ML Homework7 0856136 鍾昀諠

File `part1.py` includes PCA, Kernel PCA, LDA and Kernel LDA while file `part2.py` is a modified version inherited from the link given in spec.

I will explain my implementation of each algorithm in the following sections.

Part 1

First I define some global hyper parameters in main function.

Then I implement some shared functions such as **kernel functions**, **plotting function** and **input function**.

Refers to comments in code section below for detailed information.

```
def linear_kernel(x, y):
    k = list()
    for img in x:
        k.append(np.dot(img, y.T))
    k = np.array(k)

    return k


def rbf_kernel(x, y):
    k = cdist(x, y, metric='sqeuclidean')
    k = np.exp(-gamma*k)

    return k


def fetch_eigvecs(eig_val, eig_vec, k, kernel=None):
"""
return first k biggest eigenvectors.
eigenvectors are normalized by their corresponding eigenvalues if parameter
kernel is specified.
"""
    eig_dict = dict(zip(eig_val, eig_vec.T))
    eig_val[::-1].sort()
    principle_component = eig_dict[eig_val[0]]
    for i in range(1, k):
        if kernel:
            principle_component = np.vstack((principle_component,
eig_dict[eig_val[i]] / eig_val[i]))
        else:
            principle_component = np.vstack((principle_component,
eig_dict[eig_val[i]]))

    return principle_component.T


def plot_face(axis_x, axis_y, eigen_face, img_name, scale):
    # eigen_face dimension of (K components, scale.flatten).
```

```
    # note here each row in eigen_face represents a face.
    eig_face, row = list(), list()
    for i in range(axis_y):
        for j in range(axis_x):
            eig_face.append(eigen_face[i * axis_x + j, :].reshape(scale))
        row.append(np.hstack(eig_face)) # extend 3D 5 eigenfaces to 2D array
        eig_face = list()
    eig_faces = np.vstack(row) #extend 3D to 2D

    plt.imshow(eig_faces, cmap='gray')
    plt.imsave(img_name, eig_faces, cmap='gray')

    return


def read_images(path, scale=(195, 231)):
    dataset = list()
    for image in os.listdir(path):
        with Image.open(os.path.join(path, image)) as img:
            img = img.resize(scale, Image.ANTIALIAS)
            dataset.append(np.array(img).flatten())

    # returns a np array with shape (num_imgs)X(num_pixels_per_image)
    return np.array(dataset)


if __name__ == "__main__":
    train_path = os.path.join('Yale_Face_Database', 'Training')
    test_path = os.path.join('Yale_Face_Database', 'Testing')

    k = 25 # num dimensions after dimension reduction.
    reconst = np.random.randint(135, size=10) # 10 images selected for
reconstruction.
    gamma = 5e-9 # used in rbf kernel
    kernel_funcs = {'linear': linear_kernel, 'rbf': rbf_kernel} # dictionary for
different kernel function.
```

PCA

Function **PCA** implements PCA on input **X** to reduce it's dimension to **res_dim**.

Function **predict_PCA** transforms testing data to low dimension as **img_lowD** and performs KNN with K=**n_neighbors** with the training set on low dimension.

In main function I first get **eigen faces** and **average face** in low Dimension by PCA.

Then reconstruct the randomly picked images by eigenfaces, finally predict the subjects of testing data and calculate the accuracy.

```
def PCA(X, res_dim):
    print ('Find PC from input array')
    (m, dim) = X.shape

    avg_face = np.mean(X, axis=0)
```

```
    X = X - np.tile(avg_face, (m, 1))
    cov = np.dot(X, X.T)
    eig_val, eig_vec = np.linalg.eig(cov)
    eig_vec /= np.linalg.norm(eig_vec, axis=0)

    principle_component = fetch_eigvecs(eig_val, eig_vec.real, res_dim)
    print (principle_component.shape)

    Y = np.dot(X.T, principle_component).astype('float32')
    Y /= np.linalg.norm(Y, axis=0)
    print (Y.shape)

    return Y, avg_face


def predict_PCA(train_data, test_data, avg_face, train_W, n_neighbors):
    prd_result = list()
    diff_train = train_data - avg_face
    diff_train /= np.linalg.norm(diff_train, axis=1)[:, None]

    for image in test_data:
        diff = image - avg_face
        diff /= np.linalg.norm(diff)

        img_lowD = np.dot(train_W.T, diff)
        dist = list()
        for train_img in diff_train:
            train_lowD = np.dot(train_W.T, train_img)
            dist.append(np.linalg.norm(train_lowD - img_lowD))
        dist = np.array(dist)
        idx = np.argpartition(dist, n_neighbors)[:n_neighbors] // 9 + 1
        prd_result.append(np.argmax(np.bincount(idx)))

    return np.array(prd_result)


if __name__ == "__main__":
    # task1 & task2 of PCA
    train_imgs = read_images(train_path, (195, 231)) # note PIL scale order
    train_W, avg = PCA(train_imgs, k)
    print ('eigen face shape:' , train_W.shape)
    print ('PCA done')
    plot_face(5, 5, train_W.T, 'PCA_EigenFace.png', (231, 195))

    print ('Reconstructing random 10 images')
    print (reconst)
    plot_face(5, 2, train_imgs[reconst], 'OriginFace.png', (231, 195))
    res_mean = np.mean(train_imgs[reconst], axis=0)
    plot_face(5, 2, np.dot(train_imgs[reconst] - avg, np.dot(train_W,
train_W.T)) + avg, 'PCA_Resconstruct.png', (231, 195))

    print ('Predicting test images')
    test_imgs = read_images(test_path)
    test_labels = sorted([i for i in range(1, 16)]*2)

    prd_result = predict_PCA(train_imgs, test_imgs, avg, train_W, 3)
    print ('Accuracy of PCA: ', len(prd_result[prd_result == test_labels]) / 30)
```

Figure 1 shows the eigen faces. This image is save as **PCA_EigenFace.png** in the zip file.

Figure 2 shows the randomly picked originally data, and saved as **OriginFace.png**

Figure 3 shows the reconstructed images, saved as **PCA_Reconstruce.png**

In KNN with K = 3, PCA gets a 86.7% accuracy

```
Accuracy of PCA:  0.8666666666666667
```

---

kernel PCA

Function **kernel PCA** not only implements kernel PCA but also includes prediction.

Different from **PCA**, here **fetch_eigvecs** will return a (135, 25) matrix representing the training sets in low D instead of eigenfaces in PCA, with which causes KPCA to has **different implementation to transform testing data** from PCA and thus can't be applied to **predict_PCA**

```
def KPCA(X, test, res_dim, kernel, n_neighbors):
    # X row-based
    k = kernel_funcs[kernel](X, X)
    print ('kerenl shape:', k.shape)

    N = len(X)
    ones = np.ones((N, N)) / N
    k = k - np.dot(ones, k) - np.dot(k, ones) + np.dot(ones, np.dot(k, ones))

    eig_val, eig_vec = np.linalg.eig(k)
    alpha = fetch_eigvecs(eig_val, eig_vec, res_dim, kernel=1) # 135X25
    print (alpha.shape)
    train_img_lowD = np.dot(k, alpha)

    prd_result = list()
    for img in test:
        img = img.reshape(1, -1)

        img_lowD = kernel_funcs[kernel](X, img).flatten()
        img_lowD = np.sum((alpha.T*img_lowD).T, axis=0)

        distance = list()
        for img in train_img_lowD:
            distance.append(np.linalg.norm(img - img_lowD))
        distance = np.array(distance)
        # print (distance)
```

```
        idx = np.argpartition(distance, n_neighbors)[:n_neighbors] // 9 + 1
        prd_result.append(np.argmax(np.bincount(idx)))

    return np.array(prd_result)

if __name__ == "__main__":
    prd_linear = KPCA(train_imgs, test_imgs, k, 'linear', 3)
    prd_rbf = KPCA(train_imgs, test_imgs, k, 'rbf', 3)
    print (prd_linear)
    print ('Accuracy of KPCA_rbf: ', len(prd_rbf[prd_rbf == test_labels]) / 30)
    print ('Accuracy of KPCA_linear: ', len(prd_linear[prd_linear ==
test_labels]) / 30)
```

In KNN with k = 3, rbf kernel gets 70% accuracy while linear kernel gets only 6.7% accuracy. No idea why linear gets such a poor performance.

```
Accuracy of KPCA_rbf:  0.7
Accuracy of KPCA_linear:  0.06666666666666667
```

LDA

In LDA I resize the image size to 100X100 to avoid memory allocation error since **SW** and **SB** would be of size (45045, 45045) if no compression on images is made.

Function **LDA** implements the LDA algorithm and returns the fisherface **W**, while **predict_LDA** implements the KNN facial recognition based on **W**.

```
def LDA(train_imgs, k):
    print (train_imgs.shape)
    start = dt.now()
    center_set = list()
    within_scatter, between_scatter = 0, 0
    print ('Start calculate SW...')
    for i in range(15):
        # calculate Sk and sum up all Sks
        data_i = train_imgs[9 * i : 9 * (i + 1), :]
        center = np.mean(data_i, axis=0)
        center_set.append(center)

        for data in data_i:
            diff = (data - center).reshape((-1, 1)).astype('float32')
            within_scatter += np.dot(diff, diff.T)

    print (np.linalg.det(within_scatter))
    center_set = np.array(center_set)
    print (center_set.shape)
    print ('Start calculate SB...')
    center = np.mean(train_imgs, axis=0)
    for center_i in center_set:
        diff = (center_i - center).reshape((-1, 1)).astype('float32')
        between_scatter += 9 * np.dot(diff, diff.T)

    print ((dt.now() - start).total_seconds())
    print ('Start calculate W...')
    eig_val, eig_vec = np.linalg.eig((np.linalg.pinv(within_scatter) *
between_scatter))
```

```
    W = fetch_eigvecs(eig_val, eig_vec.real, k)
    plot_face(5, 5, W.T, 'LDA_FisherFace.png', (100, 100))

    return W


def predict_LDA(train_imgs, test_imgs, W, n_neighbors):
    prd_result = list()
    print ('W shape', W.shape)
    # print ('train shape', train_imgs_lowD.shape)

    for img in test_imgs:
        img_lowD = np.dot(img, W)

        distance = list()
        for train_img in train_imgs:
            train_img_lowD = np.dot(train_img, W)
            distance.append(np.linalg.norm(train_img_lowD - img_lowD))
        distance = np.array(distance)
        idx = np.argpartition(distance, n_neighbors)[:n_neighbors] // 9 + 1
        prd_result.append(np.argmax(np.bincount(idx)))

    return np.array(prd_result)


if __name__ == "__main__":
    print ('Start LDA')
    start = dt.now()
    train_imgs = read_images(train_path, (100, 100))
    test_imgs = read_images(test_path, (100, 100))
    W = LDA(train_imgs, k)
    print ((dt.now() - start).total_seconds())

    print ('Reconstructing random 10 images')
    plot_face(5, 2, np.dot(np.dot(train_imgs[reconst], W), W.T),
'LDA_Reconstruct.png', (100, 100))


    print ('Predicting test images')
    prd_result = predict_LDA(train_imgs,test_imgs, W, 3)
    print (prd_result)
    print ('Accuracy of LDA: ', len(prd_result[prd_result == test_labels]) / 30)
```

Figure 4 shows the first 25 fisher faces, and saved as **LDA_FisherFace.png**, but the faces are not obvious.

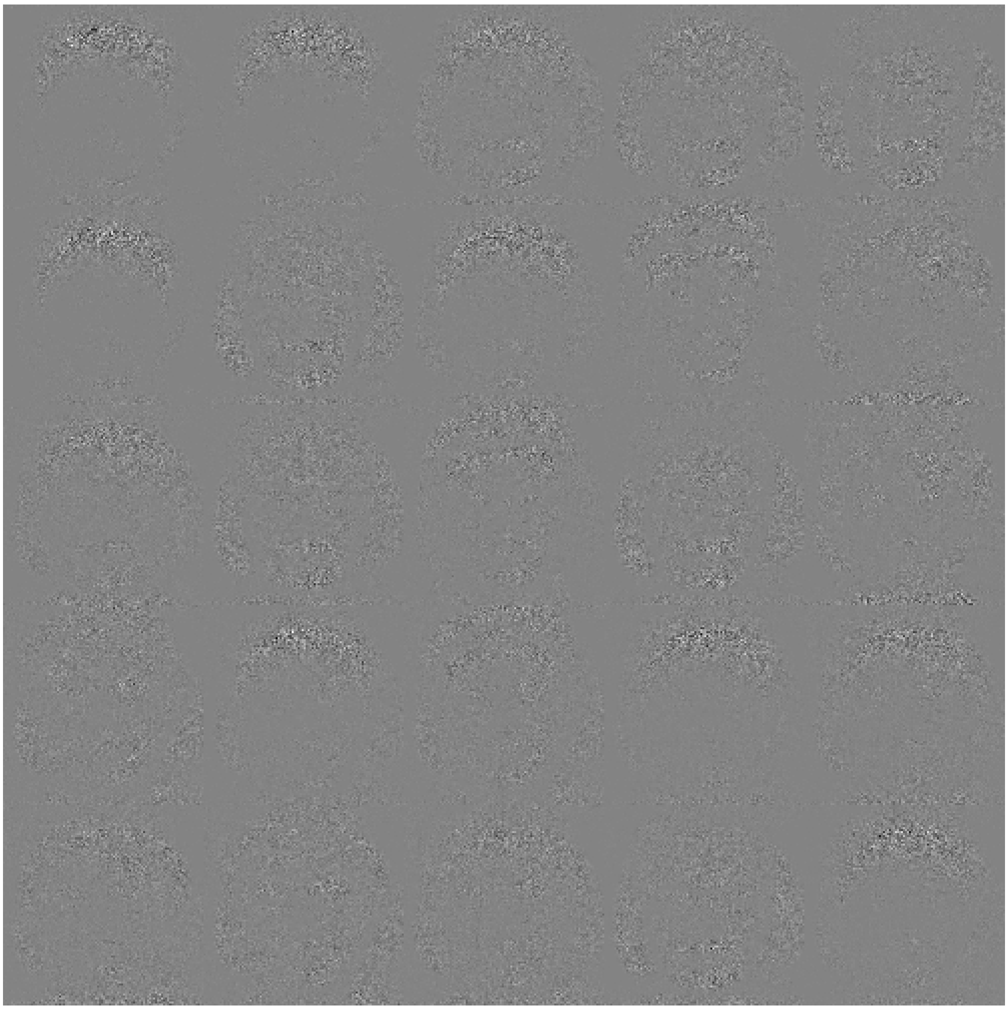
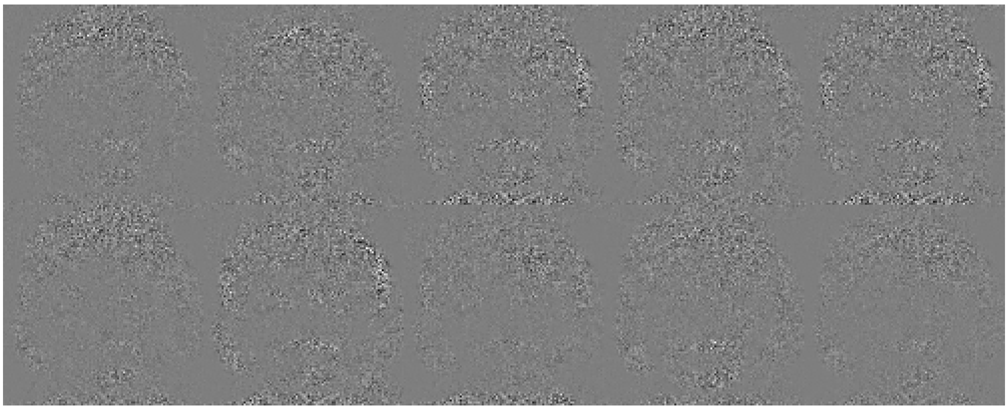


Figure 5 shows the reconstructed images, and saved as **LDA_Reconstruct.png**, but the faces are not obvious.



Predicted Label and the accuracy of LDA prediction

```
[ 1  1  2  2  3  3  4  7  5  5  1  6  3  7  8  4  9  9 10 10  6 11 12 12
 13 13 14 14 15 15]
Accuracy of LDA:  0.8333333333333334
```

---

kernel LDA

In kernel LDA, datum are first transforms to high dimension then the 2 scatter factor **SW** and **SB** are calculated to get W.

So the **origin LDA can be viewed as Kernel LDA with linear kernel**.

The only modification I made here is to change scatter factor **SW** and **SB** from linear kernel to a designated kernel function(line 17 & 26 here).

And since there are no difference in predicting images, **predict_LDA** can be reused to perform prediction of kernel LDA.

```
def KLDA(train_imgs, k, kernel):
    print (train_imgs.shape) # should be 135 X 10000
    start = dt.now()
    center_set = list()
    within_scatter, between_scatter = 0, 0
    print ('Start calculate SW...')
    for i in range(15):
        # calculate Sk and sum up all Sks
        print (i)
        data_i = train_imgs[9 * i : 9 * (i + 1), :]
        center = np.mean(data_i, axis=0)
        center_set.append(center)

        for data in data_i:
            diff = (data - center).reshape((-1, 1)).astype('float32')
            within_scatter += kernel_funcs[kernel](diff, diff)

    print (np.linalg.det(within_scatter))
    center_set = np.array(center_set)
    print (center_set.shape)
    print ('Start calculate SB...')
    center = np.mean(train_imgs, axis=0)
    for center_i in center_set:
        diff = (center_i - center).reshape((-1, 1)).astype('float32')
        between_scatter += 9 * kernel_funcs[kernel](diff, diff)

    print ((dt.now() - start).total_seconds())
    print ('Start calculate W...')
    eig_val, eig_vec = np.linalg.eig((np.linalg.pinv(within_scatter) *
between_scatter))
    W = fetch_eigvecs(eig_val, eig_vec.real, k)
    # plot_face(5, 5, W.T, 'KLDA_FisherFace.png', (100, 100))

    return W


if __name__ == "__main__":
    W = KLDA(train_imgs, k, 'rbf')
    prd_result = predict_LDA(train_imgs,test_imgs, W, 3)
    print (prd_result)
    print ('Accuracy of RBF KLDA: ', len(prd_result[prd_result == test_labels])
/ 30)

    W = KLDA(train_imgs, k, 'linear')
    prd_result = predict_LDA(train_imgs,test_imgs, W, 3)
    print (prd_result)
    print ('Accuracy of Linear KLDA: ', len(prd_result[prd_result ==
test_labels]) / 30)
```

Predicted Label and the accuracy of RBF KLDA prediction

```
[ 1  1  2  2  3  3  4 15  5  5  6  6  7  7  8  8  9  9 10 10 11 11 13 12
 13  7 14 14 15 14]
Accuracy of RBF KLDA:  0.8666666666666667
```

Predicted Label and the accuracy of Linear KLDA prediction, which is consistent with that of LDA.

Note that I forget to change the print message so it's still RBF in the image, but actually it's linear.

```
[ 1  1  2  2  3  3  4  7  5  5  1  6  3  7  8  4  9  9 10 10  6 11 12 12
 13 13 14 14 15 15]
Accuracy of RBF KLDA:  0.8333333333333334
```

Part 2

Here I will just explain my modification

```
def func(X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0,
method='tsne'): # modified: new paras method to indicate whether sym-sne or 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 = 600
    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))

    # 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):
        if iter % 20 == 0:
```

```
            pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
            # pylab.show()
            pylab.savefig(f'{method}_{perplexity}_{iter}.png')
        # Compute pairwise affinities
        sum_Y = np.sum(np.square(Y), 1)
        num = -2. * np.dot(Y, Y.T)
        if method == 'tsne':
            num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y)) # t-dist
        else:
            num = np.exp(-1 * np.add(np.add(num, sum_Y).T, sum_Y)) # Gaussian-
dist

        num[range(n), range(n)] = 0. # set diagnoal element to 0
        Q = num / np.sum(num)
        Q = np.maximum(Q, 1e-12)



        # Compute gradient
        PQ = P - Q
        if method == 'tsne':
            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) # Gaussian gradient

        # 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.
    # Return solution
    return Y
```

1. Add a parameter **method** to specify whether to use t-SNE or symmetric-SNE
2. Save current distribution for every 10 iterations in Line 38~41
3. Compute corresponding affinities in Line 45~48
4. Compute corresponding gradients in Line 57~62

If the perplexity is getting higher, the visualized neighboring data points will getting closer to each other faster.