ISYE6740-HW2

pkubsad

February 2021

1 Political blogs dataset [50 points.]

1. (10 points) Assume the number of clusters in the graph is k. Explain the meaning of k here intuitively.

   Answer: The value of k represents the number of clusters we want to divide the given data-set. In data set given in the questions, the hypothesis is that similar nodes connect with each other and form a community. If we draw the graph of all the nodes, 'k' represents the number of partitions we want the entire network to be partitioned into. When k=2, we would partition the graph into 2 networks. From the output of the code below, we can see that nodes 182 and 666 are clustered together as 1 cluster. All the remaining nodes are clustered together as 2nd cluster. If we see the edges txt file, we can see that nodes 182 and 666 are connected only to each other and form a good connected community.

2. (10 points) Use spectral clustering to find the $k = 2, 5, 10, 20$ clusters in the network of political blogs (each node is a blog, and their edges are defined in the file edges.txt). Then report the majority labels in each cluster, for different k values, respectively.

   Answer: I have used the spectral clustering method explained at the beginning of lecture 3 (not special spectral method, or the svd method). The approach is:

   (a) Remove disconnected nodes resulting in 1224 nodes.
   (b) Create a temporary list to maintain mapping between all the nodes in edges file to reduced 1224 nodes.
   (c) Create adjacency matrix based on nodes appearing in edges file.
   (d) Create a diagonal matrix based on degrees of each node.
   (e) Calculate graph laplacian L = D - A.
   (f) Calculate eigen values and eigen vector using L
   (g) Sort eigen values asecending, and pick the lowest k eigen vectors.
   (h) Run k-means algorithm using the eigen vectors shortlisted in the step above.

   Based on the output of the code at the time, here are the observations:

k	Labels	Counts	Cluster Mismatch	Overall Mismatch
2	cluster 0 ->1 cluster 1 ->0	cluster 0 ->1: 636 ; 0:586 cluster 1->0:2	cluster 0->0.478 cluster 1 ->0	0.478
5	cluster 0 ->1 cluster 1 ->0 cluster 2 ->0 cluster 3 ->0 cluster 4 ->1	cluster 0 ->1:4 cluster 1 ->0:2 cluster 2 ->0:2 ; 1:2 cluster 3 ->0:2 cluster 4 ->1: 630 ; 0:582	cluster 0 ->0 cluster 1 ->0 cluster 2 ->0.5 cluster 3 ->0 cluster 4 ->0.481	0.477
10	cluster 0 ->1 cluster 1 ->1 cluster 2 ->0 cluster 3 ->0 cluster 4 ->1 cluster 5 ->0 cluster 6 ->1 cluster 7 ->0 cluster 8 ->1 cluster 9 ->0	cluster 0 ->1:2 cluster 1 ->1:584 , 0: 556 cluster 2 ->0:2 cluster 3 ->0:2 cluster 4 ->1: 2 cluster 5 ->0:2 cluster 6 ->1:44 , 0:17 cluster 7 ->0:2 cluster 8 ->1:4 , 0:2 cluster 9 ->0:5	cluster 0 ->0 cluster 1 ->0.487 cluster 2 ->0 cluster 3 ->0 cluster 4 ->0 cluster 5 ->0 cluster 6 ->0.278 cluster 7 ->0 cluster 8 ->0.333 cluster 9 ->0	0.469
20	cluster 0 ->1 cluster 1 ->0 cluster 2 ->0 cluster 3 ->0 cluster 4 ->0 cluster 5 ->1 cluster 6 ->0 cluster 7 ->0 cluster 8 ->0 cluster 9 ->0 cluster 10 ->1 cluster 11 ->0 cluster 12 ->1 cluster 13 ->0 cluster 14 ->1 cluster 15 ->0 cluster 16 ->1 cluster 17 ->0 cluster 18 ->0 cluster 19 ->0	cluster 0 ->1:22, 0:7 cluster 1 ->0:5 cluster 2 ->1:15, 0:466 cluster 3 ->0:2 cluster 4 ->0:2 cluster 5 ->1:4, 0:2 cluster 6 ->0:1 cluster 7 ->0:2 cluster 8 ->0:2 cluster 9 ->1:3, 0:43 cluster 10 ->1:564,0:28 cluster 11 ->0:1 cluster 12 ->1:2 cluster 13 ->0:1 cluster 14 ->1:21,0:1 cluster 15 ->0:13 cluster 16 ->1:2 cluster 17 ->0:2 cluster 18 ->1:3, 0:8 cluster 9 ->0:2	cluster 0 ->0.241 cluster 1 ->0 cluster 2 ->0.031 cluster 3 ->0 cluster 4 ->0 cluster 5 ->0.333 cluster 6 ->0 cluster 7 ->0 cluster 8 ->0 cluster 9 ->0.065 cluster 10 ->0.047 cluster 11 ->0 cluster 12 ->0 cluster 13 ->0 cluster 14 ->0.045 cluster 15 ->0 cluster 16 ->0 cluster 17 ->0 cluster 18 ->0.272 cluster 19 ->0	0.0482

3. (10 points) Now compare the majority label with the individual labels in each cluster, and report the *mismatch rate* for each cluster, when $k = 2, 5, 10, 20$.

   Answer:

   The table above also has recordings of the observed mismatch rate for each cluster. The overall mismatch rate for k is calculated by $\frac{noofmismatchesinalltheclusters}{totalnoofnodes}$.

4. (10 points) Tune your k and find the number of clusters to achieve a reasonably small *mismatch rate*. Please explain how you tune k and what is the achieved mismatch rate.

   Answer: I followed brute force approach of iterating through Ks from 2 to 300. The gain in the mismatch rate is marginal once the value of k¿19. So, I reduced the runs for k=2 to 150 and tracked

the mismatch rate for each value of k. As per the run during this documentation, I see:

$$k = 99, mismatch - rate = 0.0359$$

The plot of mismatch rate vs k also looks like elbow method where we see that after k=19, there is not much improvement in the mismatch rate.

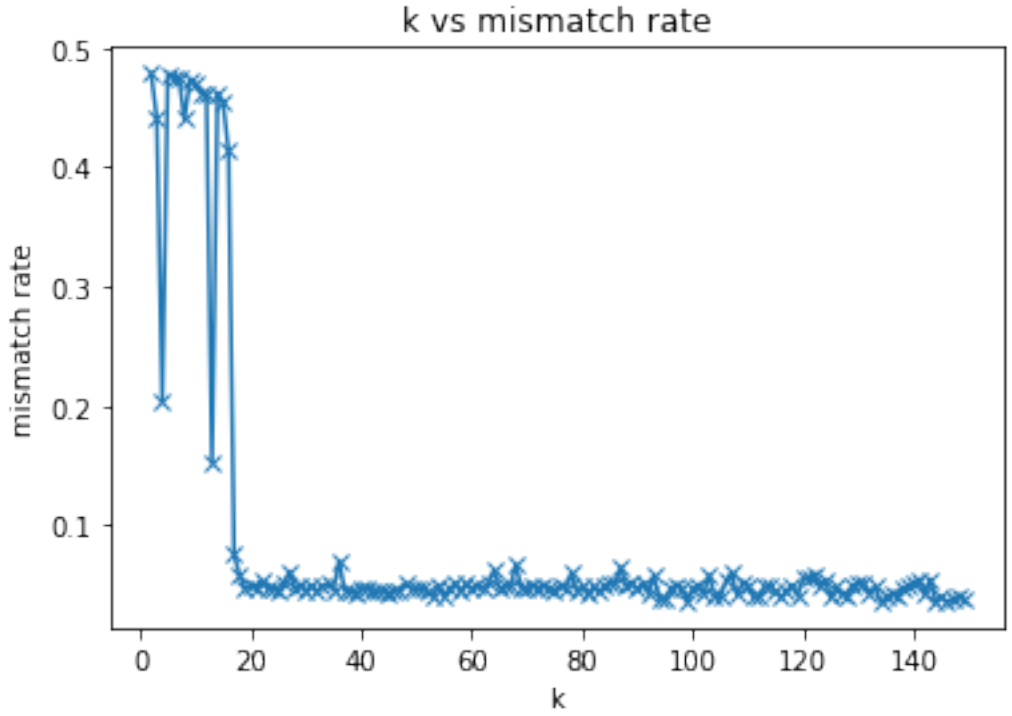


Figure 1: mismatch rate v/s k

5. (10 points) Please explain the finding and what can you learn from this data analysis.

Answer: The hypothesis of this experiment is that, similar nodes are connected to each other. These connected nodes create a community within a graph. Spectral Clustering on this graph data would divide the graph into 'k' sub networks as a cluster. Each cluster will mostly have similar nodes.
When k=2, we can see that in the 2nd cluster only 2 nodes were included. These 2 nodes, 182 and 666 are connected only to each other and form a good connected community. We can visualise by creating the graph of all the edges, we can see, nodes 636 and 586 are separated away from the overall network.

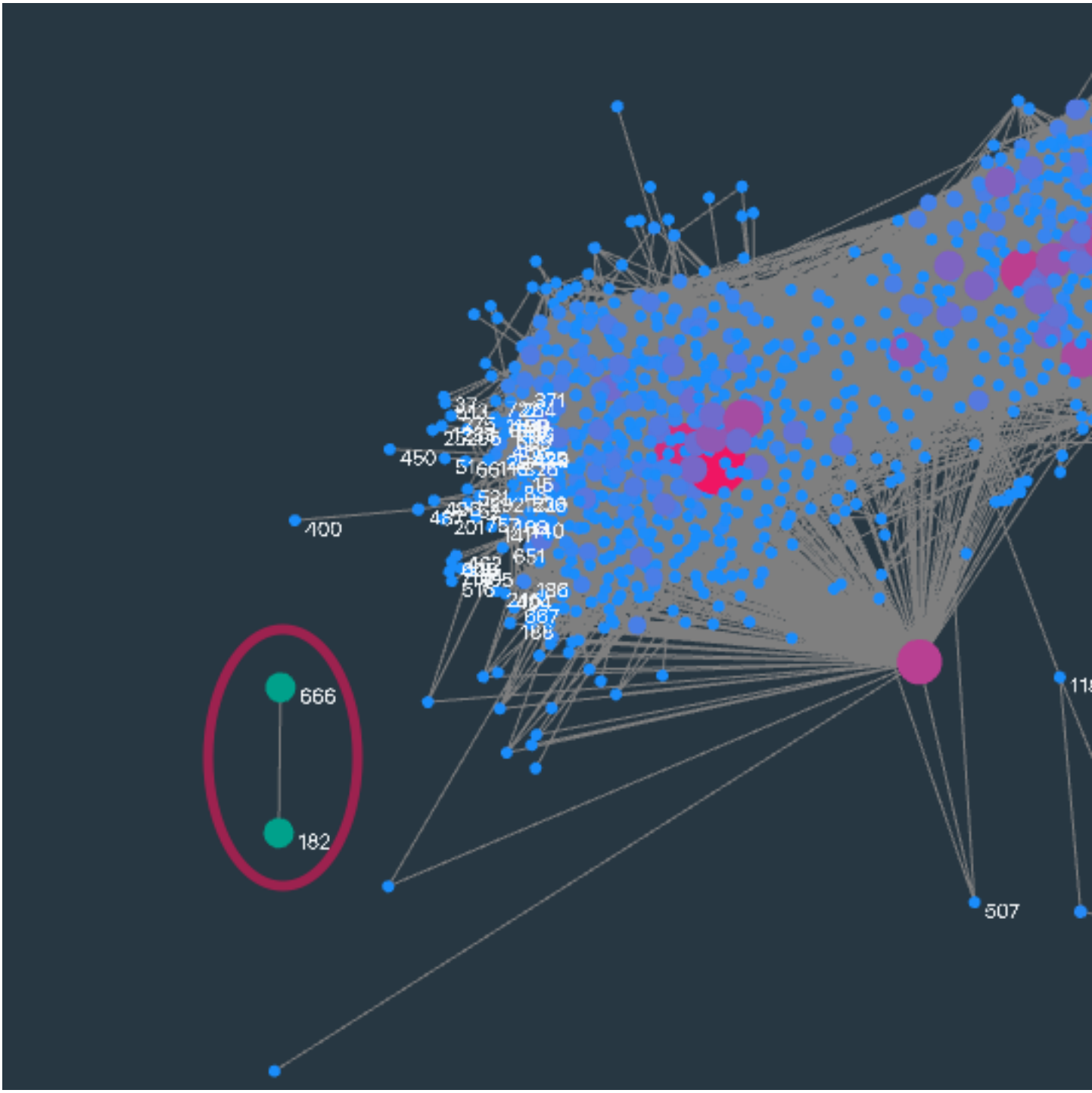


Figure 2: visualization of the edges.txt, credits: https://poloclub.github.io/argo-graph-lite/

After k value goes beyond 19, we dont see much improvement in the mismatch rate. But within the clusters formed, we see that the nodes have same political orientation are linked together. We can conclude the initial hypothesis mentioned in the homework is true.

If I run the same problem with special clustering method given in the demo code, I start seeing higher accuracy to earlier values of k. The difference between the special clustering approach v/s the normal approach comes down to the point of how python handles determination of eigen values. In my experiment and couple of posts on piazza, python performs better in determining higher eigen values compared to calculating lower eigen values. In special clustering method, we take top 'k' eigen values, it performs sligthly better than calculating lowest 'k' eigen values.

Since the approach uses kmeans at the end, the outcome is different every time we run the code. This can be attributed to the randomness of the initial centroids selected.As far as practical applications are concerned, I can easily see this approach/algorithm can be used for targeted marketing. Determine community of people within a network that share common interest and customize the marketing message to that community.

2.1 (25 points) Perform analysis on the Yale face dataset for Subject 1 and Subject 2, respectively, using all the images EXCEPT for the two pictures named subject01-test.gif and subject02-test.gif. **Plot the first 6 eigenfaces for each subject.** When visualizing, please reshape the eigenvectors into proper images. Please explain can you see any patterns in the top 6 eigenfaces?

Answer: Each eigen face that is generated using the eigen vectors from PCA output, captures some variability from the original picture, like shadow. In higher valued eigen faces these features are distinctly seen, like shadow on left side, and with eigenfaces of lower values, these features start diminishing. Each eigen face seems like approximation of the original image.

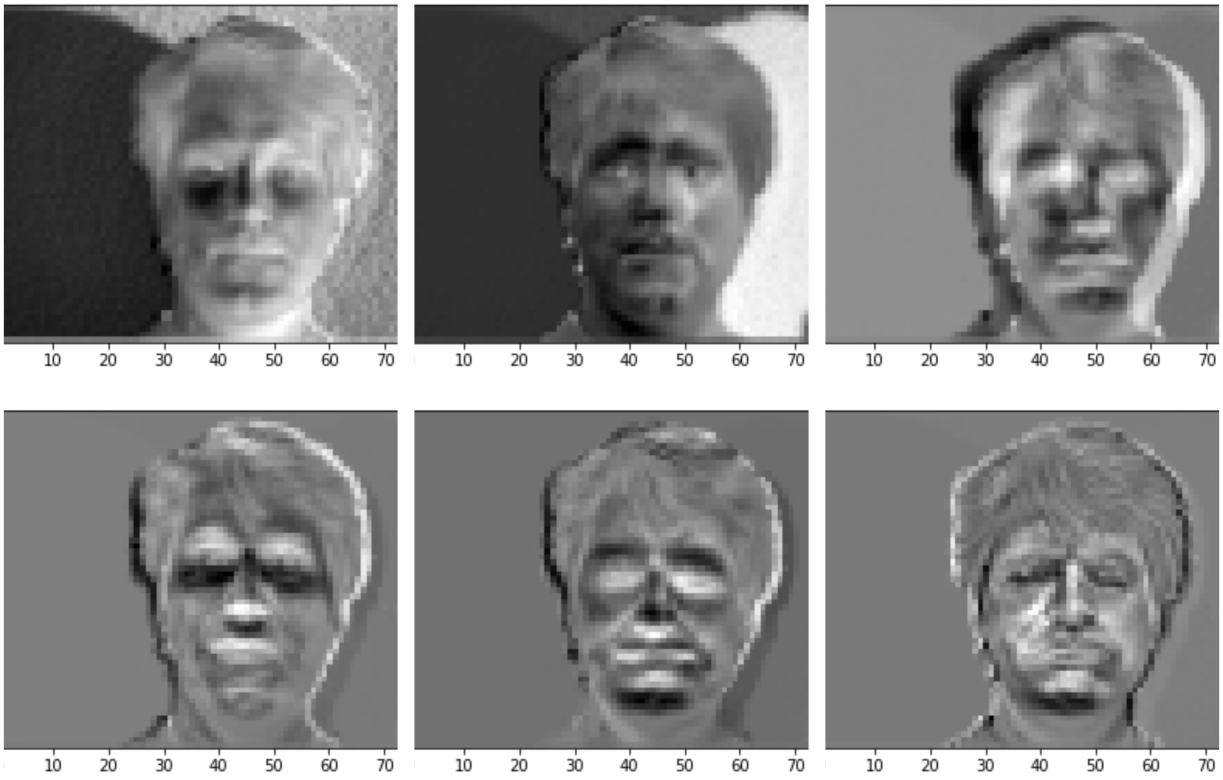


Figure 3: eigen faces of subject 1

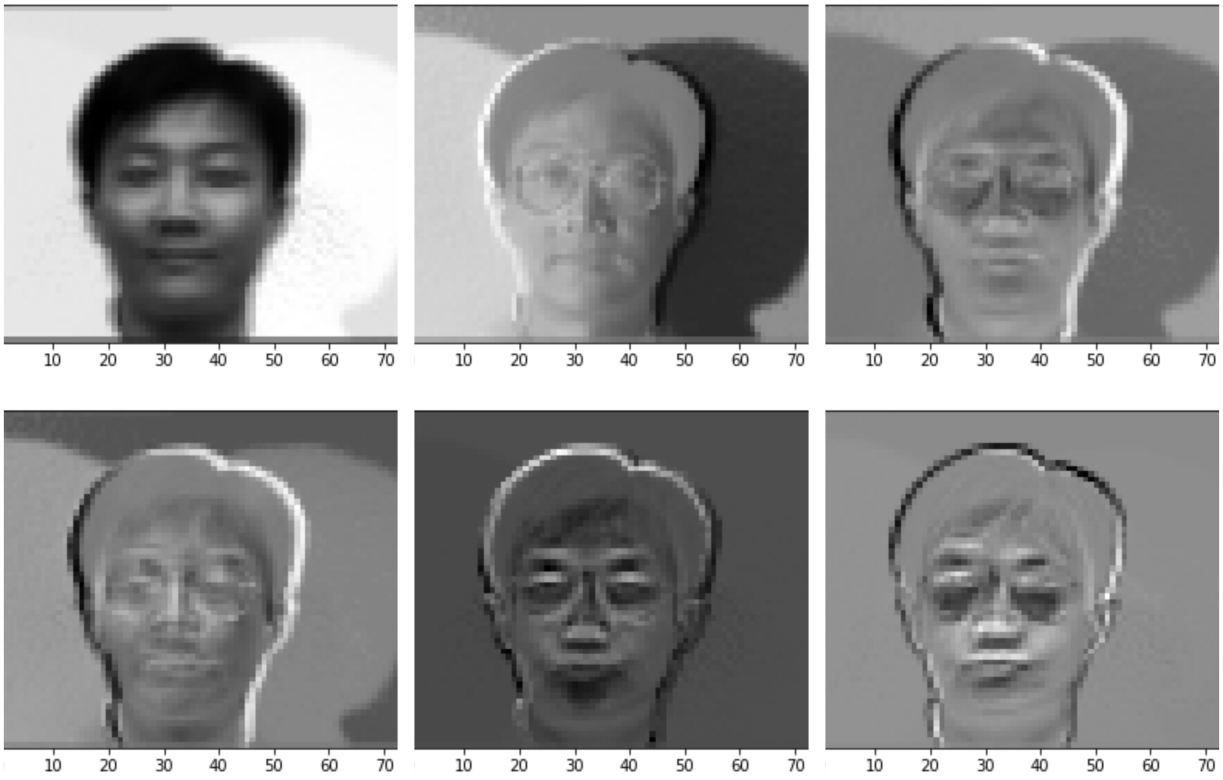


Figure 4: eigen faces of subject 2

1. (25 points) Now we will perform a simple face recognition task. ace recognition through PCA is proceeded as follows. Given the test image subject01-test.gif and subject02-test.gif, first downsize by a factor of 4 (as before), and vectorize each image. Take the top eigenfaces of Subject 1 and Subject 2, respectively. Then we calculate the *normalized inner product score* of the 2 vectorized test images with the vectorized eigenfaces:

$$s_{ij} = \frac{(\text{eigenface})_i^T (\text{test image})_j}{\|(\text{eigenface}_i)\| \cdot \|(\text{test image})_j\|}$$

Answer: Using the calculation mentioned above,I have the following table:

$$\begin{bmatrix} 0.87740349 & 0.70005538 \\ 0.0933761 & 0.41782208 \end{bmatrix}$$

The normalized inner product, also referred to with $\cos\theta$ represents how similar 2 vectors are. Value closer to 1, represents the features point in the same direction, -1 represents the features are pointing in opposite direction and 0 represents the features are orthogonal. Picking the large value closer to +1, we can conclude that S11 and S22 are matching faces.

2. (Bonus: 5 points) Explain if face recognition can work well and discuss how we can improve it possibly.

Answer: In the experiment above, we can see that S12 is somewhat close to S11. Also, the model is matching the test subject 2 more with subject1 compared to subject2. If we can train our model with

more number of faces and for each subject we can get better results with the test case.
Instead of picking the top eigen face, when I run the test images against all the eigen faces, I see better results for test2 face with 2nd and 3rd eigen face. So, I looked up, how to select best eigen face to compare against. I found a paper that talks about averaging eigen face vectors.
Reference: https://www.face-rec.org/algorithms/PCA/jcn.pdf

Q1

February 10, 2021

[9]:
```
import os
import numpy as np
from os.path import abspath, exists
from scipy import sparse
from sklearn.cluster import KMeans
from matplotlib import pyplot as plt
import csv
from collections import Counter
```

[10]:
```
def import_graph():
    # read the graph from 'play_graph.txt'
    f_path = abspath("data/edges.txt")
    if exists(f_path):
        with open(f_path) as graph_file:
            lines = [line.split() for line in graph_file]
    ret = np.array(lines).astype(int)
#     print(len(ret))
#     print(ret.shape)
    return ret
```

[11]:
```
def check_symmetric(a, rtol=1e-05, atol=1e-08):
    return np.allclose(a, a.T, rtol=rtol, atol=atol)
```

[12]:
```
def get_nodes_removing_unpaired(a):
    f_path = abspath("data/nodes.txt")
#     print(type(a[:,0][0]))
    ctr=0
    ret=[]
    if exists(f_path):
        with open(f_path) as nodes_file:
            for line in nodes_file.readlines():
                node = line.split("\t")[0]
                if int(node) in a[:,0] or int(node) in a[:,1]:
                    ctr+=1
                    ret.append([int(node),line.split("\t")[0],line.
 ↪split("\t")[2]])
```

```
#     print(ctr)
    return ret
```

[13]:
```
def get_unique_nodes_in_edges(a):
    i=a[:,0]
    j=a[:,1]
    ik=np.append(i, j, 0)
    dedupedi=list(dict.fromkeys(ik))
    dedupedi.sort()
    return dedupedi
```

[14]:
```
#@Param k_clusters: number of different values of k that the code has to be run.
#@Param verbose: prints a lot of data if set to True. Using this to avoid a lot␣
 ↪
#               of clutter in the output when the number of k values to be␣
 ↪tested is large
# Reference: Used the demo code provided by professor to read the graph and␣
 ↪determine eigen values
def runSpectralClustering(k_clusters=(2,5,10,20), verbose=True):



    # load the graph
    a = import_graph()

    # get all the unique nodes from edges
    dedupedi = get_unique_nodes_in_edges(a)

    # create an array to map nodes in edges list to seq number from 0.
    # this is done because after removing isolated nodes, we will be left with␣
 ↪1224 nodes
    # instead of original 1490 nodes. But in nodes.txt, there are nodes␣
 ↪exceeding 1224, so we
    # have to re map the nodes to new numbering. Once the clusters are␣
 ↪determined, we will refer back
    # to this map to get the original node id.
    adjacency_node_mapping=np.array([[i, dedupedi[i]] for i in␣
 ↪range(len(dedupedi))])


    if verbose:
        print("checking if the resulting node mapping is right, new index for␣
 ↪node id 1488 is ",np.where(adjacency_node_mapping[:,1]==1488))
```

```
        print("checking for reverse mapping, node id for 1221 is␣
↪",adjacency_node_mapping[1221][1])

    # initializing empty adjaceny matrix
    A = np.zeros(shape=(1224,1224))

    # Looping through all the edges,
    # getting the nodes of an edge,
    # get the new index for that node id,
    # set value equal to 1 for the co ordinates represented by te nodes in the␣
↪edge.
    for edge in a:
        A[np.where(adjacency_node_mapping[:,1]==edge[0])[0],np.
↪where(adjacency_node_mapping[:,1]==edge[1])[0]]=1
        A[np.where(adjacency_node_mapping[:,1]==edge[1])[0],np.
↪where(adjacency_node_mapping[:,1]==edge[0])[0]]=1

    #check if the adjacency matrix is symmetric
    if not check_symmetric(A):
        raise error

    if verbose:
        display(A)

    #caluclating the diagonal matrix
    D = np.diag(np.sum(A, axis=1))

    #calculate graph laplacian
    L = D - A



    mismatchList=[]
    klist=[]

     # calculating eigen values and eigen vector
    eigenValues, eigenVector = np.linalg.eig(L)


    for k in k_clusters:
        v=eigenValues.real
        x=eigenVector.real

        #sort the eigen values and pick the lowest k eigen values.
        # Reference: https://stackoverflow.com/questions/8092920/
↪sort-eigenvalues-and-associated-eigenvectors-after-using-numpy-linalg-eig-in-pyt
        idx = v.argsort()[::1]
```

```
        v = v[idx]
        x = x[:,idx]
        x = x[:, 0:k]

        # k-means
        kmeans = KMeans(n_clusters=k,init='random').fit(x)
        c_idx = kmeans.labels_

        # removing the nodes that are unpaired
        nodes_paired=np.array(get_nodes_removing_unpaired(a)).astype(int)

        # Determining the nodes that are clustered for each label.
        clusters=[]
        for i in range(k):
            ctr=0
            cluster=[]
            idx = [index for index, t in enumerate(c_idx) if t == i]
            for index in idx:
                ctr+=1
                corresponding_node=adjacency_node_mapping[index][1]
                row_idx_nodes=np.where(nodes_paired[:,0]==corresponding_node)[0]
                corresponding_node_name=nodes_paired[row_idx_nodes][0][1]
                corresponding_node_true_flag=nodes_paired[row_idx_nodes][0][2]
                item=(corresponding_node_name,corresponding_node_true_flag)
                cluster.append(item)
            clusters.append(cluster)


        i=1
        totalMismatches=0

        # Determining mismatches in each cluster and eventual mismatch for k
        for cluster in clusters:
            counter = Counter(elem[1] for elem in cluster)
            if verbose:
                print("entries in ",i,"th cluster: ",counter)
            count1= counter.get(1) if 1 in counter.keys() else 0
            count0= counter.get(0) if 0 in counter.keys() else 0
            if(count1>count0):
                clusterLabel=1
                totalMismatches=totalMismatches+count0
                if count1+count0>0:
                    mismatchRateCluster=count0/(count1+count0)
                else:
                    mismatchRateCluster = 0
            else:
                clusterLabel=0
```

```
                totalMismatches=totalMismatches+count1
                if count1+count0>0:
                    mismatchRateCluster=count1/(count1+count0)
                else:
                    mismatchRateCluster = 0
            if verbose:
                print("cluster label:", clusterLabel)
                print("count of 1:", count1)
                print("count of 0:", count0)
                print("mismatch rate:",mismatchRateCluster)
            i+=1
        totalMismatchRate=totalMismatches/1224
        klist.append(k)
        mismatchList.append(totalMismatchRate)
        if verbose or totalMismatchRate<0.05:
            print("total mismatch rate for k=",k, " is :",totalMismatchRate)
            print("***************************************************")
            print()
            print()
    if verbose is False:
        plt.plot(klist, mismatchList, marker='x')
        plt.title('k vs mismatch rate')
        plt.xlabel('k')
        plt.ylabel('mismatch rate')
        plt.show()
        minimumMismatch= min(mismatchList)
        k_min_mismatch=mismatchList.index(min(mismatchList))
        print("lowest mistmatch rate is :", minimumMismatch, " for k = ",␣
↪k_min_mismatch+2)
```

[15]:
```
#Q1.2 and Q1.3: Calling the runSpectralClustering method with default␣
↪parameters.
#This will run the logic for K=2,5,10,20 and Verbose=True(full logging.)
runSpectralClustering()
```

```
checking if the resulting node mapping is right, new index for node id 1488 is
(array([1221]),)
checking for reverse mapping, node id for 1221 is  1488
```

```
array([[0., 1., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

```
entries in  1 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
entries in  2 th cluster:  Counter({1: 636, 0: 586})
cluster label: 1
count of 1: 636
count of 0: 586
mismatch rate: 0.4795417348608838
total mismatch rate for k= 2  is : 0.47875816993464054
**************************************************


entries in  1 th cluster:  Counter({0: 2, 1: 2})
cluster label: 0
count of 1: 2
count of 0: 2
mismatch rate: 0.5
entries in  2 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
entries in  3 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
entries in  4 th cluster:  Counter({1: 630, 0: 582})
cluster label: 1
count of 1: 630
count of 0: 582
mismatch rate: 0.4801980198019802
entries in  5 th cluster:  Counter({1: 4})
cluster label: 1
count of 1: 4
count of 0: 0
mismatch rate: 0.0
total mismatch rate for k= 5  is : 0.477124183006536
**************************************************


entries in  1 th cluster:  Counter({0: 3})
cluster label: 0
count of 1: 0
count of 0: 3
mismatch rate: 0.0
```

```
entries in  2 th cluster:  Counter({1: 584, 0: 553})
cluster label: 1
count of 1: 584
count of 0: 553
mismatch rate: 0.48636763412489004
entries in  3 th cluster:  Counter({0: 1})
cluster label: 0
count of 1: 0
count of 0: 1
mismatch rate: 0.0
entries in  4 th cluster:  Counter({1: 2})
cluster label: 1
count of 1: 2
count of 0: 0
mismatch rate: 0.0
entries in  5 th cluster:  Counter({1: 2})
cluster label: 1
count of 1: 2
count of 0: 0
mismatch rate: 0.0
entries in  6 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
entries in  7 th cluster:  Counter({1: 44, 0: 17})
cluster label: 1
count of 1: 44
count of 0: 17
mismatch rate: 0.2786885245901639
entries in  8 th cluster:  Counter({0: 5})
cluster label: 0
count of 1: 0
count of 0: 5
mismatch rate: 0.0
entries in  9 th cluster:  Counter({0: 7})
cluster label: 0
count of 1: 0
count of 0: 7
mismatch rate: 0.0
entries in  10 th cluster:  Counter({1: 4})
cluster label: 1
count of 1: 4
count of 0: 0
mismatch rate: 0.0
total mismatch rate for k= 10  is : 0.46568627450980393
************************************************
```

```
entries in  1 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
entries in  2 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
entries in  3 th cluster:  Counter({0: 7, 1: 2})
cluster label: 0
count of 1: 2
count of 0: 7
mismatch rate: 0.2222222222222222
entries in  4 th cluster:  Counter({1: 4})
cluster label: 1
count of 1: 4
count of 0: 0
mismatch rate: 0.0
entries in  5 th cluster:  Counter({1: 564, 0: 37})
cluster label: 1
count of 1: 564
count of 0: 37
mismatch rate: 0.06156405990016639
entries in  6 th cluster:  Counter({0: 5, 1: 3})
cluster label: 0
count of 1: 3
count of 0: 5
mismatch rate: 0.375
entries in  7 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
entries in  8 th cluster:  Counter({1: 22, 0: 7})
cluster label: 1
count of 1: 22
count of 0: 7
mismatch rate: 0.2413793103448276
entries in  9 th cluster:  Counter({1: 2})
cluster label: 1
count of 1: 2
count of 0: 0
mismatch rate: 0.0
entries in  10 th cluster:  Counter({0: 476, 1: 17})
cluster label: 0
```

```
count of 1: 17
count of 0: 476
mismatch rate: 0.034482758620689655
entries in  11 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
entries in  12 th cluster:  Counter({0: 33, 1: 1})
cluster label: 0
count of 1: 1
count of 0: 33
mismatch rate: 0.029411764705882353
entries in  13 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
entries in  14 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
entries in  15 th cluster:  Counter({1: 21, 0: 1})
cluster label: 1
count of 1: 21
count of 0: 1
mismatch rate: 0.045454545454545456
entries in  16 th cluster:  Counter({0: 4})
cluster label: 0
count of 1: 0
count of 0: 4
mismatch rate: 0.0
entries in  17 th cluster:  Counter({0: 1})
cluster label: 0
count of 1: 0
count of 0: 1
mismatch rate: 0.0
entries in  18 th cluster:  Counter({0: 1})
cluster label: 0
count of 1: 0
count of 0: 1
mismatch rate: 0.0
entries in  19 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
```

```
entries in  20 th cluster:  Counter({0: 2})
cluster label: 0
count of 1: 0
count of 0: 2
mismatch rate: 0.0
total mismatch rate for k= 20  is : 0.05555555555555555
**************************************************
```

[16]:
```
#Q1.4: Calling the runSpectralClustering method with K= 2 to 150 and
 ↪Verbose=False.
k = range(2,150)
runSpectralClustering(k,verbose=False)
```

```
total mismatch rate for k= 18  is : 0.04084967320261438
**************************************************


total mismatch rate for k= 19  is : 0.04820261437908497
**************************************************


total mismatch rate for k= 21  is : 0.04738562091503268
**************************************************


total mismatch rate for k= 24  is : 0.04656862745098039
**************************************************


total mismatch rate for k= 25  is : 0.042483660130718956
**************************************************


total mismatch rate for k= 26  is : 0.04656862745098039
**************************************************


total mismatch rate for k= 27  is : 0.041666666666666664
**************************************************


total mismatch rate for k= 28  is : 0.04820261437908497
**************************************************


total mismatch rate for k= 29  is : 0.041666666666666664
```

```
**************************************************


total mismatch rate for k= 30  is : 0.04411764705882353
**************************************************


total mismatch rate for k= 31  is : 0.04820261437908497
**************************************************


total mismatch rate for k= 32  is : 0.04656862745098039
**************************************************


total mismatch rate for k= 33  is : 0.0457516339869281
**************************************************


total mismatch rate for k= 36  is : 0.049019607843137254
**************************************************


total mismatch rate for k= 37  is : 0.04656862745098039
**************************************************


total mismatch rate for k= 38  is : 0.04738562091503268
**************************************************


total mismatch rate for k= 41  is : 0.04493464052287582
**************************************************


total mismatch rate for k= 42  is : 0.04738562091503268
**************************************************


total mismatch rate for k= 43  is : 0.04493464052287582
**************************************************


total mismatch rate for k= 44  is : 0.04820261437908497
**************************************************


total mismatch rate for k= 45  is : 0.04656862745098039
```

```
**************************************************

total mismatch rate for k= 46  is : 0.04411764705882353
**************************************************

total mismatch rate for k= 47  is : 0.04656862745098039
**************************************************

total mismatch rate for k= 48  is : 0.04493464052287582
**************************************************

total mismatch rate for k= 49  is : 0.0457516339869281
**************************************************

total mismatch rate for k= 50  is : 0.04493464052287582
**************************************************

total mismatch rate for k= 51  is : 0.041666666666666664
**************************************************

total mismatch rate for k= 52  is : 0.042483660130718956
**************************************************

total mismatch rate for k= 53  is : 0.041666666666666664
**************************************************

total mismatch rate for k= 54  is : 0.049019607843137254
**************************************************

total mismatch rate for k= 56  is : 0.04656862745098039
**************************************************

total mismatch rate for k= 57  is : 0.04738562091503268
**************************************************

total mismatch rate for k= 58  is : 0.042483660130718956
```

```
**************************************************


total mismatch rate for k= 60  is : 0.04656862745098039
**************************************************


total mismatch rate for k= 61  is : 0.0392156862745098
**************************************************


total mismatch rate for k= 68  is : 0.049836601307189546
**************************************************


total mismatch rate for k= 71  is : 0.04738562091503268
**************************************************


total mismatch rate for k= 72  is : 0.0392156862745098
**************************************************


total mismatch rate for k= 73  is : 0.04656862745098039
**************************************************


total mismatch rate for k= 74  is : 0.04738562091503268
**************************************************


total mismatch rate for k= 76  is : 0.04411764705882353
**************************************************


total mismatch rate for k= 77  is : 0.04411764705882353
**************************************************


total mismatch rate for k= 78  is : 0.04656862745098039
**************************************************


total mismatch rate for k= 79  is : 0.04493464052287582
**************************************************


total mismatch rate for k= 80  is : 0.04493464052287582
```

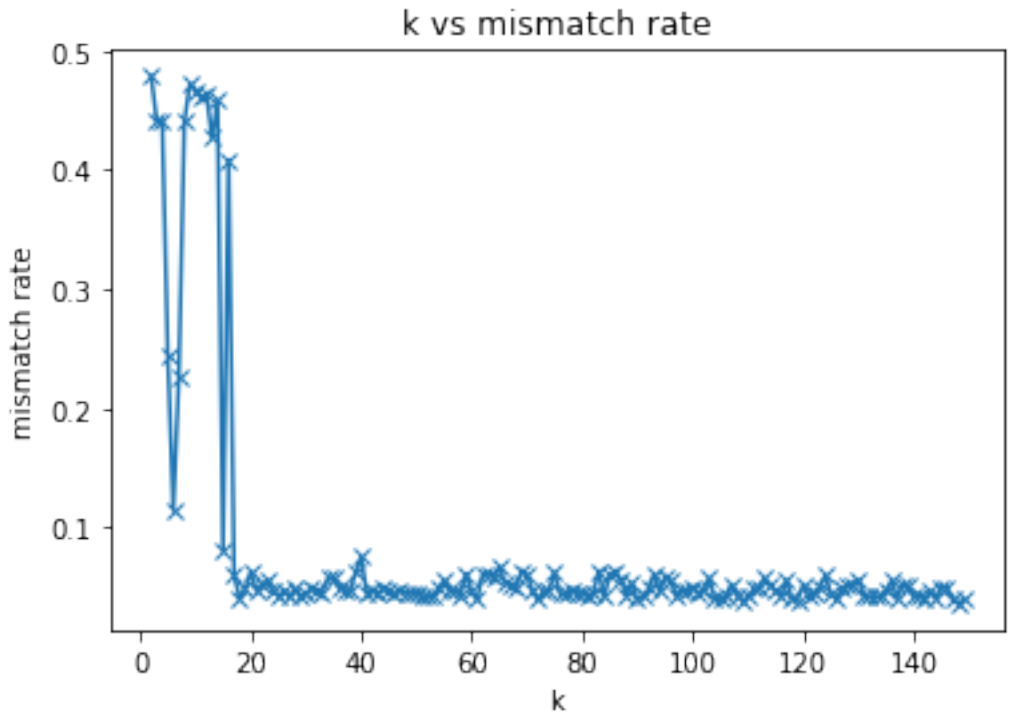
```
**************************************************

total mismatch rate for k= 81  is : 0.04330065359477124
**************************************************

total mismatch rate for k= 82  is : 0.04411764705882353
**************************************************

total mismatch rate for k= 84  is : 0.041666666666666664
**************************************************

total mismatch rate for k= 88  is : 0.04330065359477124
**************************************************

total mismatch rate for k= 90  is : 0.0392156862745098
**************************************************

total mismatch rate for k= 91  is : 0.042483660130718956
**************************************************

total mismatch rate for k= 92  is : 0.04820261437908497
**************************************************

total mismatch rate for k= 94  is : 0.04656862745098039
**************************************************

total mismatch rate for k= 97  is : 0.04330065359477124
**************************************************

total mismatch rate for k= 98  is : 0.04738562091503268
**************************************************

total mismatch rate for k= 99  is : 0.04656862745098039
**************************************************

total mismatch rate for k= 100  is : 0.049836601307189546
```

```
**************************************************

total mismatch rate for k= 101  is : 0.04493464052287582
**************************************************

total mismatch rate for k= 102  is : 0.04738562091503268
**************************************************

total mismatch rate for k= 104  is : 0.04003267973856209
**************************************************

total mismatch rate for k= 105  is : 0.0392156862745098
**************************************************

total mismatch rate for k= 106  is : 0.04330065359477124
**************************************************

total mismatch rate for k= 108  is : 0.04330065359477124
**************************************************

total mismatch rate for k= 109  is : 0.03839869281045752
**************************************************

total mismatch rate for k= 110  is : 0.041666666666666664
**************************************************

total mismatch rate for k= 111  is : 0.04820261437908497
**************************************************

total mismatch rate for k= 112  is : 0.049836601307189546
**************************************************

total mismatch rate for k= 115  is : 0.04738562091503268
**************************************************

total mismatch rate for k= 116  is : 0.042483660130718956
```

```
**************************************************


total mismatch rate for k= 118  is : 0.0392156862745098
**************************************************


total mismatch rate for k= 119  is : 0.03839869281045752
**************************************************


total mismatch rate for k= 121  is : 0.041666666666666664
**************************************************


total mismatch rate for k= 122  is : 0.0457516339869281
**************************************************


total mismatch rate for k= 123  is : 0.049836601307189546
**************************************************


total mismatch rate for k= 125  is : 0.04493464052287582
**************************************************


total mismatch rate for k= 126  is : 0.0392156862745098
**************************************************


total mismatch rate for k= 127  is : 0.049019607843137254
**************************************************


total mismatch rate for k= 131  is : 0.042483660130718956
**************************************************


total mismatch rate for k= 132  is : 0.041666666666666664
**************************************************


total mismatch rate for k= 133  is : 0.041666666666666664
**************************************************


total mismatch rate for k= 134  is : 0.041666666666666664
```

```
**************************************************


total mismatch rate for k= 135  is : 0.04656862745098039
**************************************************


total mismatch rate for k= 137  is : 0.04084967320261438
**************************************************


total mismatch rate for k= 140  is : 0.041666666666666664
**************************************************


total mismatch rate for k= 141  is : 0.042483660130718956
**************************************************


total mismatch rate for k= 142  is : 0.04003267973856209
**************************************************


total mismatch rate for k= 143  is : 0.04656862745098039
**************************************************


total mismatch rate for k= 144  is : 0.04084967320261438
**************************************************


total mismatch rate for k= 145  is : 0.049836601307189546
**************************************************


total mismatch rate for k= 146  is : 0.049836601307189546
**************************************************


total mismatch rate for k= 147  is : 0.0392156862745098
**************************************************


total mismatch rate for k= 148  is : 0.03594771241830065
**************************************************


total mismatch rate for k= 149  is : 0.0392156862745098
```

****




```
lowest mistmatch rate is : 0.03594771241830065  for k =  148
```

Q2

February 10, 2021

[10]:
```
from PIL import Image
from skimage.measure import block_reduce
from skimage.io import imread_collection
import numpy as np
from matplotlib.pyplot import imshow,show
import scipy.sparse.linalg as ll
import math
from numpy import linalg as LA
```

[11]:
```
def pre_process_img(col,pca_matrix):
    i=0
    for img in col:

        imgArray=np.array(img, dtype='int32')

        reducedImg =block_reduce(imgArray, block_size=(4,4), func=np.mean)

#         display(reducedImg)
#         imshow(reducedImg,cmap="gray")
#         show()
        vectorizedImg=reducedImg.reshape(-1)

#         print(vectorizedImg.shape)
#         display(vectorizedImg)

        pca_matrix[:, i] = vectorizedImg
        i+=1
```

[12]:
```
# Reference: Demo code provided by prof X, along with the hw.
def apply_pca(pca_matrix,K=6):
    m,n= pca_matrix.shape
    mu = np.mean(pca_matrix,axis = 1)
    xc = pca_matrix - mu[:,None]

    C = np.dot(xc,xc.T)/m
    print("---C1----")
    print(C)
```

```
    S,W = ll.eigs(C,k = K)
    S = S.real
    W = W.real
    print ("\n======== Top 6 Eigen Vectors ==========\n")
    print (W)
    print ("\n======== Top 6 Eigenvalues ==========\n")
    print (S)
    return S,W
```

[13]:
```
def print_eigen_faces(W):
    for col in W.T:
        W_reshaped=col.reshape(61,80)
        imshow(W_reshaped,cmap="gray")
        show()
```

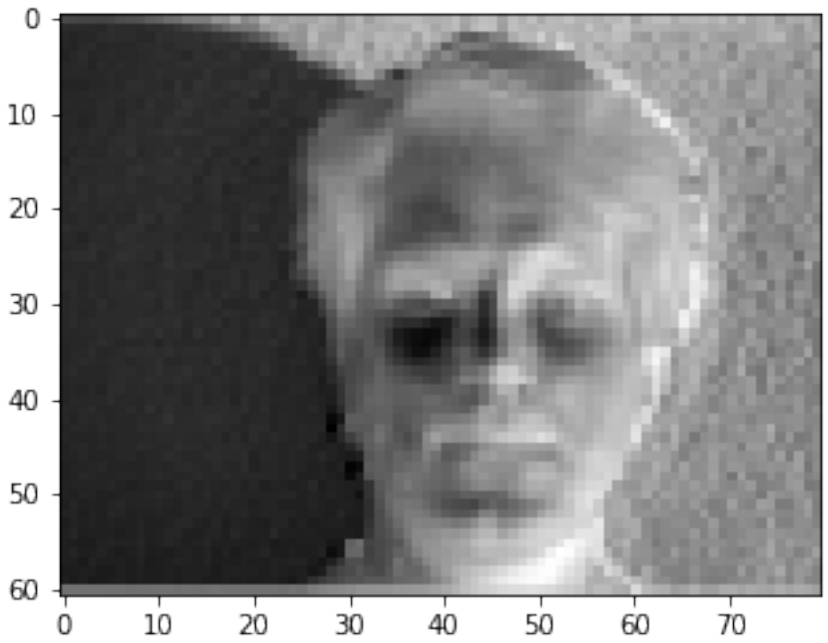
[14]:
```
pca_matrix_1 = np.zeros(shape=(4880,10))
pca_matrix_2 = np.zeros(shape=(4880,9))

collection_1 = imread_collection("data/yalefaces/subject01.*.gif")
collection_2 = imread_collection("data/yalefaces/subject02.*.gif")
print("coll 2 length: ",len(collection_2))


pre_process_img(collection_1,pca_matrix_1)
pre_process_img(collection_2,pca_matrix_2)


display(pca_matrix_1)
display(pca_matrix_2)

S1,W1 =apply_pca(pca_matrix_1)
S2,W2 =apply_pca(pca_matrix_2)



print_eigen_faces(W1)
print_eigen_faces(W2)
```

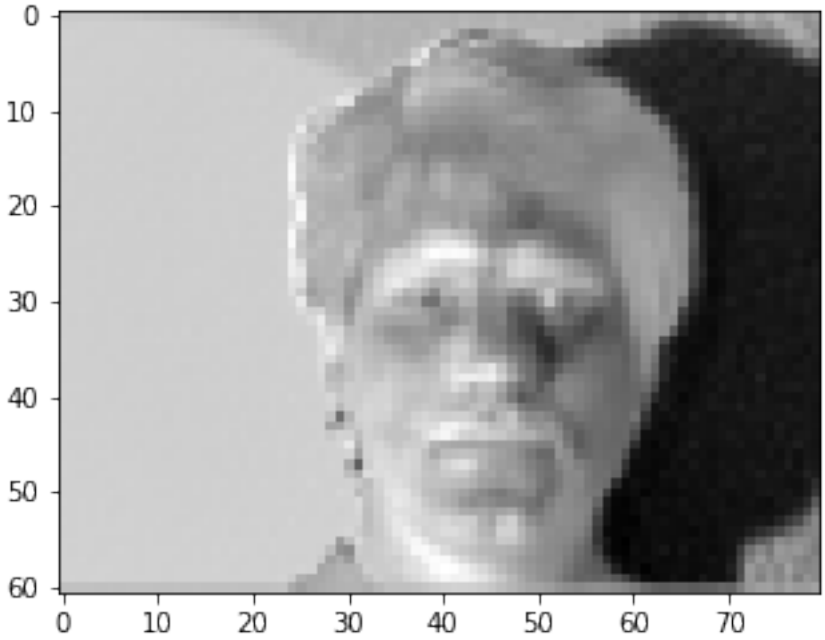
```
coll 2 length:  9
```

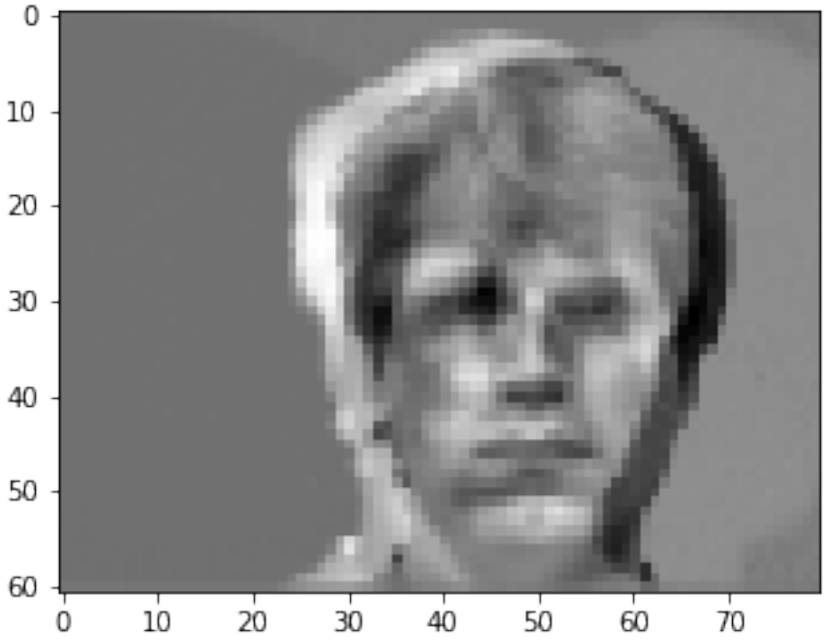
```
array([[223.75  , 223.75  , 223.75  , ..., 223.75  , 223.75  , 223.75  ],
       [223.75  , 223.75  , 223.75  , ..., 223.75  , 223.75  , 223.75  ],
       [223.75  , 223.75  , 223.6875, ..., 223.75  , 223.75  , 223.75  ],
       ...,
       [144.5   , 144.5   , 110.625 , ..., 144.5   , 144.5   , 144.0625],
       [143.6875, 143.5625, 110.75  , ..., 143.5625, 143.9375, 142.4375],
```

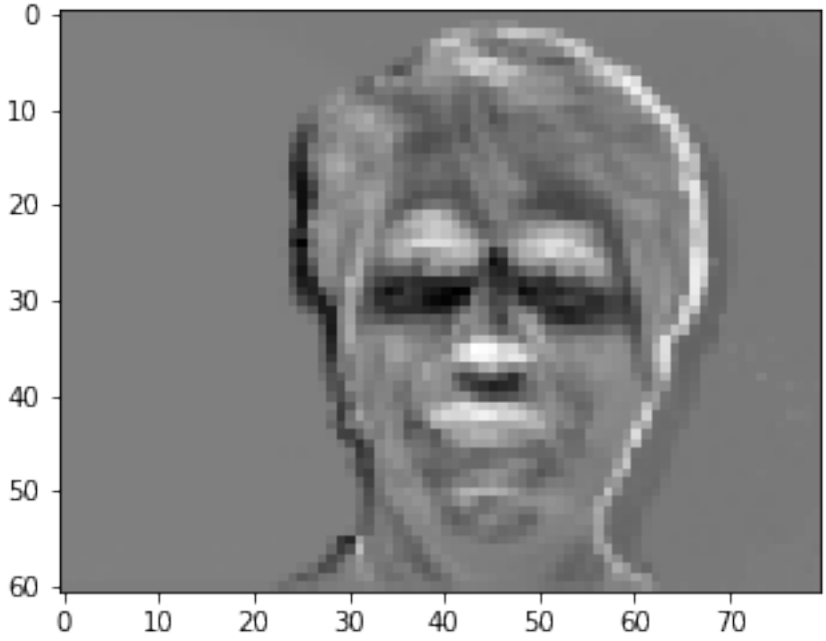
```
       [144.3125, 144.    , 117.    , ..., 144.25  , 144.375 , 143.3125]])


array([[223.75  , 223.75  , 223.75  , ..., 223.75  , 223.75  , 223.75  ],
       [223.375 , 223.1875, 223.75  , ..., 223.1875, 223.625 , 223.25  ],
       [221.0625, 223.75  , 223.75  , ..., 221.125 , 221.5625, 221.125 ],
       ...,
       [144.5   , 144.5   , 140.625 , ..., 144.5   , 143.75  , 144.5   ],
       [144.5   , 144.5   , 136.375 , ..., 144.5   , 144.375 , 144.5   ],
       [144.5   , 144.5   , 136.25  , ..., 144.5   , 144.375 , 144.5   ]])


---C1----
[[4.07729004 4.11156738 4.1662207  … 0.60842285 0.74134277 1.05250488]
 [4.11156738 4.14613289 4.20124568 … 0.6135378  0.74757516 1.06135318]
 [4.1662207  4.20124568 4.25709176 … 0.62207832 0.75788662 1.07577116]
 …
 [0.60842285 0.6135378  0.62207832 … 0.29916952 0.31325283 0.32481549]
 [0.74134277 0.74757516 0.75788662 … 0.31325283 0.33202973 0.3546474 ]
 [1.05250488 1.06135318 1.07577116 … 0.32481549 0.3546474  0.40687284]]

======== Top 6 Eigen Vectors ==========

[[-1.69689287e-02  4.73295464e-03 -3.25438700e-03  1.75100934e-03
  -5.82962315e-05  1.77323116e-03]
 [-1.71115847e-02  4.77274408e-03 -3.28174629e-03  1.76572989e-03
  -5.87863218e-05  1.78813853e-03]
 [-1.73406680e-02  4.82723478e-03 -3.32289393e-03  1.78871927e-03
  -5.88455933e-05  1.81174714e-03]
 …
 [-3.41188170e-03 -4.13925320e-03  8.48790500e-04  1.31547643e-05
   2.02132285e-04  2.81944717e-04]
 [-3.94095648e-03 -3.85219497e-03  7.03091737e-04 -1.52261571e-05
  -3.28367494e-04  4.40846382e-04]
 [-5.08989983e-03 -2.67633059e-03  2.73779507e-04  1.98256084e-04
  -2.60053354e-04  4.53887661e-04]]

======== Top 6 Eigenvalues ==========

[13411.96992116  8237.64820354  2520.02358817   911.53138026
   613.97588019   303.19377095]
---C1----
[[ 0.15584016  0.14376921  0.04303919 … -0.00374616 -0.00549436
  -0.00557761]
 [ 0.14376921  0.1327305   0.03975712 … -0.00374296 -0.00551464
  -0.00559825]
 [ 0.04303919  0.03975712  0.01372862 … -0.00233414 -0.00449912
  -0.00456814]
 …
```

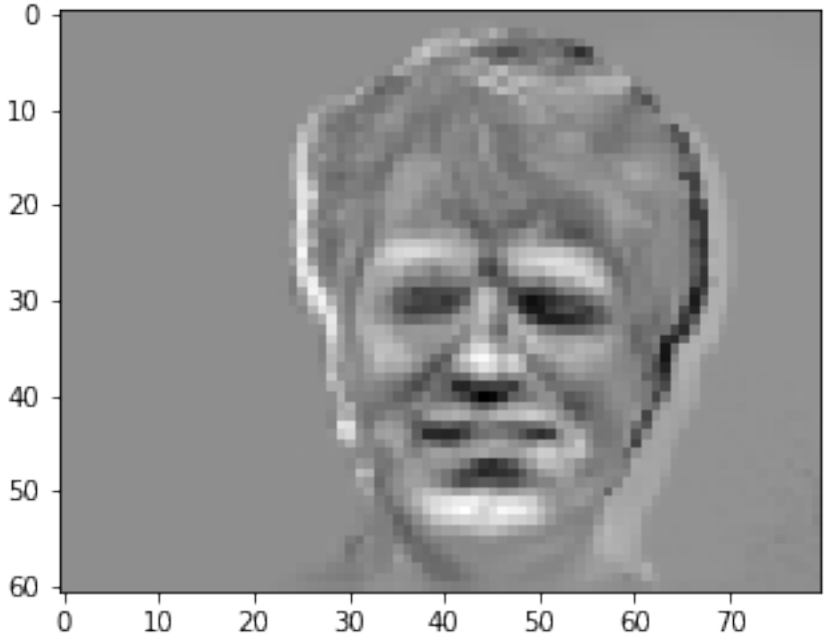
```
 [-0.00374616 -0.00374296 -0.00233414 …  0.00257428  0.00541432
   0.00549757]
 [-0.00549436 -0.00551464 -0.00449912 …  0.00541432  0.0119813
   0.01216594]
 [-0.00557761 -0.00559825 -0.00456814 …  0.00549757  0.01216594
   0.01235343]]

======== Top 6 Eigen Vectors ==========

[[ 1.81720250e-03  4.96492895e-03 -2.56283552e-03 -3.83840059e-04
   1.01098423e-03 -1.59315257e-04]
 [ 1.72718565e-03  4.57994981e-03 -2.15930257e-03 -4.14178311e-04
   1.12173306e-03 -1.49532964e-04]
 [ 7.64140742e-04  1.19356642e-03 -8.43639303e-04 -7.36996053e-04
  -4.07710446e-04  2.13077581e-03]
 …
 [-5.41906702e-04  1.48192281e-04 -1.97285097e-04 -7.99454638e-05
  -9.91499569e-05 -2.32184420e-04]
 [-1.14949179e-03  4.95649482e-04  8.57297842e-05  1.90490284e-05
   9.64383805e-05 -1.57246456e-04]
 [-1.16717922e-03  5.03381970e-04  8.73826808e-05  1.98639232e-05
   9.78953051e-05 -1.59241401e-04]]

======== Top 6 Eigenvalues ==========

[8158.60711064 4816.27429093 1493.4856265   361.43731129  247.67132279
  176.86543594]
```

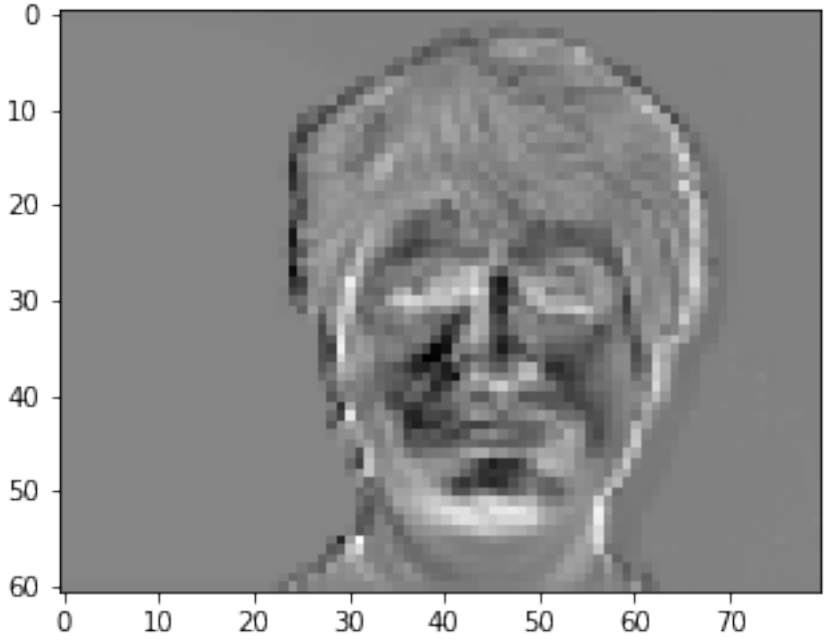


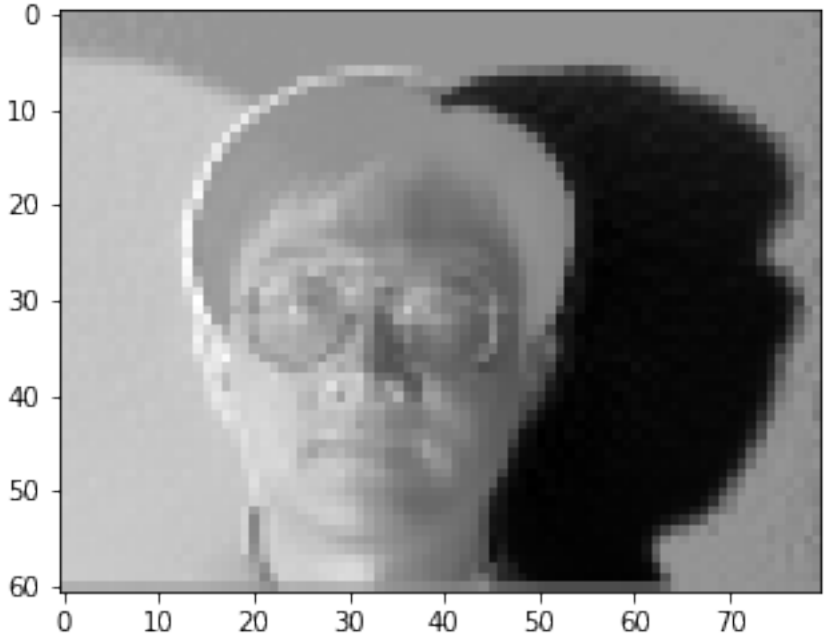

0
10
20
30
40
50
60
0
10
20
30
40
50
60
70

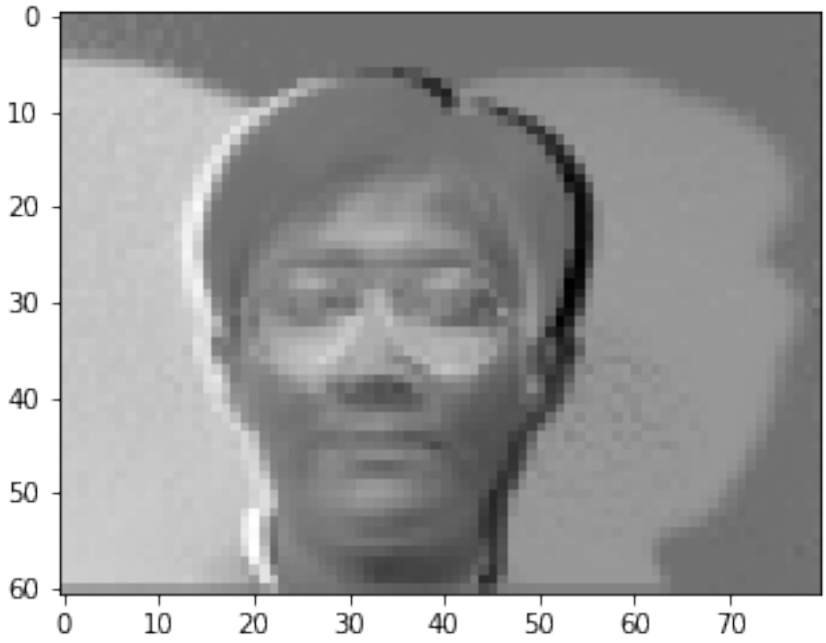

0
10
20
30
40
50
60
0
10
20
30
40
50
60
70

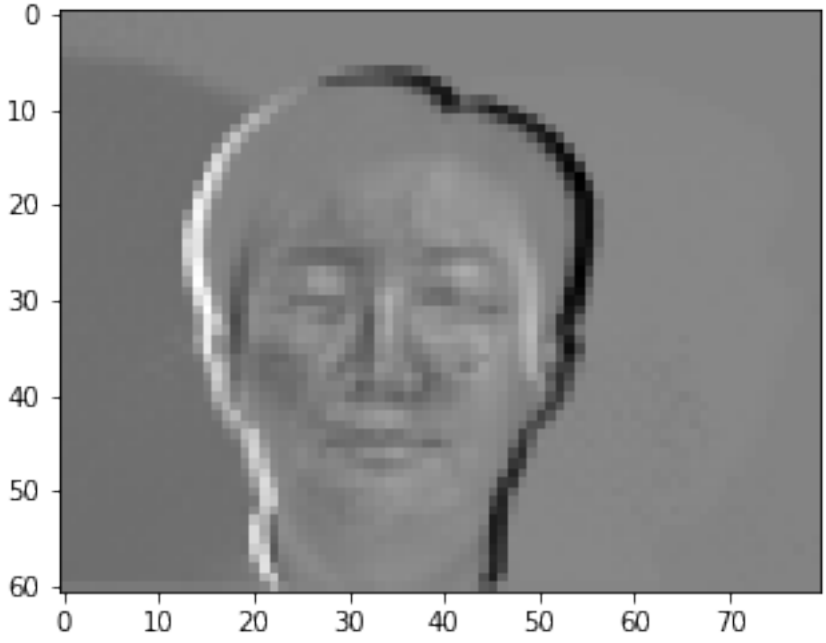

0
10
20
30
40
50
60
0
10
20
30
40
50
60
70

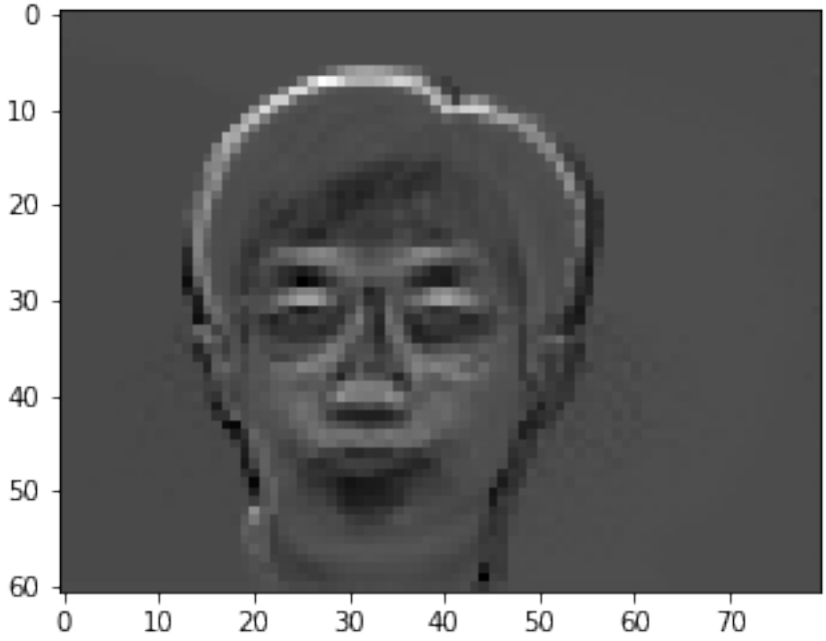

0
10
20
30
40
50
60
0
10
20
30
40
50
60
70

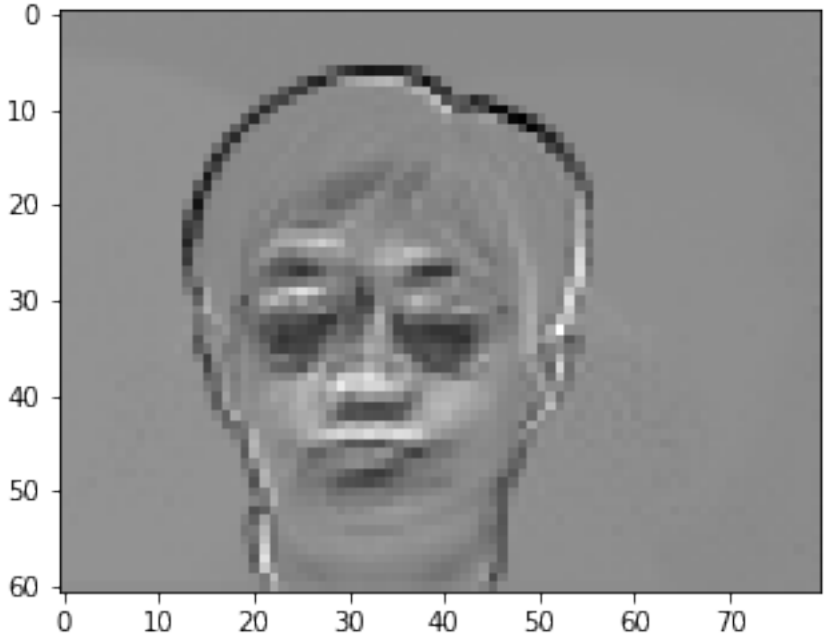

0
10
20
30
40
50
60
0
10
20
30
40
50
60
70

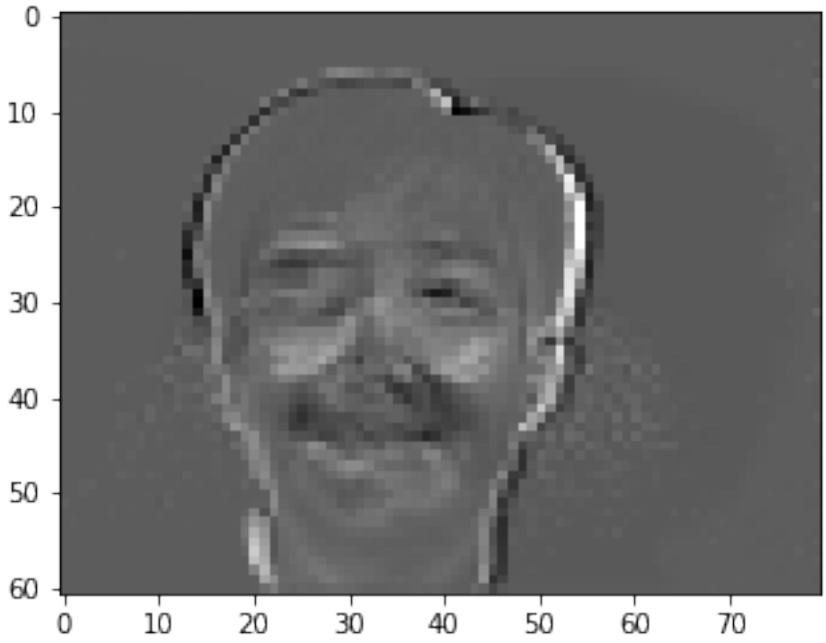

0
10
20
30
40
50
60
0
10
20
30
40
50
60
70


0
10
20
30
40
50
60
0
10
20
30
40
50
60
70


0
10
20
30
40
50
60
0
10
20
30
40
50
60
70


0
10
20
30
40
50
60
0
10
20
30
40
50
60
70

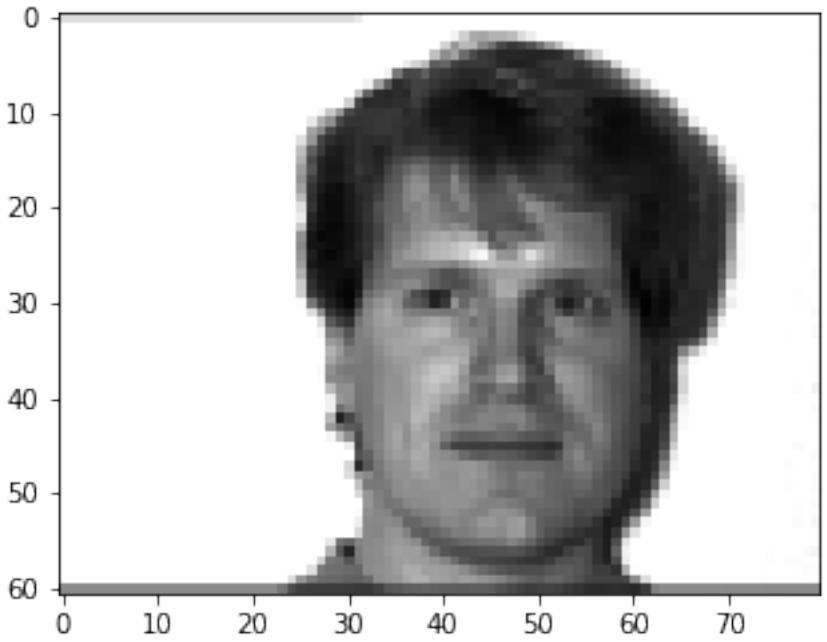

0
10
20
30
40
50
60
0
10
20
30
40
50
60
70



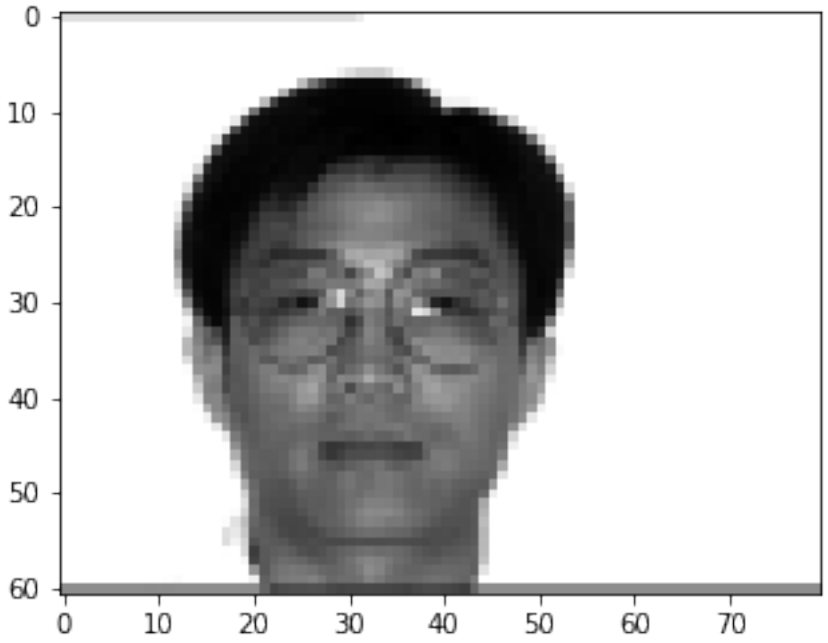
[15]:
```
topImage1=W1[:,0]
topImage1Norm=LA.norm(topImage1)

topImage2=W2[:,0]
topImage2Norm=LA.norm(topImage2)


col2 = imread_collection("data/yalefaces/*test*.gif")
result = np.zeros(shape=(2,2))
i=0
print(len(col2))
for img in col2:
    imgArray=np.array(img, dtype='int32')
    reducedImg =block_reduce(imgArray, block_size=(4,4), func=np.mean)

    imshow(reducedImg,cmap="gray")
    show()
    vectorizedImg=reducedImg.reshape(-1)

    testImageNorm = LA.norm(vectorizedImg)
    numerator=np.inner(topImage1.T,vectorizedImg)
    denom=topImage1Norm*testImageNorm
    sij1=numerator/denom
    print("s",i+1,"1:",sij1)
    result[0,i]=sij1
```

```
    numerator=np.inner(topImage2.T,vectorizedImg)
    denom=topImage2Norm*testImageNorm
    sij2=numerator/denom
    print("s",i+1,"2:",sij2)
    result[1,i]=sij2

    i+=1


print(result)
```

2




```
s 1 1: -0.8774034891566553
s 1 2: -0.093376100443711
```




```
s 2 1: -0.7000553801635848
s 2 2: -0.41782208225037215
[[-0.87740349 -0.70005538]
 [-0.0933761  -0.41782208]]
```