2022 Spring Machine Learning - Homework 7 (Kernel Eigenfaces and t-SNE)

Ta-Yang Lin(林大洋) 310554017

Code

Kernel Eigenfaces

Part 1

```
def load_data():
    Training_path = "./Yale_Face_Database/Training/"
    Testing_path = "./Yale_Face_Database/Testing/"

    test_file = glob.glob(Testing_path+"*.pgm")
    train_file = glob.glob(Training_path+"*.pgm")
    tmp = cv2.imread(train_file[0],0);
    training_data = np.zeros((len(train_file),RESIZE[1],RESIZE[0]))
    testing_data = np.zeros((len(test_file),RESIZE[1],RESIZE[0]))
    training_label = np.zeros(len(train_file))
    testing_label = np.zeros(len(test_file))

    for f in range(len(train_file)):
        tmp = cv2.imread(train_file[f],0);
        tmp = cv2.resize(tmp, (RESIZE[0], RESIZE[1]),
interpolation=cv2.INTER_AREA)
        training_data[f] = tmp
        training_label[f] = int(train_file[f].split("/")[-1].split(".")[0]
[7:9])
    for f in range(len(test_file)):
        tmp = cv2.imread(test_file[f],0);
        tmp = cv2.resize(tmp, (RESIZE[0],RESIZE[1]),
interpolation=cv2.INTER_AREA)
        testing_data[f] = tmp
        testing_label[f] = int(test_file[f].split("/")[-1].split(".")[0]
[7:9])

    training_data =
training_data.reshape(training_data.shape[0],training_data.shape[1]*trainin
g_data.shape[2])
    testing_data =
testing_data.reshape(testing_data.shape[0],testing_data.shape[1]*testing_da
ta.shape[2])

    return training_data,testing_data,training_label,testing_label

def show_eig_face(eig_vectors_n, algo):
```

```
    row = int(np.sqrt(N_component))
    fig, ax = plt.subplots(row,row)
    for i in range(eig_vectors_n.shape[1]):
        img = eig_vectors_n[:,i].reshape(RESIZE[0],RESIZE[1])
        ax[i//row][i%row].imshow(img, cmap='gray')
        ax[i//row][i%row].axis('off')

    plt.savefig(f'./eig_faces/{algo}.png')

def reconstruct(data,data_scaled,eig_vectors_n, algo, rand_idx ,mean,
train_label=0):
    random_data = data_scaled[rand_idx,:]
    res_images = np.dot(np.dot(random_data,eig_vectors_n),eig_vectors_n.T)
+ mean

    fig, ax = plt.subplots(2,5)
    for i in range(rand_idx.shape[0]):
        img = res_images[i,:].reshape(RESIZE[0],RESIZE[1])
        if(i<5):
            ax[0][i%5].imshow(img, cmap='gray')
            ax[0][i%5].axis('off')
        else:
            ax[1][i%5].imshow(img, cmap='gray')
            ax[1][i%5].axis('off')

    plt.savefig(f'./random_pick/{algo}.png')

    # original data
    random_data_ori = data[rand_idx,:]
    fig, ax = plt.subplots(2,5)
    for i in range(rand_idx.shape[0]):
        img = random_data_ori[i,:].reshape(RESIZE[0],RESIZE[1])
        if(i<5):
            ax[0][i%5].imshow(img, cmap='gray', vmin = 0, vmax =
255,interpolation='none')
            ax[0][i%5].axis('off')
        else:
            ax[1][i%5].imshow(img, cmap='gray', vmin = 0, vmax =
255,interpolation='none')
            ax[1][i%5].axis('off')

    plt.savefig(f'./random_pick/original.png')

def PCA(train_data, rand_idx):
    # calculate mean_face
    mean = train_data.mean(axis = 0)
    train_data_scaled = train_data - mean
    # covariance matrix
    cov_matrix = np.dot(train_data_scaled.T,train_data_scaled)
    # solve eigen_problem
    eig_values, eig_vectors = np.linalg.eigh(cov_matrix)
    # pick 25 eigen_vectors
    idx = eig_values.argsort()[::-1][:N_component]
    eig_vectors_n = eig_vectors[:,idx].astype(np.float64)
```

```
    show_eig_face(eig_vectors_n,"PCA")
    reconstruct(train_data,train_data_scaled,eig_vectors_n, "PCA", rand_idx
,mean)

    return eig_vectors_n

def LDA(train_data, train_label, rand_idx):
    #S_w and S_b
    label_mean =np.zeros((15,RESIZE[0]*RESIZE[1]))
    S_w = np.zeros((RESIZE[0]*RESIZE[1],RESIZE[0]*RESIZE[1]))
    S_b = np.zeros((RESIZE[0]*RESIZE[1],RESIZE[0]*RESIZE[1]))
    all_mean = np.mean(train_data,axis=0)

    for i in range(15):
        idx = np.where(train_label == i+1)[0]
        label_mean[i,:] = np.mean(train_data[idx,:],axis=0)
        wi = (train_data[idx,:]-label_mean[i,:])
        S_w += np.dot(wi.T,wi)

        bi = (label_mean[i,:] - all_mean).reshape(-1,1)
        S_b += len(idx) * np.dot(bi,bi.T)

    matrix = np.dot(np.linalg.pinv(S_w),S_b)
    #matrix = np.dot(S_w,S_b)

    eig_values, eig_vectors = np.linalg.eigh(matrix)
    idx = eig_values.argsort()[::-1][:N_component]
    eig_vectors_n = eig_vectors[:,idx].astype(np.float64)

    show_eig_face(eig_vectors_n,"LDA")
    train_data_scaled = train_data-all_mean
    reconstruct(train_data,train_data_scaled,eig_vectors_n, "LDA", rand_idx
,all_mean,train_label)

    return eig_vectors_n

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-q", default=1, type=int)
    args = parser.parse_args()

    RESIZE = [70,60]
    N_component = 25
    random_n = 10
    # (135, RESIZE[0]*RESIZE[1]), (30, RESIZE[0]*RESIZE[1])
    train_data, test_data,train_label,test_label = load_data()
    # random picking
    rand_idx = np.random.choice(test_data.shape[0],random_n,replace=False)
    #rand_idx = np.array([15,17,0,29,11,24,13,3,21,9])

    if(args.q == 1):
        eig_vectors_PCA = PCA(train_data, rand_idx)
        eig_vectors_LDA = LDA(train_data, train_label, rand_idx)
```

1. I call `load_data` that load the training data and testing data.**(resize 70*60)**
2. I pick 10 image randomly.
3. Implementation of PCA(`PCA`) -> get first 25 eigenvectors(eigenfaces)
    1. Calculate the mean of training data
    2. Training data minuses the mean -> `Training_data_scaled`
    3. Calculate the corvariance matrix of `Training_data_scaled`
    4. Solve eigen problem of corvariance matrix
    5. Sort eigenvalues
    6. Get first 25 eigenvalue corrsponding eigenvectors
4. Implementation of LDA(`LDA`) -> get first 25 eigenvectors(fisherfaces)
    1. Calculate `S_w`

$$S_w = \sum_{i=1}^{g}(N_i - 1)S_i = \sum_{i=1}^{g}\sum_{j=1}^{N_i}(x_{i,j} - \bar{x}_i)(x_{i,j} - \bar{x}_i)^T$$

    2. Calculate `S_b`

$$S_b = \sum_{i=1}^{g} N_i(\bar{x}_i - \bar{x})(\bar{x}_i - \bar{x})^T$$

    3. Solve eigen problem

$$S_W^{-1}S_B w = \lambda w$$

    4. Sort eigenvalues
    5. Get first 25 eigenvalue corrsponding eigenvectors
5. `show_eig_face` show the 25 eigenvectors as pictures
6. `reconstruct` reconstruct 10 pictures using `np.dot(np.dot(random_data,eig_vectors_n),eig_vectors_n.T) + mean`

Part 2

```
def KNN(train_X,train_y,test_X,test_y,type,k=3):
    prediction = np.zeros(test_X.shape[0])
```

```
    correct = 0
    for i in range(test_X.shape[0]):
        dis_matrix = distance.cdist(test_X[i].reshape(1,-1), train_X,
'euclidean').reshape(-1)
        idx = np.argsort(dis_matrix)[:k]
        prediction[i] =
np.argmax(np.bincount(train_y[idx].astype('int64')))
        if(prediction[i] == test_y[i]):
            correct+=1

    metrics(prediction,test_y)
    plt.savefig(f'./cm/cm-{type}.png')
    print(f'Accuracy = {correct/test_X.shape[0]}')

def metrics(prediction,test_y):
    res = np.zeros((15,15))
    for i in range(len(test_y)):
        res[int(test_y[i])-1,int(prediction[i])-1] +=1
    plot_confusion_matrix(res, [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
cmap="BuPu")

def plot_confusion_matrix(cm, classes, cmap):

    plt.figure(figsize = (10,10))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title("Confusion matrix")
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(int(cm[i, j]), 'd'),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.ylabel('True')
    plt.xlabel('Predicted')
```

1. Calculate the features of PCA and LDA

```
mean = np.mean(train_data,axis=0)

eig_vectors_PCA = PCA(train_data, rand_idx)
train_X_PCA = np.dot((train_data - mean),eig_vectors_PCA)
test_X_PCA = np.dot((test_data-mean),eig_vectors_PCA)

eig_vectors_LDA = LDA(train_data, train_label, rand_idx)
train_X_LDA = np.dot((train_data-mean),eig_vectors_LDA)
test_X_LDA = np.dot((test_data-mean),eig_vectors_LDA)
```

2. Run KNN using different k

```
k=[3,5,10,20]
for i in range(len(k)):
    print("PCA_",k[i])
    KNN(train_X_PCA,train_label,test_X_PCA,test_label,"PCA",k[i])
    print("LDA_",k[i])
    KNN(train_X_LDA,train_label,test_X_LDA,test_label,"LDA",k[i])
```

- `knn` is Implementation of KNN
    1. Calculate Euclidean distance between `test_X` and `train_X`
    2. Sort the distance from small to big
    3. Pick labels of first k `train_X`
    4. Prediction of `test_X` is mode of labels of first k `train_X`
    5. Calculate accuracy
    6. plot confusion matrix (`metrics` and `plot_confusion_matrix`)

Part 3

```
    if(args.q == 3):
        mean = np.mean(train_data,axis=0)
        train_y = train_label
        test_y = test_label
        kernel= ['sigmoid','rbf','poly']
        k=[3,5,10,20]
        for i in range(len(k)):
            for j in range(len(kernel)):
                eig_vectors_PCA = kernelPCA(train_data, rand_idx,kernel[j])
                train_X_PCA = np.dot((train_data-mean),eig_vectors_PCA)
                test_X_PCA = np.dot((test_data-mean),eig_vectors_PCA)
                eig_vectors_LDA = kernelLDA(train_data, train_label,
rand_idx, kernel[j])
                train_X_LDA = np.dot((train_data-mean),eig_vectors_LDA)
                test_X_LDA = np.dot((test_data-mean),eig_vectors_LDA)

KNN(train_X_PCA,train_y,test_X_PCA,test_y,f'Kernel_{kernel[j]}_PCA',k[i])

KNN(train_X_LDA,train_y,test_X_LDA,test_y,f'Kernel_{kernel[j]}_LDA',k[i])

def rbf(data_1,data_2,gamma=1e-5):
    ret = np.exp((-gamma**2) * distance.cdist(data_1,data_2,
'sqeuclidean'))
    return ret

def sigmoid(data_1,data_2,gamma=1e-5,coef=1e-5):
    ret = np.tanh(gamma * np.dot(data_1,data_2.T) + coef)
    return ret

def poly(data_1, data_2, gamma=1e-6, coef=1e-2, degree=2):
    ret = (gamma * np.dot(data_1,data_2.T) + coef) ** degree
    return ret
```

```
def kernelPCA(train_data, rand_idx , kernel="sigmoid"):
    # calculate mean_face
    mean = train_data.mean(axis = 0)
    train_data_scaled = train_data - mean

    if (kernel == "rbf"):
        gram_matrix = rbf(train_data.T,train_data.T)
    elif (kernel == "sigmoid"):
        gram_matrix = sigmoid(train_data.T,train_data.T)
    else:
        gram_matrix = poly(train_data.T,train_data.T)

    n_1 = np.ones((gram_matrix.shape[0],gram_matrix.shape[0])) *
(1/gram_matrix.shape[0])
    gram_matrix_ = gram_matrix - np.dot(n_1,gram_matrix) -
np.dot(gram_matrix,n_1) + np.dot(np.dot(n_1,gram_matrix),n_1)
    # solve eigen_problem
    eig_values, eig_vectors = np.linalg.eigh(gram_matrix_)
    # pick 25 eigen_vectors
    idx = eig_values.argsort()[::-1][:N_component]
    eig_vectors_n = eig_vectors[:,idx].astype(np.float64)

    show_eig_face(eig_vectors_n,f'Kernel_{kernel}_PCA')
    reconstruct(train_data,train_data_scaled,eig_vectors_n,
f'Kernel_{kernel}_PCA', rand_idx ,mean)

    return eig_vectors_n

def kernelLDA(train_data, train_label, rand_idx, kernel="sigmoid"):

    if (kernel == "rbf"):
        gram_matrix = rbf(train_data.T,train_data.T)
    elif (kernel == "sigmoid"):
        gram_matrix = sigmoid(train_data.T,train_data.T)
    else:
        gram_matrix = poly(train_data.T,train_data.T)

    #S_w and S_b
    label_mean =np.zeros((15,RESIZE[0]*RESIZE[1]))
    S_w = np.zeros((RESIZE[0]*RESIZE[1],RESIZE[0]*RESIZE[1]))
    S_b = np.zeros((RESIZE[0]*RESIZE[1],RESIZE[0]*RESIZE[1]))
    all_mean = np.mean(gram_matrix,axis=1)

    for i in range(15):
        idx = np.where(train_label == i+1)[0]
        t = np.identity(len(idx)) * (1/len(idx))
        S_w += np.dot(np.dot(gram_matrix[:,idx],t),gram_matrix[:,idx].T)

        label_mean[i,:] = np.mean(gram_matrix[idx,:],axis=0)
        bi = (label_mean[i,:] - all_mean).reshape(-1,1)
        S_b += len(idx) * np.dot(bi,bi.T)

    # pseudo-inverse
```

```
    matrix = np.dot(np.linalg.pinv(S_w),S_b)


    eig_values, eig_vectors = np.linalg.eigh(matrix)
    idx = eig_values.argsort()[::-1][:N_component]
    eig_vectors_n = eig_vectors[:,idx].astype(np.float64)

    show_eig_face(eig_vectors_n,f'Kernel_{kernel}_LDA')
    train_data_scaled = train_data-all_mean
    reconstruct(train_data,train_data_scaled,eig_vectors_n,
f'Kernel_{kernel}_LDA', rand_idx ,all_mean,train_label)

    return eig_vectors_n
```

1. Calculate the features of Kernel_PCA and Kernel_LDA
   - Kernel_PCA
     1. Calculate gram matrix(`poly`,`sigmoid`,`rbf`)
     2. Calculate `gram matrix_`(Kc)

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

     3. Solve eigen problem of `gram matrix_`
     4. Sort eigenvalues
     5. Get first 25 eigenvalue corrsponding eigenvectors
   - Kernel_LDA
     1. Calculate gram matrix(`poly`,`sigmoid`,`rbf`)
     2. Calculate `S_w` and `S_b`

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

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

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

$$S_W^{-1} S_B w = \lambda w$$

     3. Solve eigen problem
     4. Sort eigenvalues
     5. Get first 25 eigenvalue corrsponding eigenvectors
2. KNN prediction as **Part 2**

t-SNE

Part 1

```
def Hbeta(D=np.array([]), beta=1.0):
    """
        Compute the perplexity and the P-row for a specific value of the
        precision of a Gaussian distribution.
    """

    # Compute P-row and corresponding perplexity
    P = np.exp(-D.copy() * beta)
    sumP = sum(P)
    H = np.log(sumP) + beta * np.sum(D * P) / sumP
    P = P / sumP
    return H, P

def x2p(X=np.array([]), tol=1e-5, perplexity=30.0):
    """
        Performs a binary search to get P-values in such a way that each
        conditional Gaussian has the same perplexity.
    """

    # Initialize some variables
    print("Computing pairwise distances...")
    (n, d) = X.shape
    sum_X = np.sum(np.square(X), 1)
    D = np.add(np.add(-2 * np.dot(X, X.T), sum_X).T, sum_X)
    P = np.zeros((n, n))
    beta = np.ones((n, 1))
    logU = np.log(perplexity)

    # Loop over all datapoints
    for i in range(n):

        # Print progress
        if i % 500 == 0:
            print("Computing P-values for point %d of %d..." % (i, n))

        # Compute the Gaussian kernel and entropy for the current precision
        betamin = -np.inf
        betamax = np.inf
        Di = D[i, np.concatenate((np.r_[0:i], np.r_[i+1:n]))]
        (H, thisP) = Hbeta(Di, beta[i])

        # Evaluate whether the perplexity is within tolerance
        Hdiff = H - logU
        tries = 0
        while np.abs(Hdiff) > tol and tries < 50:

            # If not, increase or decrease precision
            if Hdiff > 0:
                betamin = beta[i].copy()
                if betamax == np.inf or betamax == -np.inf:
                    beta[i] = beta[i] * 2.
                else:
                    beta[i] = (beta[i] + betamax) / 2.
```

```
            else:
                betamax = beta[i].copy()
                if betamin == np.inf or betamin == -np.inf:
                    beta[i] = beta[i] / 2.
                else:
                    beta[i] = (beta[i] + betamin) / 2.

            # Recompute the values
            (H, thisP) = Hbeta(Di, beta[i])
            Hdiff = H - logU
            tries += 1

        # Set the final row of P
        P[i, np.concatenate((np.r_[0:i], np.r_[i+1:n]))] = thisP

    # Return final P-matrix
    print("Mean value of sigma: %f" % np.mean(np.sqrt(1 / beta)))
    return P

def tsne(X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0,
algo="t_SNE"):
    """
        Runs t-SNE on the dataset in the NxD array X to reduce its
        dimensionality to no_dims dimensions. The syntaxis of the function
is
        `Y = tsne.tsne(X, no_dims, perplexity), where X is an NxD NumPy
array.
    """

    # Check inputs
    if isinstance(no_dims, float):
        print("Error: array X should have type float.")
        return -1
    if round(no_dims) != no_dims:
        print("Error: number of dimensions should be an integer.")
        return -1

    # Initialize variables
    X = pca(X, initial_dims).real
    (n, d) = X.shape
    max_iter = 300
    initial_momentum = 0.5
    final_momentum = 0.8
    eta = 500
    min_gain = 0.01
    Y = np.random.randn(n, no_dims)
    dY = np.zeros((n, no_dims))
    iY = np.zeros((n, no_dims))
    gains = np.ones((n, no_dims))

    Y_history = np.zeros((max_iter,n, no_dims))

    # Compute P-values
    P = x2p(X, 1e-5, perplexity)
```

```
    P = P + np.transpose(P)
    P = P / np.sum(P)
    P = P * 4.                                    # early exaggeration
    P = np.maximum(P, 1e-12)

    # Run iterations
    for iter in range(max_iter):

        # Compute pairwise affinities
        sum_Y = np.sum(np.square(Y), 1)
        num = -2. * np.dot(Y, Y.T)

        # different Q
        if(algo == "t_SNE"):
            num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
        elif (algo =="s_SNE"):
            num = np.exp(-(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 (different gradient)
        PQ = P - Q
        for i in range(n):
            if(algo == "t_SNE"):
                dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims,
1)).T * (Y[i, :] - Y), 0)
            elif (algo =="s_SNE"):
                dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i,
:] - Y), 0)

        # Perform the update
        if iter < 20:
            momentum = initial_momentum
        else:
            momentum = final_momentum
        gains = (gains + 0.2) * ((dY > 0.) != (iY > 0.)) + \
                (gains * 0.8) * ((dY > 0.) == (iY > 0.))
        gains[gains < min_gain] = min_gain
        iY = momentum * iY - eta * (gains * dY)
        Y = Y + iY
        Y = Y - np.tile(np.mean(Y, 0), (n, 1))

        # 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))

        # Stop lying about P-values
        if iter == 100:
            P = P / 4.
        Y_history[iter,:,:] = Y
```

```
    # Return solution
    return Y_history, P, Q
```

In symmetric SNE

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

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

$$\frac{\partial C}{\partial y_i} = 2\sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

In t-SNE

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

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

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

The differences between SSNE and t-SNE are `qij` and the gradient. In low-dimension, t-SNE alleviates crowding problem using T-distribution, so `qij` is different with SSNE, t-SNE mainly preserves local similarity structure of data. Because `qij` is changed, so the gradient is changed as well.

```
# different qij
        if(algo == "t_SNE"):
            num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
        elif (algo =="s_SNE"):
            num = np.exp(-(1. + np.add(np.add(num, sum_Y).T, sum_Y)))
```

According to formula of SSNE, I changed above part that means `qij`

```
# Compute gradient (different gradient)
        PQ = P - Q
        for i in range(n):
            if(algo == "t_SNE"):
                dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims,
1)).T * (Y[i, :] - Y), 0)
            elif (algo =="s_SNE"):
                dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i,
:] - Y), 0)
```

Above part means gradient, so I changed them.

Part 2

```
def visualize(Y_history, labels, perplexity, algo):

    if not os.path.isdir(f'./SNE/{algo}_{perplexity}/'):
        os.mkdir(f'./SNE/{algo}_{perplexity}/')
        os.mkdir(f'./SNE/{algo}_{perplexity}/gif/')
        os.mkdir(f'./SNE/{algo}_{perplexity}/dis/')

    cmap =
['red','orange','blue','gray','purple','yellow','green','pink','lightskyblu
e','springgreen']
    label_unique = np.unique(labels)


    for i in range(Y_history.shape[0]):
        plt.figure(figsize = (10,10))
        for j in range(len(label_unique)):
            plt.title(f'{algo}-{perplexity}_{i}')
            idx = np.where(labels == label_unique[j])[0]
            plt.scatter(Y_history[i,idx,0],Y_history[i,idx,1],color =
cmap[j],label= j)
            plt.legend()

        plt.savefig(f'./SNE/{algo}_{perplexity}/{algo}-
{perplexity}_{i}.png')

    # plot gif
    imgs = []
    for i in range(Y_history.shape[0]):
        temp = Image.open(f'./SNE/{algo}_{perplexity}/{algo}-
{perplexity}_{i}.png')
        imgs.append(temp)
    save_name = f'./SNE/{algo}_{perplexity}/gif/{algo}_{perplexity}.gif'
    imgs[0].save(save_name, save_all=True, append_images=imgs, duration=10)
```

`visualize` can visualize the embedding results, and make `.gif`. `Y_history` records the result of each iteration, so I can visualize them, observes the difference of each iteration

Part 3

```
def plot_dis(P, Q, perplexity, algo):
    P = P.reshape(-1)
    Q = Q.reshape(-1)
    min_P_idx = np.where(P == np.min(P))[0]
    min_Q_idx = np.where(P == np.min(Q))[0]
    P = np.delete(P,min_P_idx)
    Q = np.delete(Q,min_Q_idx)
    log_P = np.log(P)
    log_Q = np.log(Q)

    plt.figure(figsize = (10,20))
    fig, ax = plt.subplots(2)

    ax[0].hist(log_P, bins=200)
    ax[0].set_title(f'{algo}-{perplexity}_high-dimensional space')
    ax[1].hist(log_Q, bins=200)
    ax[1].set_title(f'{algo}-{perplexity}_low-dimensional space')

    plt.subplots_adjust(hspace=1)
    plt.savefig(f'./SNE/{algo}_{perplexity}/dis/{algo}-
{perplexity}_dis.png')
```

When I plot the distribution, I found the values of P and Q are too small, for observing easily, I use `np.log` on these data. Another problem is that there are a lot of minimum values, it will affect the distribution, so I delete them for observing easily `plot_dis` plot the distributions of high-dimension and low-dimension.

Part 4

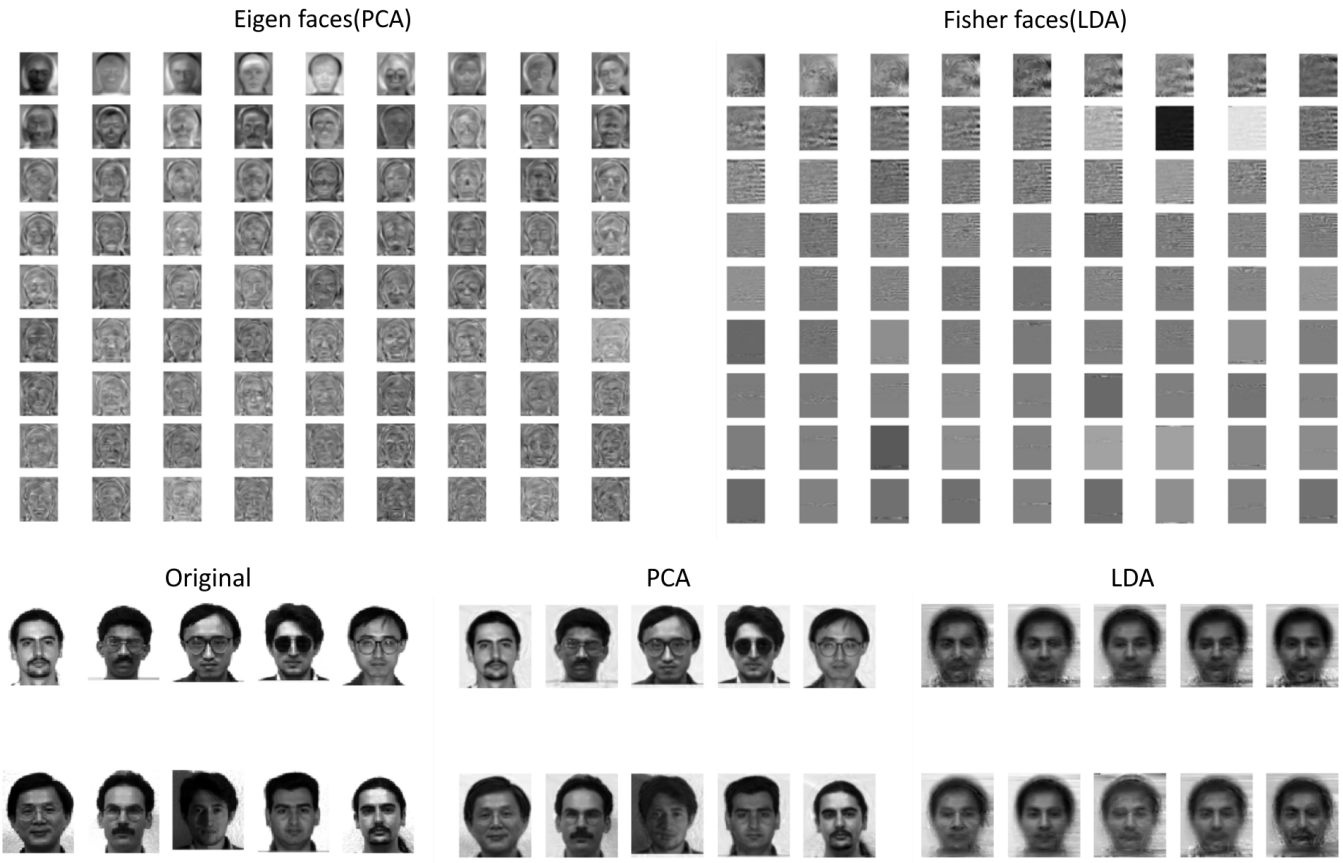
```
if __name__ == "__main__":
    X,labels = load_data()
    perplexity = [5,15,30,50]
    algo = ["t_SNE","s_SNE"]
    for i in range(len(perplexity)):
        for j in range(len(algo)):
            print(f'{algo[j]}_{perplexity[i]}')
            Y_history, P, Q = tsne(X, 2, 50, perplexity[i], algo[j])
            visualize(Y_history, labels, perplexity[i], algo[j])
            plot_dis(P, Q, perplexity[i], algo[j])
            plt.close('all')
```

In Part 4, I need to try different perplexities in t-SNE and SSNE, so I set a perplexity list `[5,15,30,50]`, I try the effects using them.

Experiments Results & Discussion

Kernel Eigenfaces

Part 1



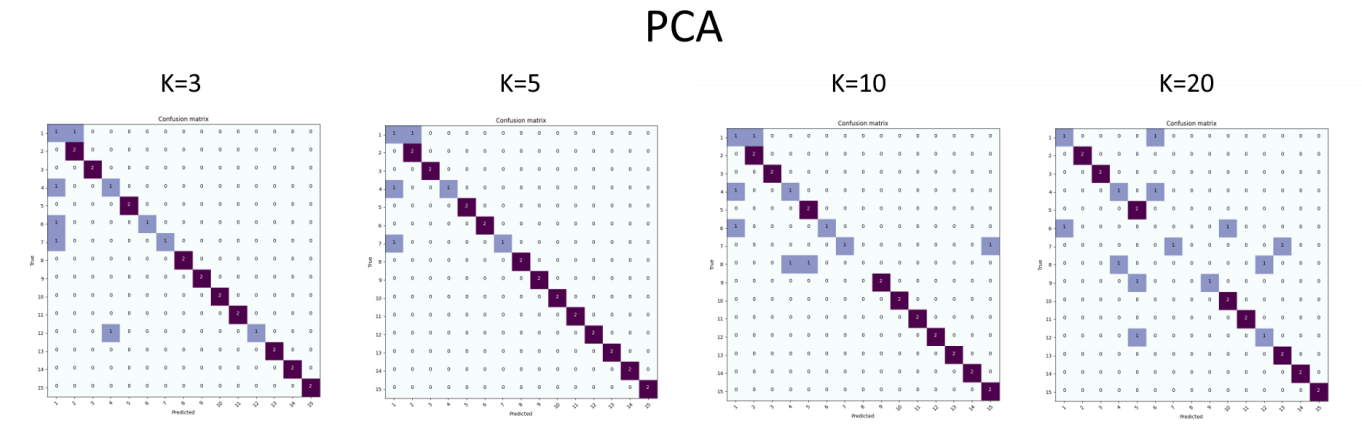
1. PCA got better eigenfaces than LDA, I found some fisher faces aren't human faces, they like noises.
2. I try to reconstruct face picture, I found that PCA has good performance, but LDA can't reconstruct the correct faces, because fisher face is not good.
3. If I need to reconstruct faces, I will use PCA.

Part 2

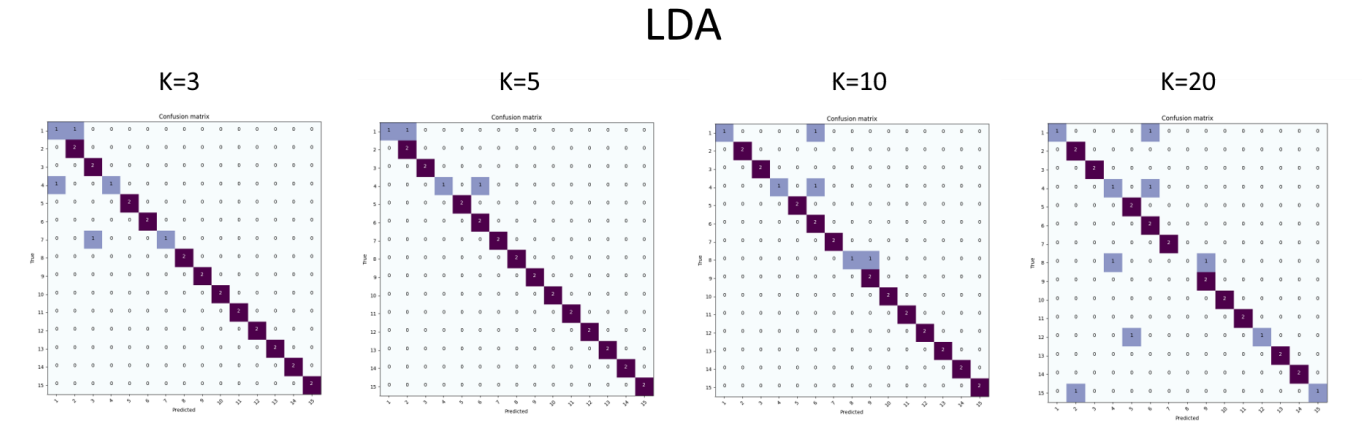
Algorithm	k	Accuracy
PCA	3	0.8333
PCA	5	0.9000
PCA	10	0.8000
PCA	20	0.7000
LDA	3	0.9000
LDA	5	0.9333

Algorithm	k	Accuracy
LDA	10	0.9000
LDA	20	0.8000

PCA confusion matrix



LDA confusion matrix



1. I try to predict with different k value, I found performances of LDA are better than PCA.
2. k=5, performances of PCA and LDA are best than other k values.
3. k=20, performances of PCA and LDA are worst than other k values.
4. subject 01 is hard to predict for LDA and PCA.

Part 3

Algorithm	k	kernel	Accuracy
PCA	3		0.8333
PCA	5		0.9000
PCA	10		0.8000
PCA	20		0.7000
LDA	3		0.9000

Algorithm	k	kernel	Accuracy
LDA	5		**0.9333**
LDA	10		0.9000
LDA	20		0.8000
KernelPCA	3	rbf	0.8666
KernelPCA	5	rbf	0.8666
KernelPCA	10	rbf	**0.9333**
KernelPCA	20	rbf	0.8333
KernelPCA	3	poly	0.8666
KernelPCA	5	poly	0.8666
KernelPCA	10	poly	0.8666
KernelPCA	20	poly	0.8333
KernelPCA	3	sigmoid	0.8000
KernelPCA	5	sigmoid	0.8333
KernelPCA	10	sigmoid	0.7666
KernelPCA	20	sigmoid	0.7000
KernelLDA	3	rbf	0.8333
KernelLDA	5	rbf	0.8333
KernelLDA	10	rbf	0.8000
KernelLDA	20	rbf	0.6666
KernelLDA	3	poly	0.7666
KernelLDA	5	poly	0.7666
KernelLDA	10	poly	0.7000
KernelLDA	20	poly	0.6333
KernelLDA	3	sigmoid	0.7666
KernelLDA	5	sigmoid	0.7333
KernelLDA	10	sigmoid	0.7333
KernelLDA	20	sigmoid	0.6333

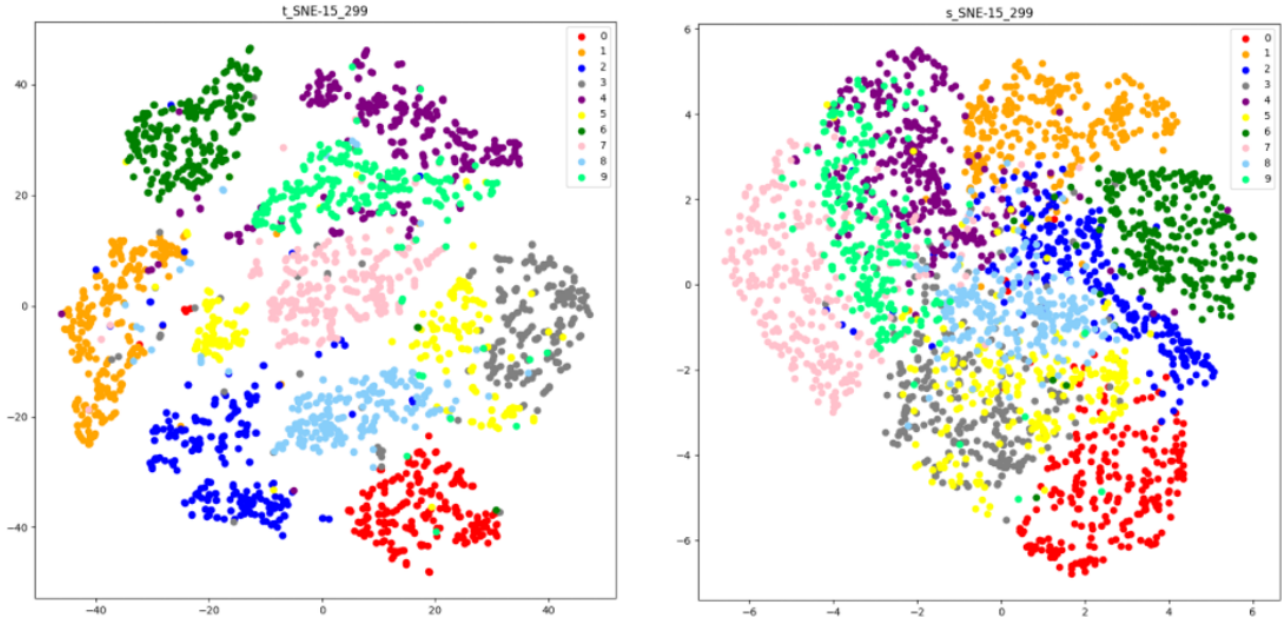
1. Kernel PCA(rbf) got 93.3% accuracy with k=10, which is the same as LDA.
2. I think Kerenl LDA is not suitable for predition because the accuracies are worse than LDA.
3. In Kernel PCA, rbf is best kernel, sigmoid is not suitable for this case, Kernel PCA(rbf) is better than PCA averagely.

4. Averagely, I think LDA is best method for prediction in this case.
5. K=20 usually got worse performance in each algorithm, I think k=5 or 10 ca n get good performances.

t-SNE

Part 1

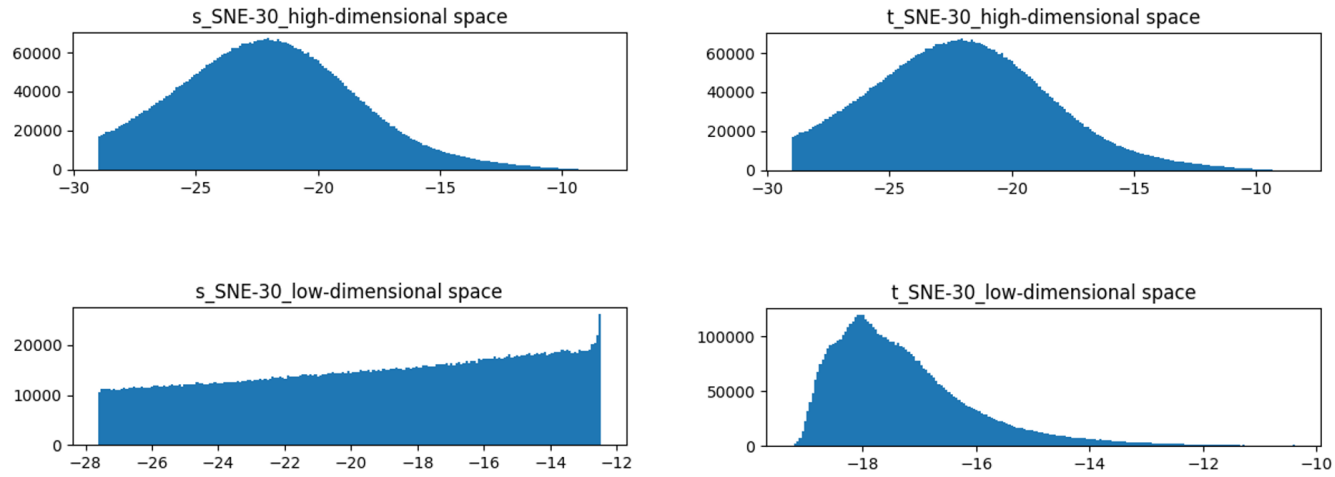
In Symmetric SNE, crowding problem will affect results in low-dimension, so t-SNE uses t-distribution to compute the similarity between two points in the low-dimensional space rather than a Gaussian distribution, which helps resolve crowding problem. Another advantage of t-distribution is it can preserve the local structure of the data. I try to visualize 2-dimensional data of t-SNE and Symmetric SNE, I found clusters of t-SNE are more scattered than Symmetric SNE, which means t-SNE resolves crowding problem.



Part 2

ALL `.gif` can be found in `gif_folder`, According to these results, I found t-SNE divides each cluster in fewer iterations, Symmetric SNE can't divide well.

Part 3

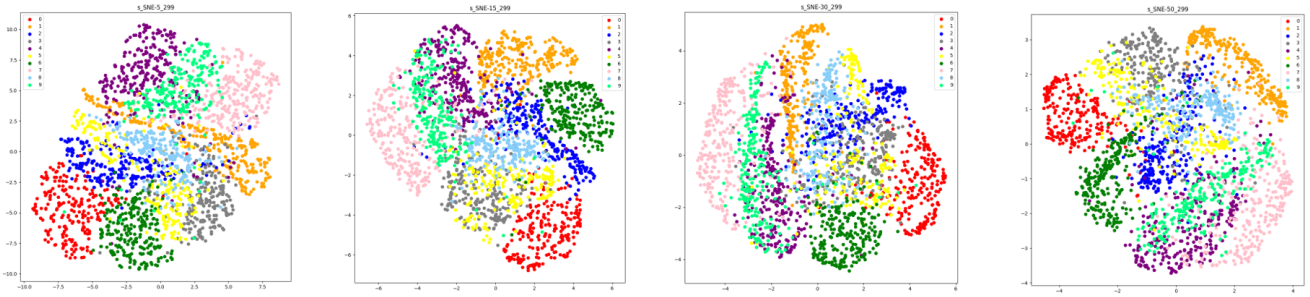


The distribution of high-dimension is a gaussion distribution in t-SNE and Symmetric SNE. About distribution of low-dimension, In t-SNE, it is a t-distribution that has long tail as we known. In Symmetric SNE, I think it more like uniform distribution, more high value has more high probability.

Part 4

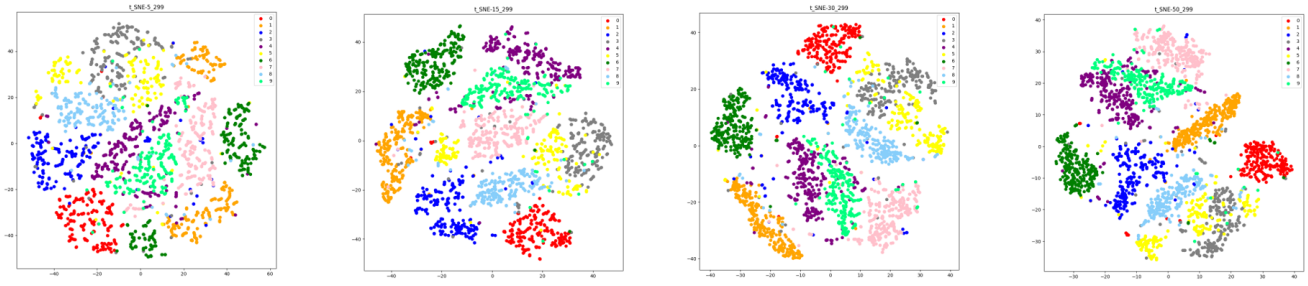
Symmetric SNE (iteration=300)

Perplexity :5 Perplexity :15 Perplexity :30 Perplexity :50



t-SNE(iteration=300)

Perplexity :5 Perplexity :15 Perplexity :30 Perplexity :50



Obviously, t-SNE results are better than Symmetric SNE's. With different perplexity

- I think results of Symmetric SNE aren't too different, if I haven't the labels, it's hard to know each point belongs which number or cluster.
- I think results of t-SNE are good
- high perplexities are better than low perplexities in t-SNE, I think the clusters are more scattered in 50 perplexity.
- I found effect of some numbers aren't quite good, 5(yellow) has two clusters, 4(purple) and 9(light green) overlaped, maybe their pictures are similar.

Observations and Discussion

Kernel Eigenfaces

- In each kernel, they have some hyperparameters, maybe try to optimize them, it can get better performance.
- I have tried to predict by different facial expressions,but can't get good performance, so I think prediction of facial expressions is harder than prediction of subject.
- For face reconstruction, PCA is better, but for prediction of subject, LDA is better, it's interesting, which means the two algorithm got very different features, We need to consider different algorithms in different datasets.

t-SNE

- I think `iteration` is important in t-SNE, maybe it can use some condiction to stop because I found the some results are good in early interation
- In t-SNE and Symmetric SNE, it will do PCA, I set the number of principal components = 50, but I think this is a hyperparameter, in research, we can consider optimize it.
- I think t-SNE's disadvantage is that need to take much time, so if I want to don't use PCA and directly use t-SNE, too high-dimension will take a lot of time in t-SNE.