Homework 4

Collaborators:
Name: Jiang Shibiao
Student ID: 3170102587

Problem 4-1. Spectral Clustering

In this problem, we will try a dimensionality reduction based clustering algorithm Spectral Clustering.

(a) We will first experiment Spectral Clustering on synthesis data

Answer:

Implementing Spectral Clustering using Python is easy.

```
n = W.shape[0]
D = np.zeros((n, n))
for i in range(n):
    D[i][i] = np.sum(W[i])
eigen, vec = np.linalg.eig(D-W)
choose = vec[:, np.argsort(eigen)[:k]]
return kmeans(choose, k)
```

Figure 1: The code for Spectral Clustering

And I found that the hyper-parameters for building graph are **not sensitive**.

```
k_in_knn_graph = 50
threshold = 0.5

# implement knn_graph in knn_graph.py
from knn_graph import knn_graph
W = knn_graph(X, k_in_knn_graph, threshold)

# implement spectral in spectral
from spectral import spectral

idx = spectral(W, 2)
cluster_plot(X, idx)
```

Figure 2: The settings for Spectral Clustering

And the results for Spectral Clustering and Kmeans are as follows.

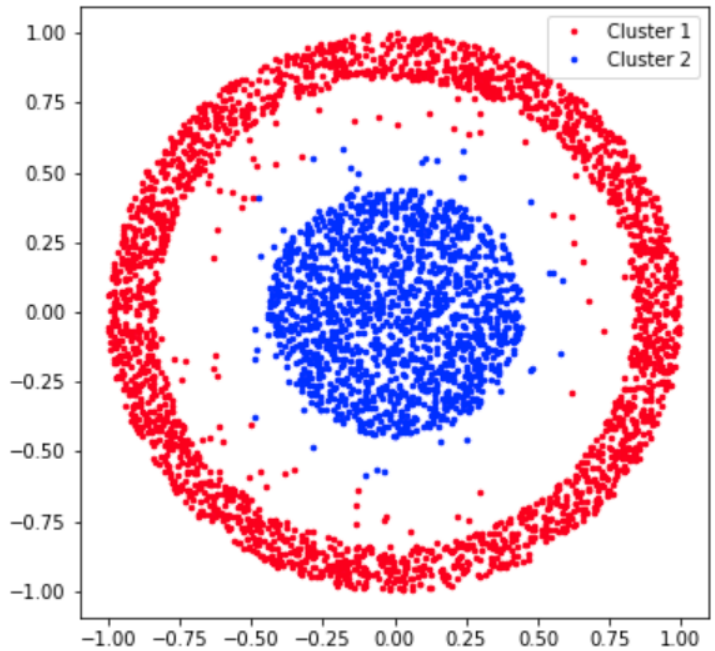


Figure 3: The result for Spectral Clustering

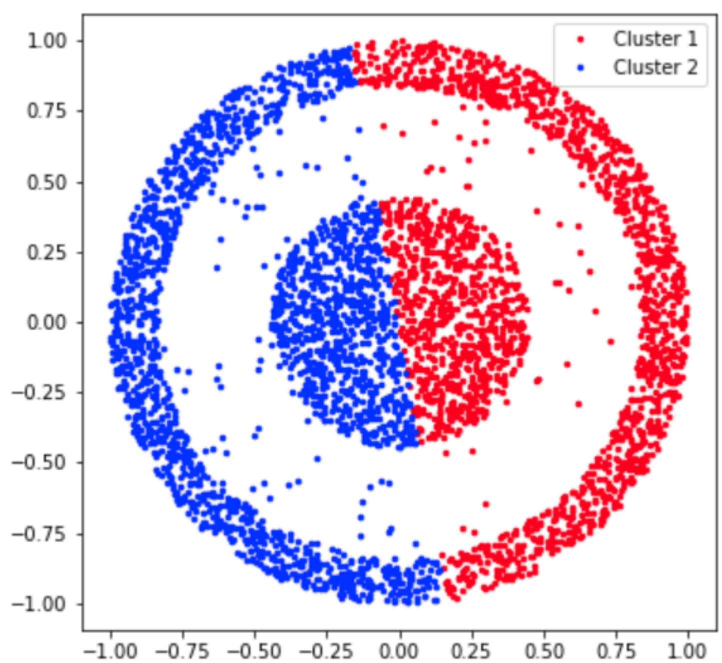


Figure 4: The result for Kmeans Clustering

(b) Now let us try Spectral Clustering on real-world data.

Answer:
constructW.py is used to build graph for Spectral Clustering.
bestMap.py is used to match two two clustering results.
MutualInfo.py is used to calculate the nmi for two clustering results.
To check the influence of different graph types, I tried two types for Spectral Clustering *Binary* and *HeatKernel*. And the expriment repeats 100 times to increase stability.

```
for t in range(testCase):
    print ("Step", t, "Start!")
    kmean_ans = bestMap(gnd, kmeans(fea, K))
    kmean_acc += np.sum(kmean_ans == gnd) / gnd.shape[0]
    kmean_nmi += MutualInfo(gnd, kmean_ans)
    options = {'NeighborMode': 'KNN'}
    options['k'] = 5
    options['WeightMode'] = 'HeatKernel'

    W = np.array(constructW(fea, **options).todense())
    spectral_binary_ans = bestMap(gnd, spectral(W, K))
    spectral_binary_acc += np.sum(spectral_binary_ans == gnd) / gnd.shape[0]
    spectral_binary_nmi += MutualInfo(gnd, spectral_binary_ans)

    options = {'NeighborMode': 'KNN'}
    options['k'] = 5
    options['WeightMode'] = 'Binary'

    W = np.array(constructW(fea, **options).todense())
    spectral_kernel_ans = bestMap(gnd, spectral(W, K))
    spectral_kernel_acc += np.sum(spectral_kernel_ans == gnd) / gnd.shape[0]
    spectral_kernel_nmi += MutualInfo(gnd, spectral_kernel_ans)
```

Figure 5: The code for real-word clustering

From the following results we can find:

- Spectral Clustering is **better** than (direct) Kmeans Clustering.
- Different types of graph influence Spectral Clustering **slightly**.

```
kmeans accuracy: 0.5142077331311604
kmeans normalized mutual information: 0.33509074327507066
spectral(binary) accuracy: 0.7161334344200148
spectral(binary) normalized mutual information: 0.6096607693414727
spectral(kernel) accuracy: 0.7113343442001514
spectral(kernel) normalized mutual information: 0.6082433917330253
```

Figure 6: The result for real-word clustering

Problem 4-2. Principal Component Analysis Let us deepen our understanding of PCA by the following problems.

(a) Your task is to implement *hack_pca.m* to recover the rotated CAPTCHA image using PCA.

Answer: The code for **PCA.py** is quite simple.

```
D, N = data.shape
X = normal(data)
S = np.matmul(X, X.T) / N
eigen_val, eigen_vec = np.linalg.eig(S)
idx = np.argsort(eigen_val)[::-1]
eigen_val = eigen_val[idx]
eigen_vec = eigen_vec[:, idx]
return eigen_vec, eigen_val
```

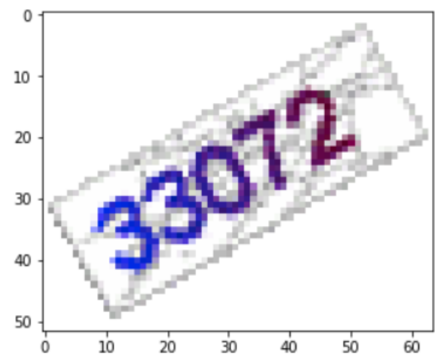
Figure 7: The code for OCA

In **hack_ pca.py**, we should find each non-empty position for the current picture. Then we call **PCA** to find the rotation matrix and make a mapping from the original picture to the new picture. Note the order in the picture is different from that in numpy.

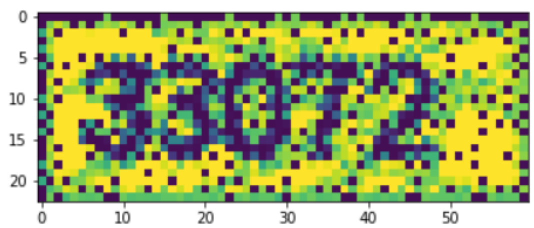
```
img_r = (plt.imread(filename)).astype(np.float64) # 4 channels: R,G,B,A
img_gray = img_r[:,:,0] * 0.3 + img_r[:,:,1] * 0.59 + img_r[:,:,2] * 0.11

X_int = np.array(np.where(img_gray > 0))
X = X_int.astype(np.float64)
D, N = X.shape

eigen_vec, eigen_val = PCA(X)
print (eigen_vec, eigen_val)
Y = np.matmul(X.T, eigen_vec).T

Y_int = Y.astype(np.int32)
dmin = np.min(Y_int, axis = 1).reshape(D, 1)
Y_int = Y_int - dmin
bound = np.max(Y_int, axis = 1) + 1
new_img = np.zeros(bound)
for t in range(Y_int.shape[1]):
    new_img[tuple(Y_int[:, t])] = img_gray[tuple(X_int[:, t])]
new_img = new_img.T[::-1, ::-1]

return new_img
```

Figure 8: The code for OCA

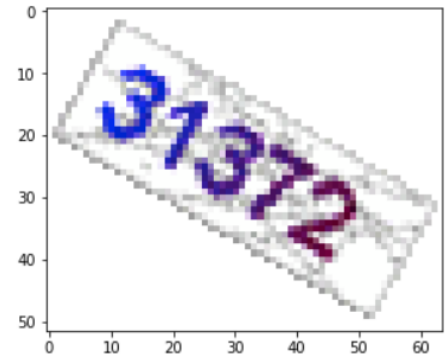
The direction of eigenvector **can be flipped**, so the result picture may be **upside down**. (I debug it for a long time and finally find out it can not be avoided.)



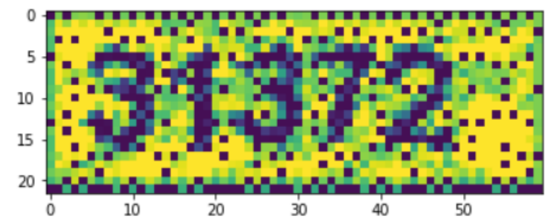
```
[[ 108.61524523 -115.54792126]
 [-115.54792126  243.26640419]]
[[ 0.4982774  -0.86701767]
 [-0.86701767 -0.4982774 ]] [309.67210661   42.20954281]

<function matplotlib.pyplot.show(*args, **kw)>
```



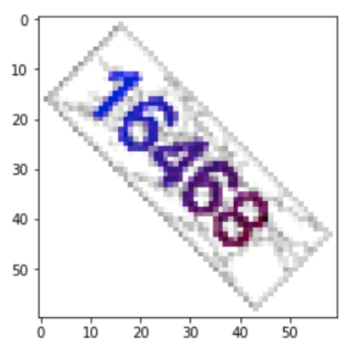
(a) 1.gif



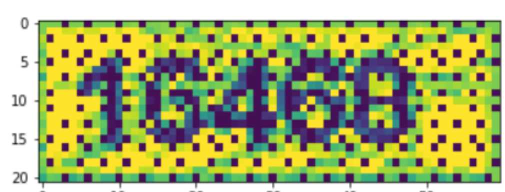
```
[[108.46280937 115.39412243]
 [115.39412243 241.93947993]]
[[-0.49967503 -0.86621295]
 [-0.86621295  0.49967503]] [308.5045913   41.897698 ]

<function matplotlib.pyplot.show(*args, **kw)>
```



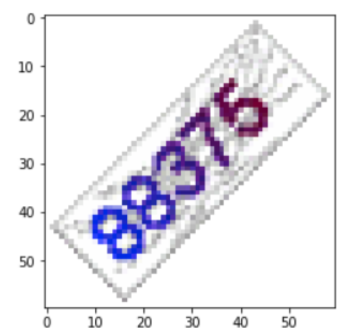
(b) 2.gif



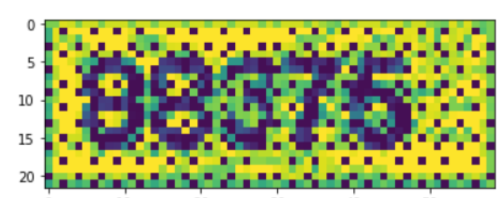
```
[[177.16339233 135.3372086 ]
 [135.3372086  177.42622634]]
[[-0.70676339 -0.70745001]
 [-0.70745001  0.70676339]] [312.63208174  41.95753693]

<function matplotlib.pyplot.show(*args, **kw)>
```



(c) 3.gif



```
[[ 175.49458031 -133.40097667]
 [-133.40097667  175.57951524]]
[[ 0.70699422 -0.70721932]
 [-0.70721932 -0.70699422]] [308.9380312   42.13606435]

<function matplotlib.pyplot.show(*args, **kw)>
```



(d) 4.gif

Figure 9: The results for rotation

(b) Now let us apply PCA to a face image dataset.

Answer:

1. After **Feature preprocessing** (Normalization), I apply PCA to all faces. If consider each eigenvector as a face, we can get their visualizations.

```
from pca import PCA
eigen_vec, eigen_val = PCA(fea_Train.T)
eigen_vec = np.real(eigen_vec)
eigen_val = np.real(eigen_val)
# end answer
from show_face import show_face
show_face(eigen_vec.T)
```

Figure 10: The visualization for eigenvectors

2. Then I iterate K from 8 to 128 to do dimension reductions and reconstructions.

```
from knn import knn

k_list = [8, 16, 32, 64, 128]
for k in k_list:
    # 4. Project data on to low dimensional space
    # begin answer
    compress_Train = np.matmul(fea_Train, eigen_vec[:, :k])
    compress_Test = np.matmul(fea_Test, eigen_vec[:, :k])
    # end answer
    # 5. Run KNN in low dimensional space
    # begin answer
    pred_Test = knn(compress_Test, compress_Train, gnd_Train, 1)
    error_rate = 1.0 - np.sum(pred_Test == gnd_Test) / pred_Test.shape[0]
    print ("Error rate for k = %2d:" % k, error_rate)
    # end answer
    # 6. Recover face images form low dimensional space, visualize them
    # begin answer
    rebuild = np.matmul(compress_Train, eigen_vec[:, :k].T)
    show_face(rebuild)
    # end answer
```

Figure 11: The code for dimension reductions and reconstructions

I find that the best hyper-parameter for knn is 1. And under this parameter, the error rate for knn is $[26\%, 18.5\%, 14.5\%, 12\%, 12.5\%]$, where K is $[8, 16, 32, 64, 128]$.

I also try **without normalization** and the error rate is $[24.5\%, 20\%, 18\%, 15\%, 15\%]$. It means that feature preprocessing will make the answer more steady.

From the following figures, we can find dimensionality reduction causes loss of information but not much.

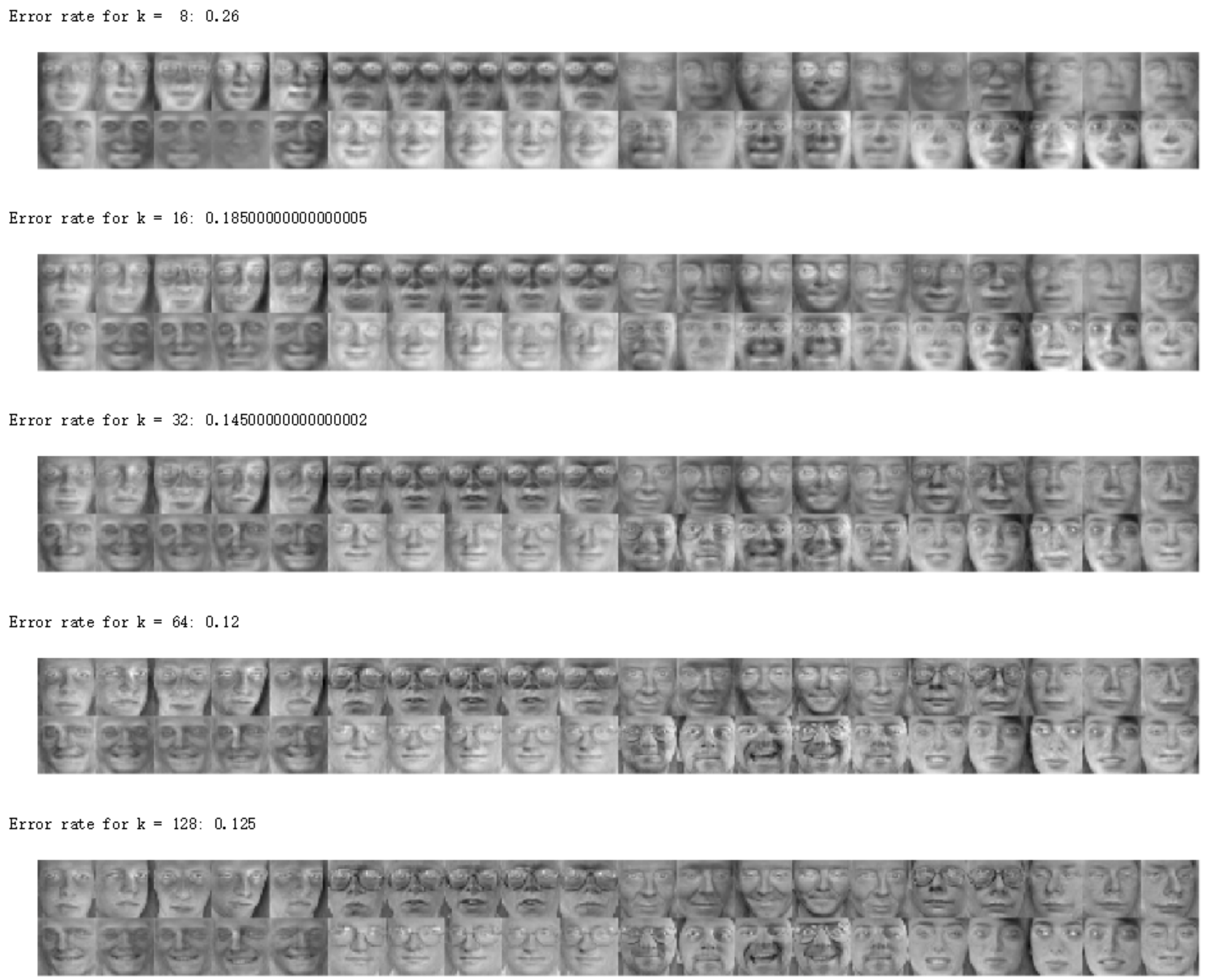


Figure 12: The error results and face visualizations for each K

3. The LDA result is much better than that PCA! Only 3%!

```
eigen_vec, eigen_val = LDA(fea_Train, gnd_Train)
compress_Train = np.matmul(fea_Train, eigen_vec)
compress_Test = np.matmul(fea_Test, eigen_vec)
pred_Test = knn(compress_Test, compress_Train, gnd_Train, 1)
error_rate = 1.0 - np.sum(pred_Test == gnd_Test) / pred_Test.shape[0]
print ("Error rate for LDA:", error_rate)
```

```
Error rate for LDA: 0.030000000000000027
```

Figure 13: The error result for LDA