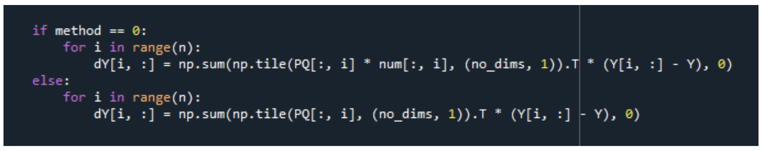
t- SNE gradient compared to SSNE:

$$C = KL(P||Q) = \sum_i \sum_{j \neq i} p_{ij} \log \frac{p_{ij}}{q_{ij}} \longrightarrow \frac{\partial C}{\partial y_i} = 2 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

Based n the observaton, we inly need to change two parts in the original source code:



```
    num = -2. * np.dot(Y, Y.T)
    if method == 0:
        num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
    else:
        num = np.exp(-np.add(np.add(num, sum_Y).T, sum_Y))
    num[range(n), range(n)] = 0.
```

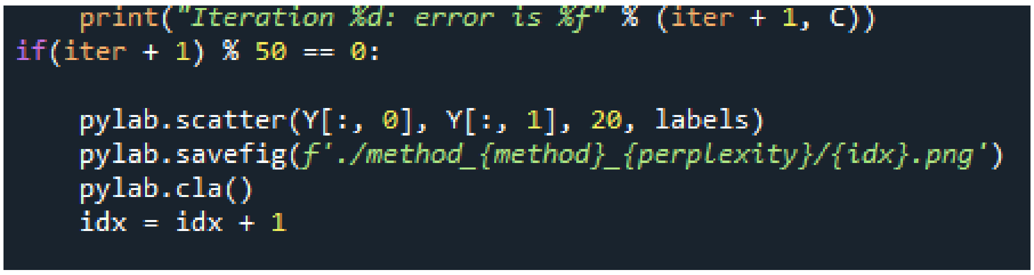


```
if method == 0:
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
else:
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

Where the method 0 is original t-SNE and 1 stands for SSNE.

(2) Part 2

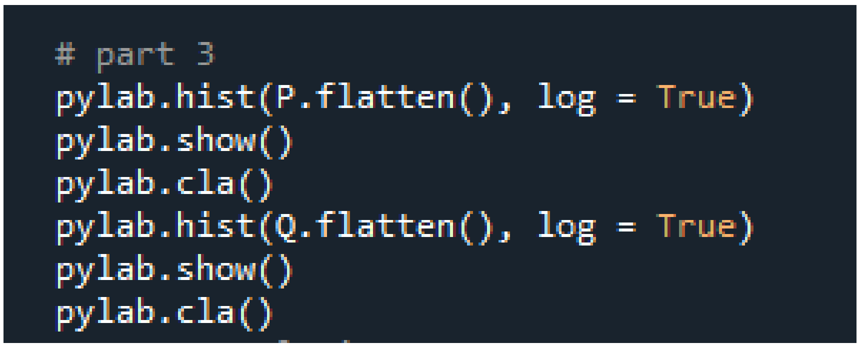
// The gif and each iteration file is in the zip file with the format method_{method}_{perplexity}, where method 0 is t-SNE, 1 is SSNE and perplexity is from 5, 10, 50, 100



```
    print("Iteration %d: error is %f" % (iter + 1, C))
if(iter + 1) % 50 == 0:

    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    pylab.savefig(f'./method_{method}_{perplexity}/{idx}.png')
    pylab.cla()
    idx = idx + 1
```

Each 50 epochs, record the output image and save for gif file.

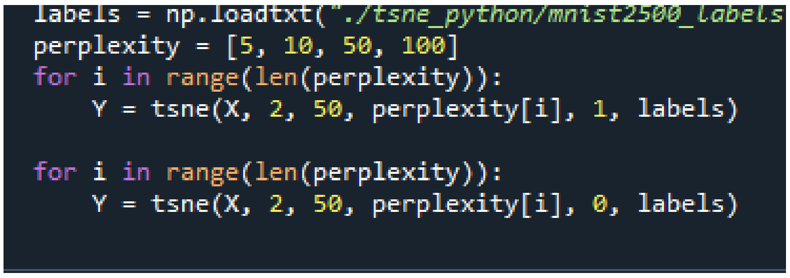
(3) Part 3



```
# part 3
pylab.hist(P.flatten(), log = True)
pylab.show()
pylab.cla()
pylab.hist(Q.flatten(), log = True)
pylab.show()
pylab.cla()
```

Show the pairwise similarities b/w high and low dimension using the histogram.

(4) Part 4



```
labels = np.loadtxt("./tsne_python/mnist2500_labels
perplexity = [5, 10, 50, 100]
for i in range(len(perplexity)):
    Y = tsne(X, 2, 50, perplexity[i], 1, labels)

for i in range(len(perplexity)):
    Y = tsne(X, 2, 50, perplexity[i], 0, labels)
```

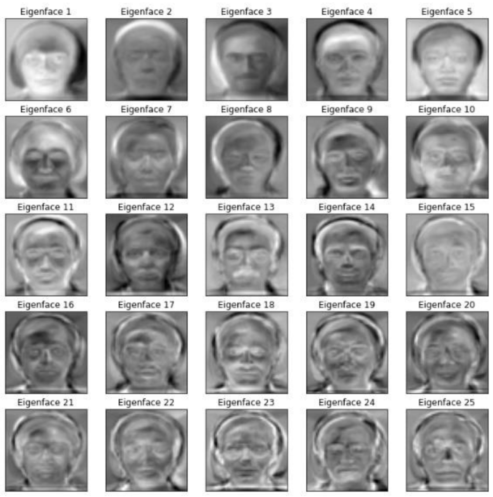
Using different perplexity values, here I try 4 different values.

2. Experiments and Discussion (50%)

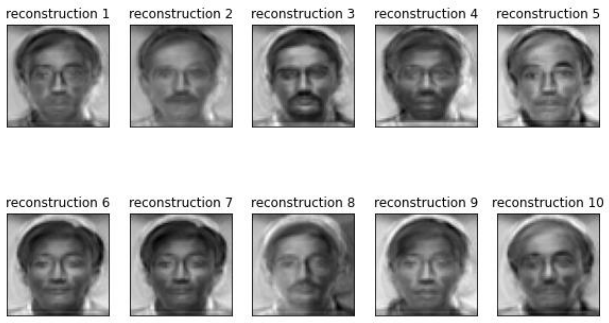
A. Kernel Eigenfaces

(1) Part 1:

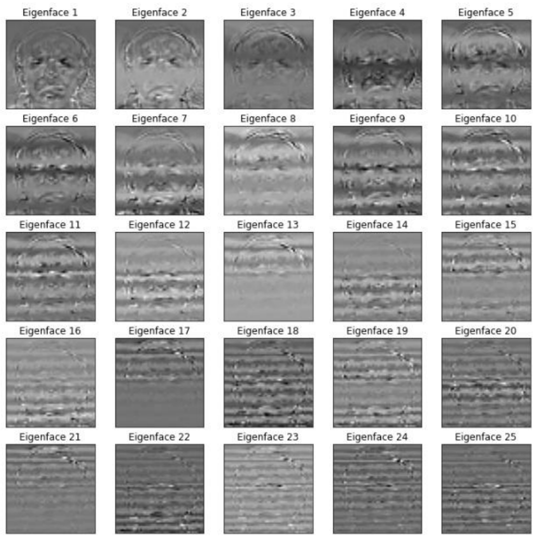
the eigenface (PCA):



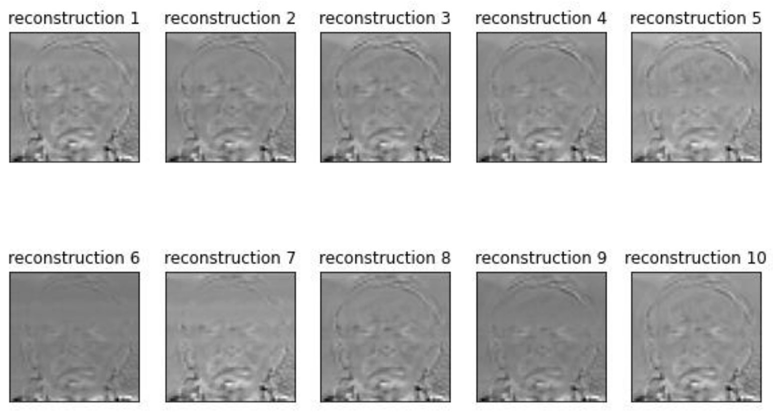
the reconstruction of eigenface (PCA):



the fisherface (LDA):



The reconstruction of fisherface (LDA):



It seems that the result of eigenfaces is more better than the fisherfaces

(2) Part 2

	Accuracy
PCA	86.67%
LDA	83.83%

This accuracy report aligns with the result in first part. The PCA in this testing scenario is better than the LDA algoriyhm.

(3) Part 3

	Accuracy
PCA + RBF	83.83%
PCA + Linear	76.67%
PCA + Polynomial	73.33%
LDA + RBF	80.00%
LDA + Linear	80.00%
LDA + Polynomial	73.33%

We can tell that the RBF kernel is the best, But with the kernel function, both algorithms perform worse compare to their original algorithm.

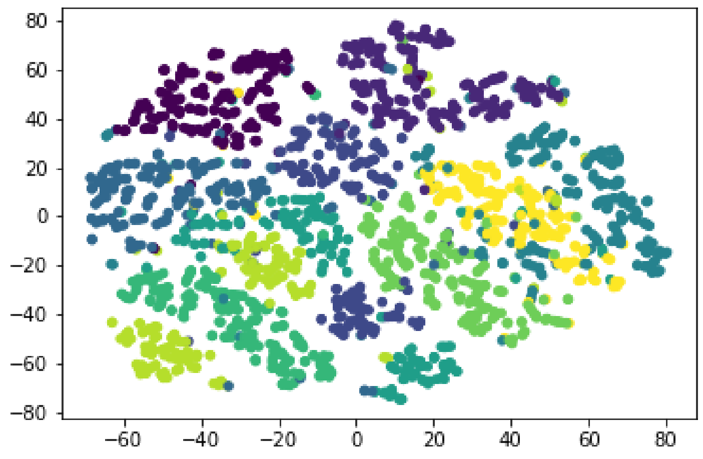
B. t-SNE

// The gif and each iteration file is in the zip file with the format method_{method}_{perplexity}, where method 0 is t-SNE, 1 is SSNE and perplexity is from 5, 10, 50, 100

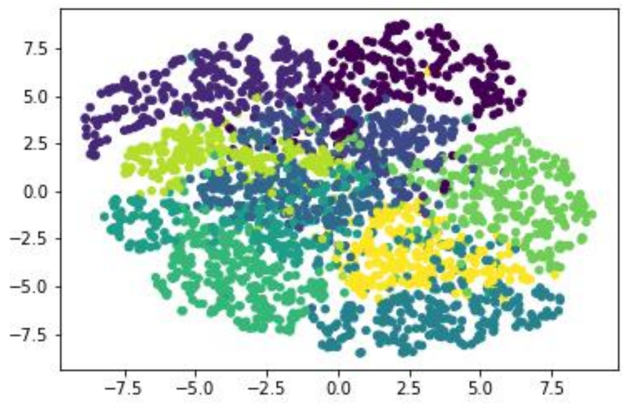
(1) Part 1

// only using 500 iterations to save more time to experiment with different settings. The result is quite similar after 500 iterations.

t-SNE:



SSNE

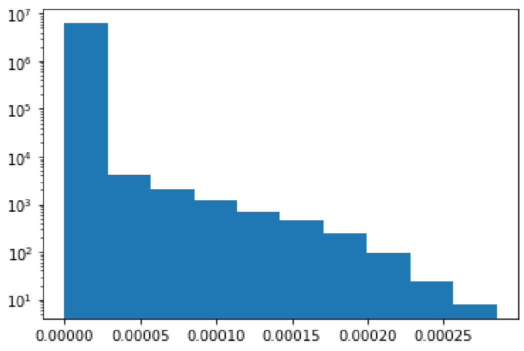
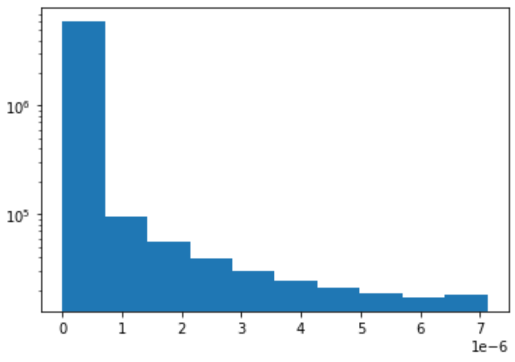
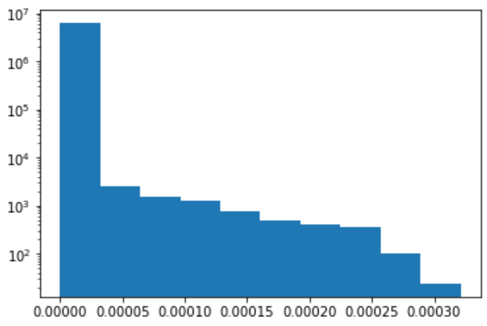
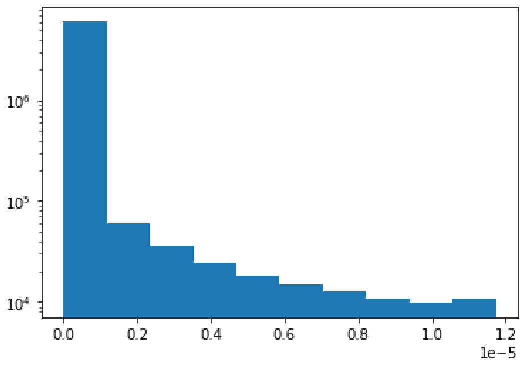


The crowding effect is quite significant, due to the apparent difference in the t-student distrivution and gaussain distribution.

(2) Part 2

The gif for different setting is in different directories, please refer to them.

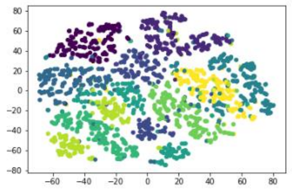
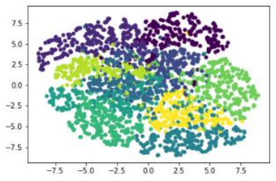
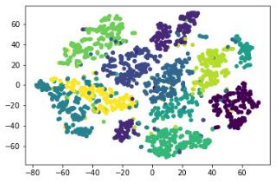
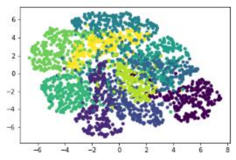
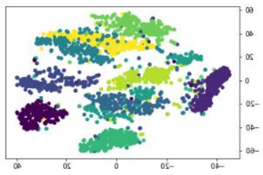
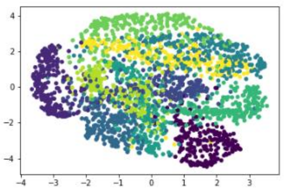
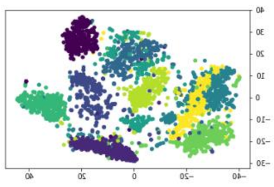
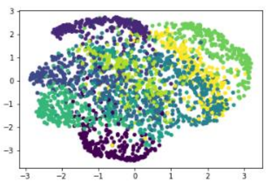
(3) Part 3

	High-d	Low-D
t-SNE		
SSNE		

We can tell that no matter in T-SNE or SSNE, the low-d graph shows more crowding or low space compared to the original highdimension. Also, It seems that point don't need to reach to far to get the small p in the T-SNE cases, whicj aligns to the concept between t-SNE and SSNE.

(4) Part 4

The visulaiztion is also in the specific directory, here just show some images in different scenarios.

Perplexity	t-SNE	SSNE
5		
10		
50		
100		

First, It seems that when the perplexity = 10 is the best setting for both t-SNE and SSNE, when perplexity is high, the result seems crowded in both cases. Also, SSNE is more crowded in each perplexity, which also aligns with our conclusion in above parts. Finally, the t-SNE seems have a greater ability in prediction.

3. Observations and Discussion

- The kernel PCA and kernel LDA have worse performance is quite surprised. I think the possible reason is because the kernel trick can't be utilized due to the dimension we have is already is high enough.
- The parameter in the kernel is quote important, especially in the RBF ones. The difference in parameters will lead to really different outcome. (aligns with previous hws).

- The t-SNE is quite powerful due to its non-crowded attributes. But here we only test 4 kinds of perplexity, so we can only guess that the optimal perplexity is around 10. Maybe there is a better seeting.
-