178 Hw 5

Justin Fu

80167602

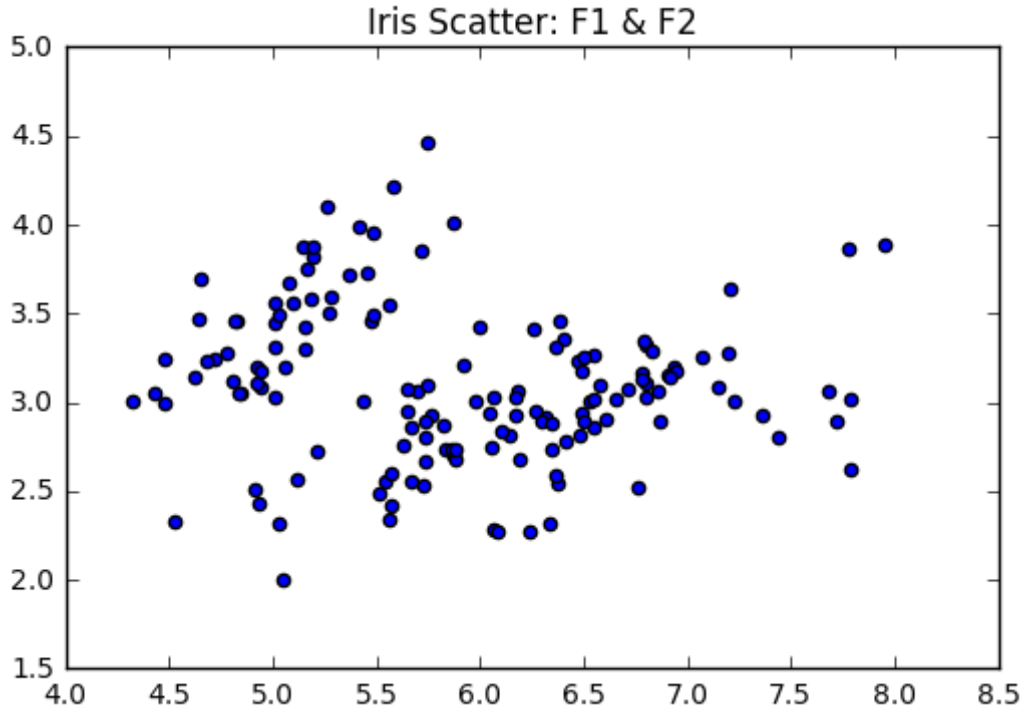
Problem 1: Basics of Clustering

Three types of clustering:

1) K-means clustering
2) Agglomerative
3) EM for Gaussian mixture

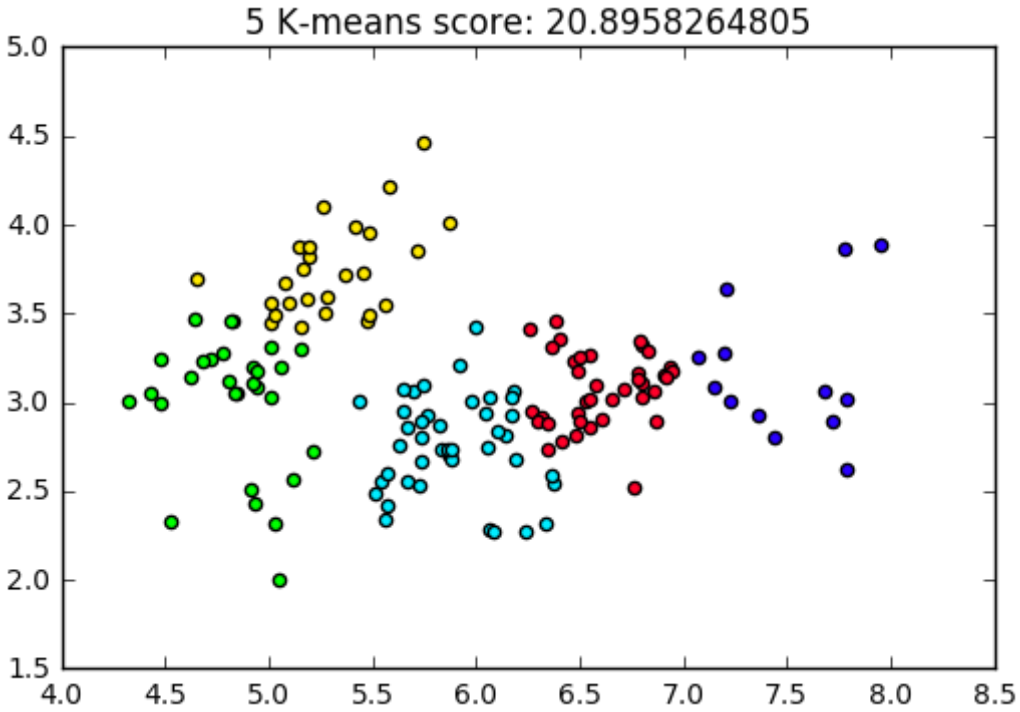
(a) Plotting the data

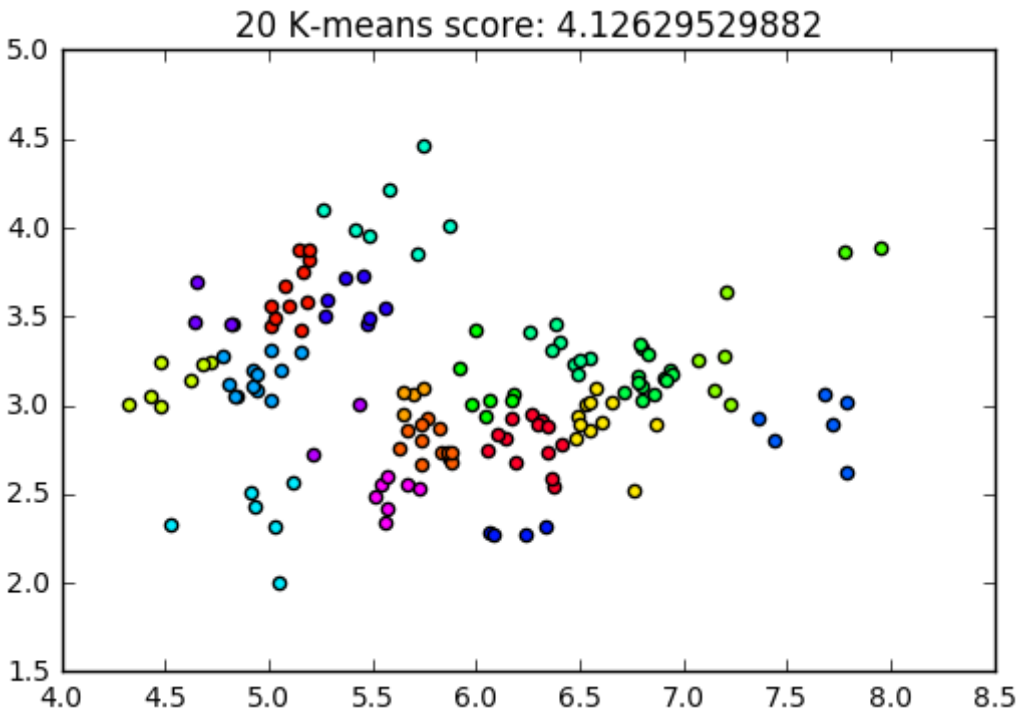
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import mltools as ml

iris = np.genfromtxt("data/iris.txt", delimiter = None )
X1 = iris[:,0:1]
X2 = iris[:,1:2]

fig1, xy1 = plt.subplots()
xy1.scatter(X1, X2)
xy1.set_title("Iris Scatter: F1 & F2")
plt.show()
```



(b) K-means

In [150]:
```
Xk = iris[:,0:2]
knumb = [5,20]
for k in knumb:
    numb_colors = k
    cm = plt.get_cmap('gist_rainbow')
    k_means = ml.cluster.kmeans(Xk, k)
    figK, xyK = plt.subplots()
    Cr = [cm(1.*i/numb_colors) for i in range(numb_colors)]
    xyK.set_title('{} K-means score: {}'.format(k, k_means[2]))
    for a,b in enumerate(k_means[0]):
        xyK.scatter(X1[a], X2[a], c = Cr[int(b)])
    plt.show()
```

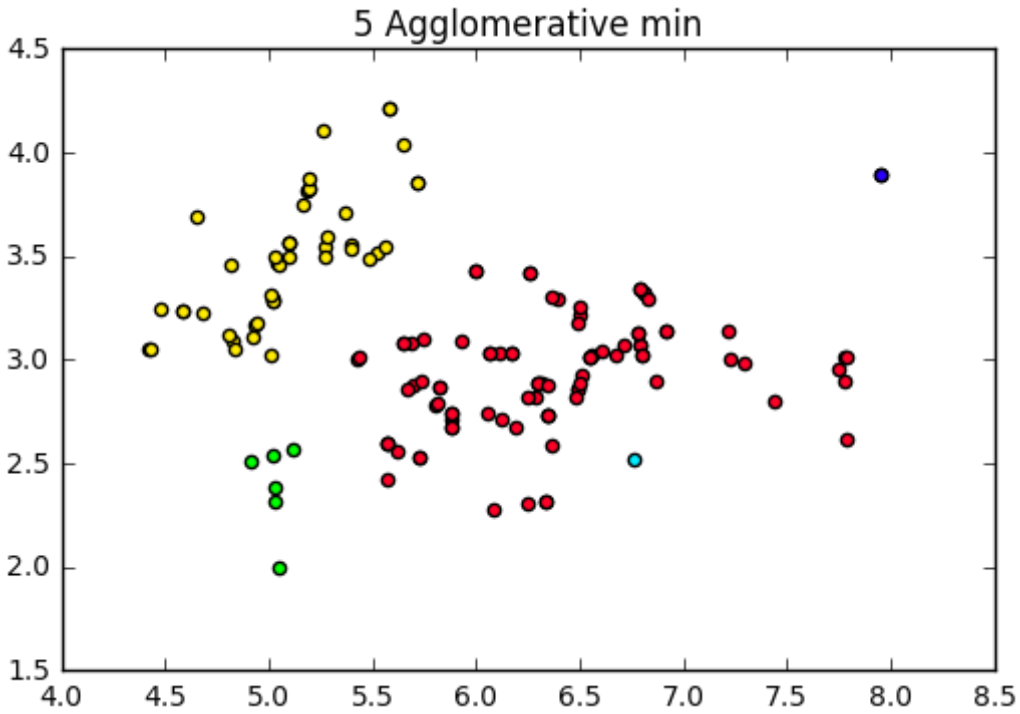


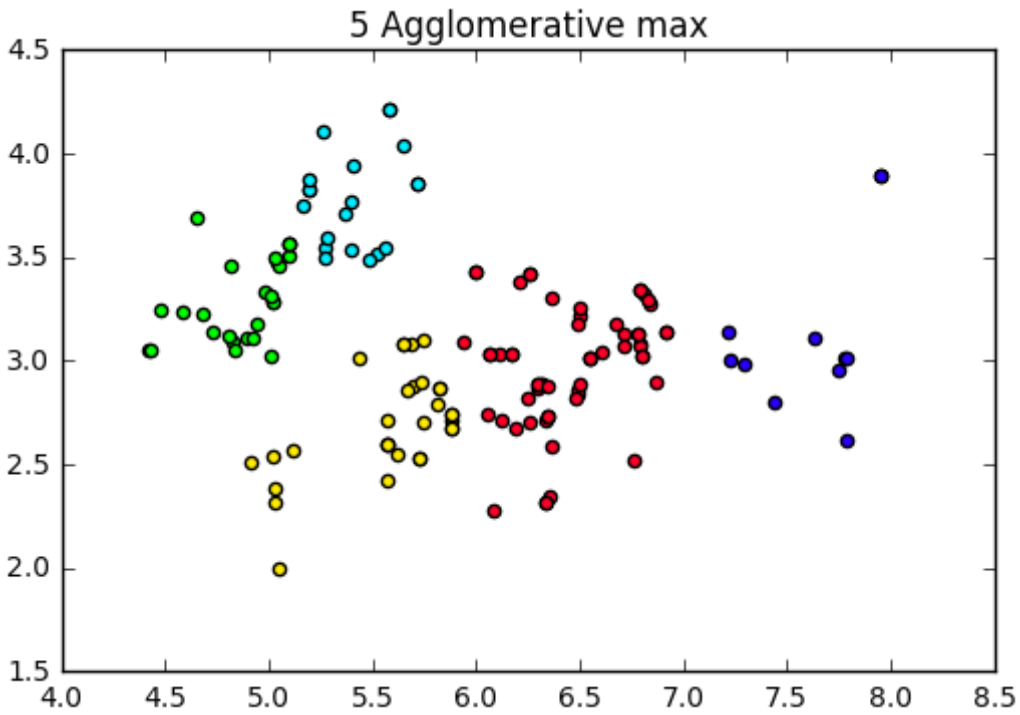


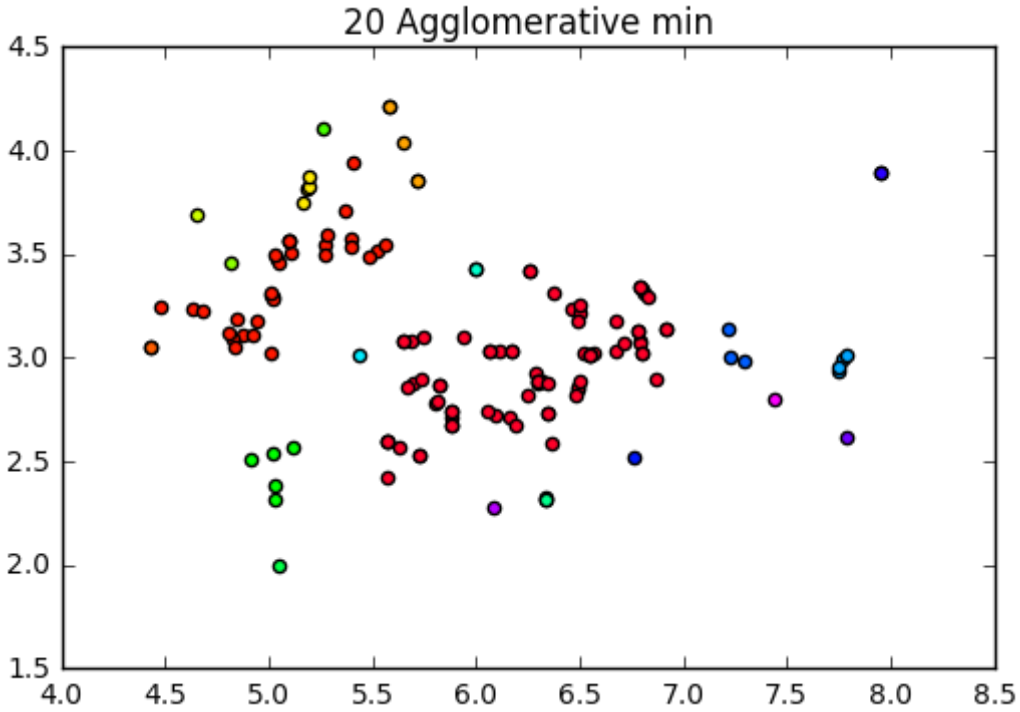
You can see that the sum of the square euclidean distances decreases when K gets larger. It is interesting in that 20 is 4 times larger than 5 and the sum of the square euclidean distances almost mirrors that factor. I would predict that if K was two, then the clusters would be one on the top and one on the bottom. When k=5, you can clearly see separation of each cluster, on the other hand, when k = 20, it almost seems like there are too few data poitns to categorize 20 clusters. I would like to mention that when k = 20, it does give us more clusters to work with if k=5 was too few.

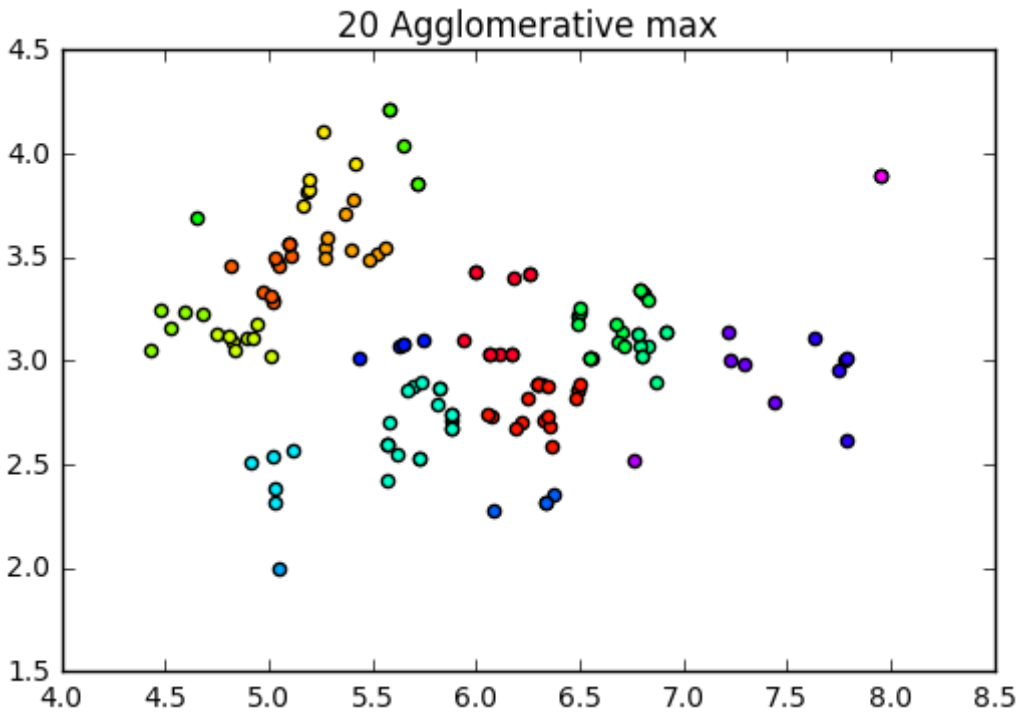
(c) Agglomerative

In [163]:
```
knumb = [5,20]
Amethod = ['min','max']
Xa = iris[:,0:2]
for k in knumb:
    for m in Amethod:
        numb_colors = k
        cm = plt.get_cmap('gist_rainbow')
        Cr = [cm(1.*i/numb_colors) for i in range(numb_colors)]
        agglo = ml.cluster.agglomerative(Xa, k, method = m)
        figA, xyA = plt.subplots()
        xyA.set_title('{} Agglomerative {}'.format(k, m))
        for a,b in enumerate(agglo[0]):
            xyA.scatter(X1[a], X2[a], c = Cr[int(b)])
        plt.show()
```


5 Agglomerative min
4.5
4.0
3.5
3.0
2.5
2.0
1.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5


5 Agglomerative max
4.5
4.0
3.5
3.0
2.5
2.0
1.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5





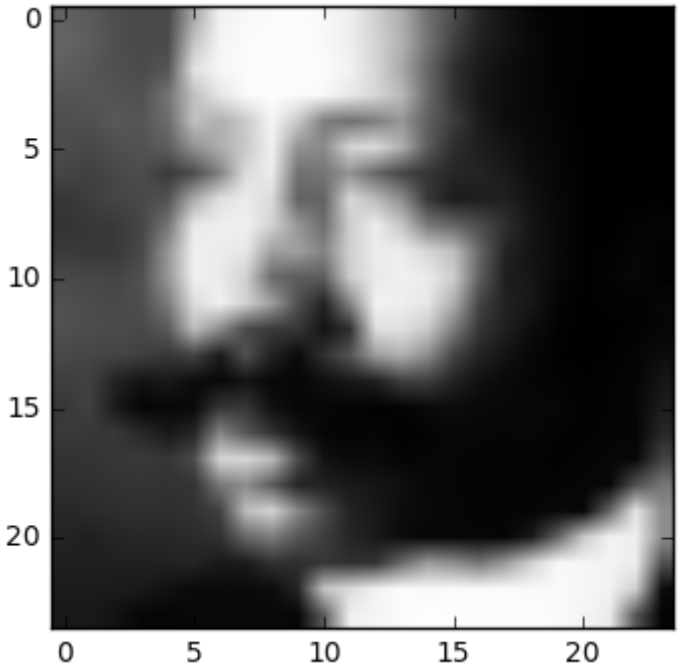
In both cases, using a single linkage seemed to make the clustering seem more bias and made the graph largely one color. This is due to the fact that when we do a single linkage (min) the clustering algorithm doesn't a lot of options and is bound by the first choice. On the other hand, when we use complete linkage, the graphs look more evenly distributed. The usage of k =5 or k =20 seems to just be in the number of clusters and not much else. Having a different sized k may help in other application suhc as shrinking the size of extrememly large data sets.

K-means vs Agglomerative

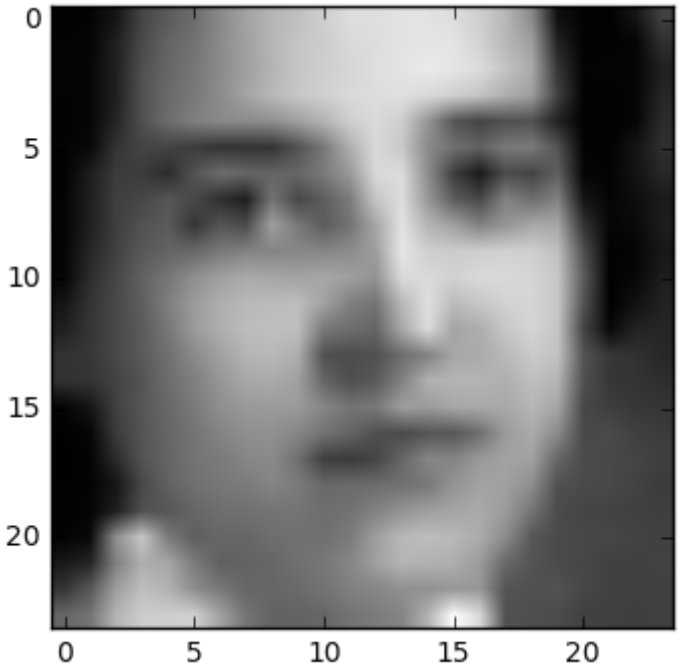
The main difference I see between the graphs I provided is that k-means tends to have tighter clusters of data. This is more noticable when k=20. We can see few data points that are spread out a bit more in Agglomerative than in K-means. And this could be a consequence of k-means having a random factor into its initialization where as Agglomerative is more deterministic.

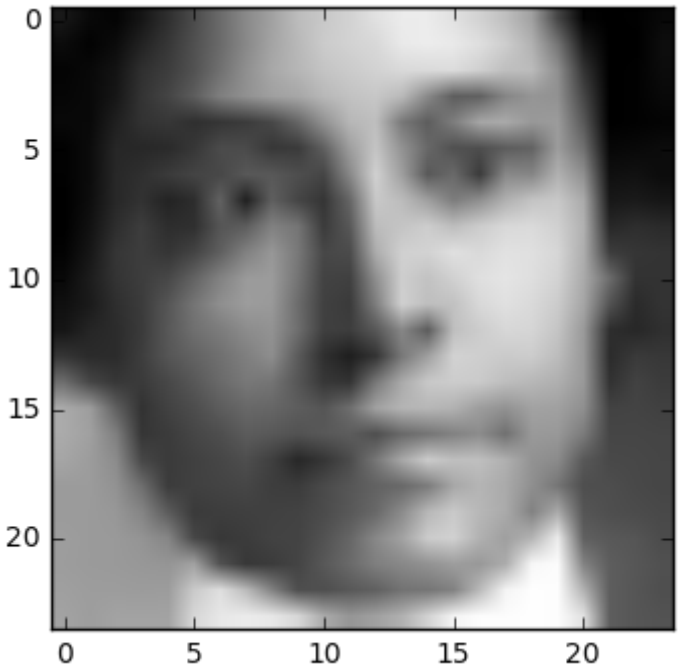
Problem 2: EigenFaces

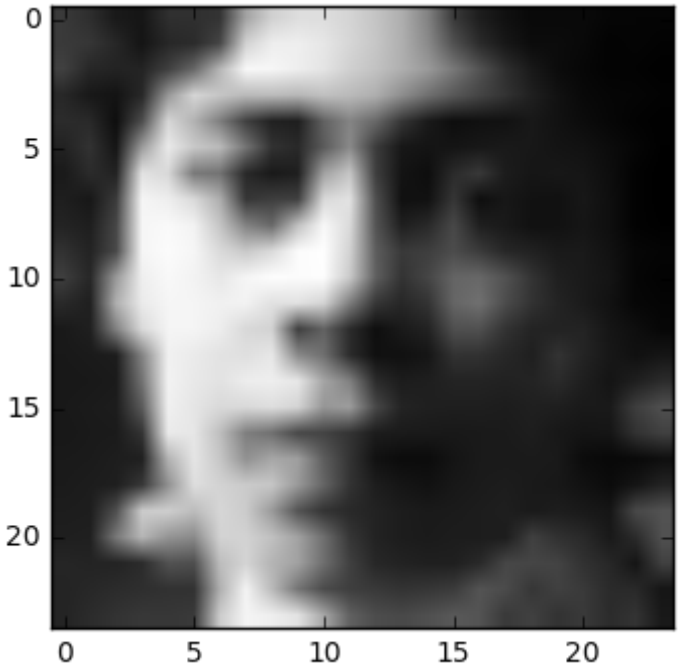
In [4]:
```
Xe = np.genfromtxt("data/faces.txt", delimiter=None)
for x in range(10):
    plt.figure()
    img = np.reshape(Xe[x,:],(24,24))
    plt.imshow( img.T , cmap="gray")
    plt.show()
```

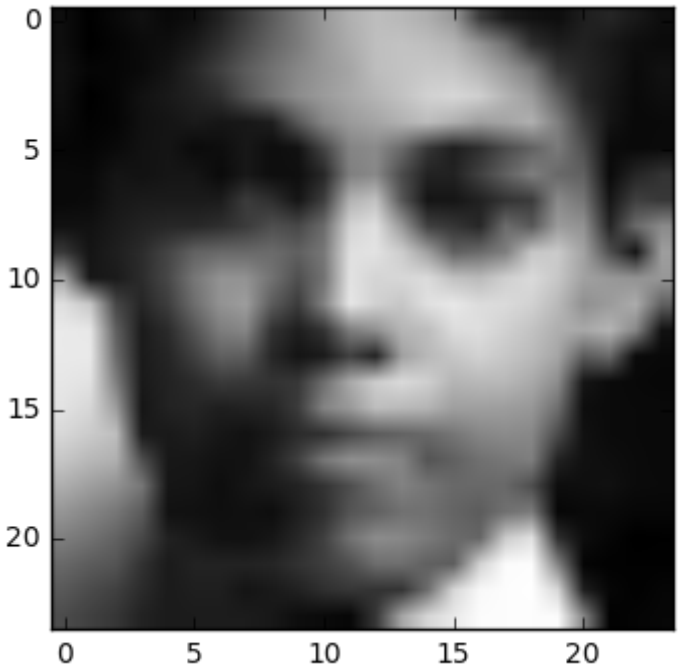

0
5
10
15
20
0
5
10
15
20

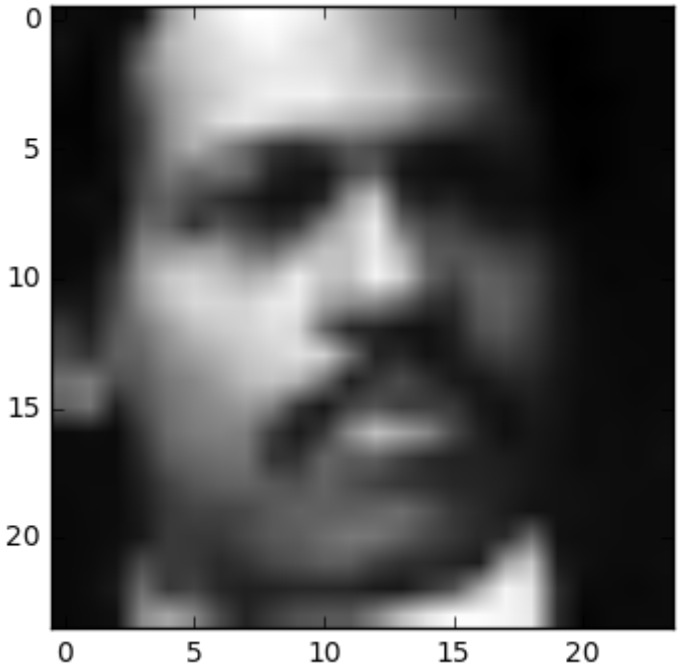

0
5
10
15
20
0
5
10
15
20

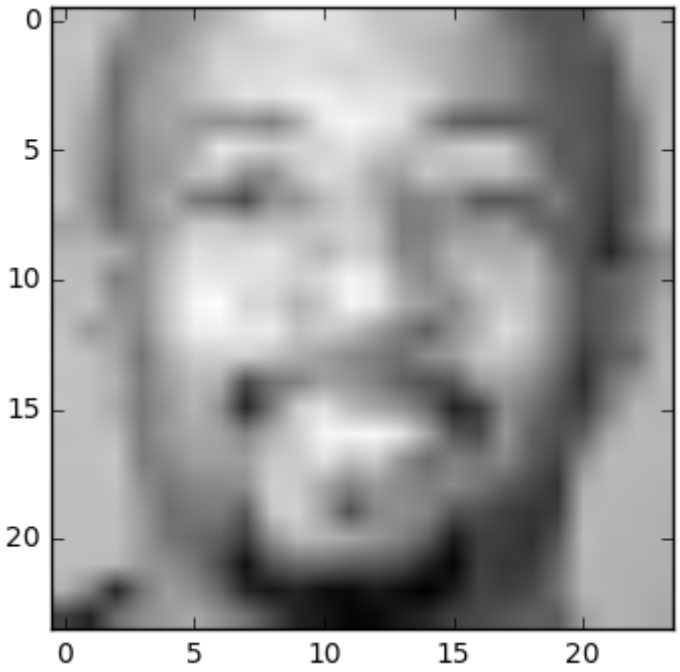

0
5
10
15
20
0
5
10
15
20

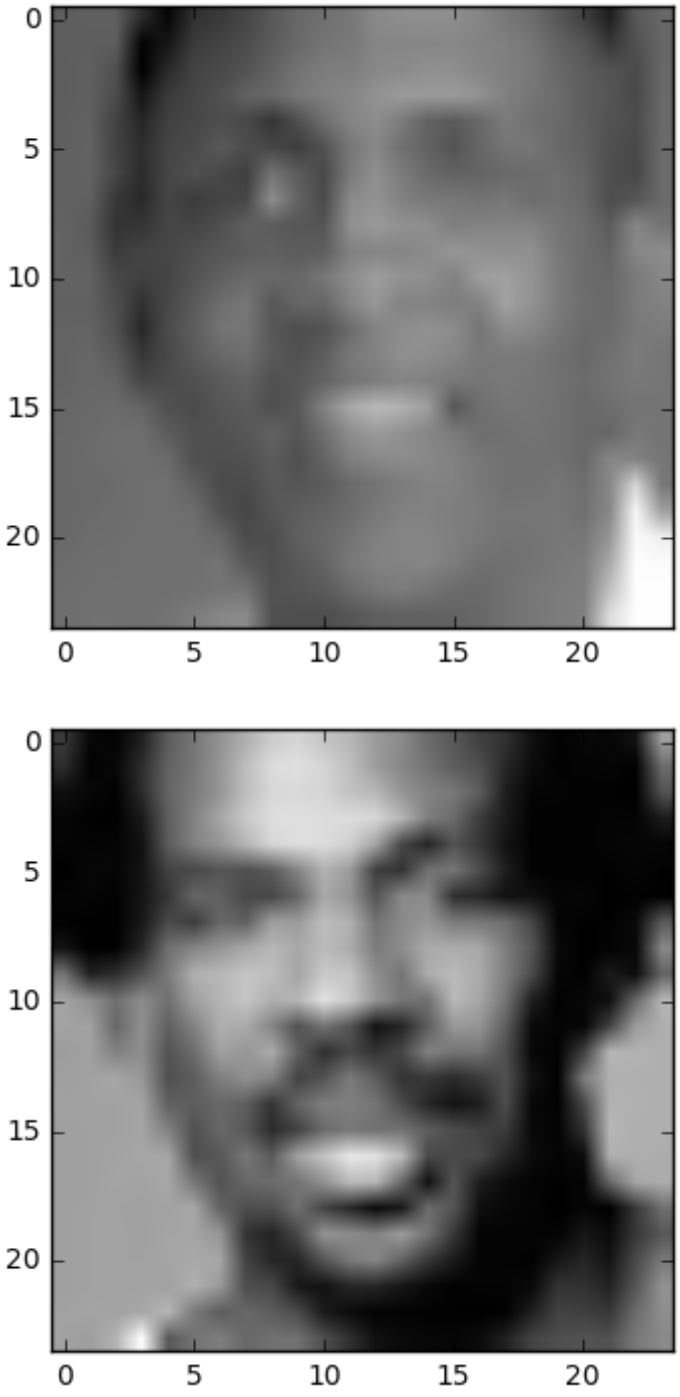

0
5
10
15
20
0
5
10
15
20


0
5
10
15
20
0
5
10
15
20


0
5
10
15
20
0
5
10
15
20


0
5
10
15
20
0
5
10
15
20


0
5
10
15
20
0
5
10
15
20



The data has 4916 faces. Each face uses 576 numbers in order to generate the face.

(a)

In [14]:
```
import scipy.linalg as sl
mu = np.mean( Xe, axis=0, keepdims=True )
X0 = Xe - mu
print(X0)
```

```
[[   9.95240033   15.13161107   10.03254679 ...,  -18.58136697
   -84.98494711  -94.25386493]
 [ -64.04759967  -65.86838893  -61.96745321 ...,  -19.58136697
   -57.98494711  -97.25386493]
 [ -80.04759967  -76.86838893  -74.96745321 ...,  -35.58136697
   -38.98494711  -40.25386493]
 ...,
 [ -66.04759967  -64.86838893  -64.96745321 ...,  -72.58136697
   -71.98494711  -68.25386493]
 [ -21.04759967  -17.86838893  -15.96745321 ...,  -69.58136697
   -70.98494711  -72.25386493]
 [ -29.04759967  -30.86838893  -28.96745321 ...,  152.41863303
   150.01505289  149.74613507]]
```

This bit of code takes the mean of each face and subtracts each number with the mean.

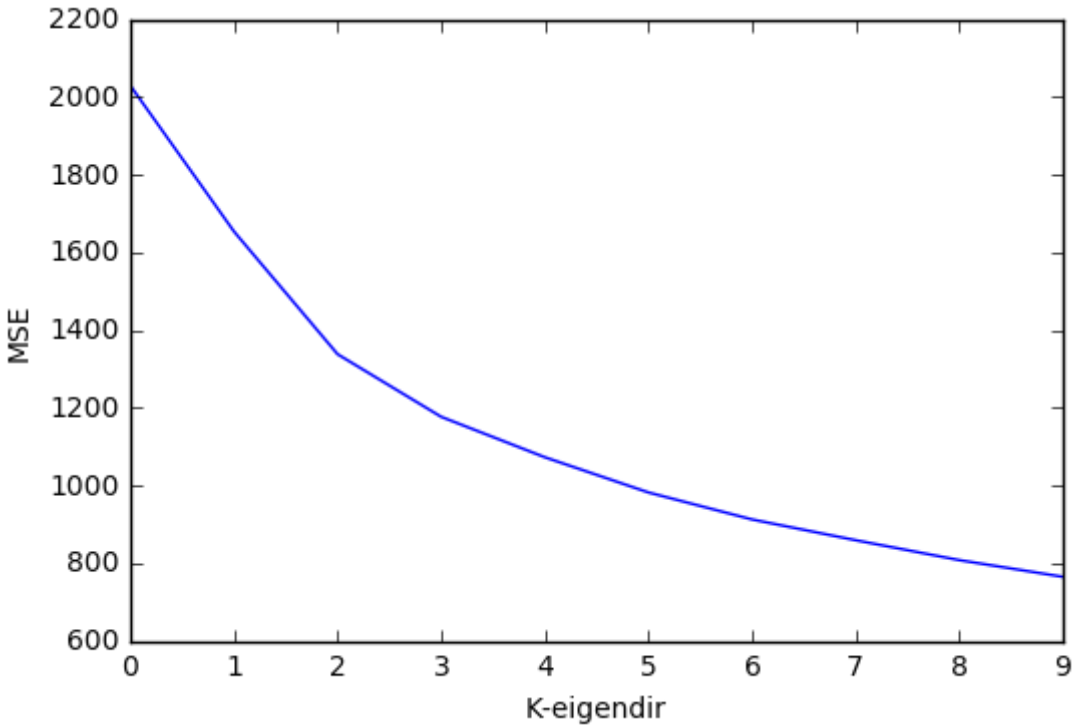
(b)

In [23]:
```
U,S,Vh = sl.svd(X0, False)
```

```
(4916L, 576L)
```

(c)

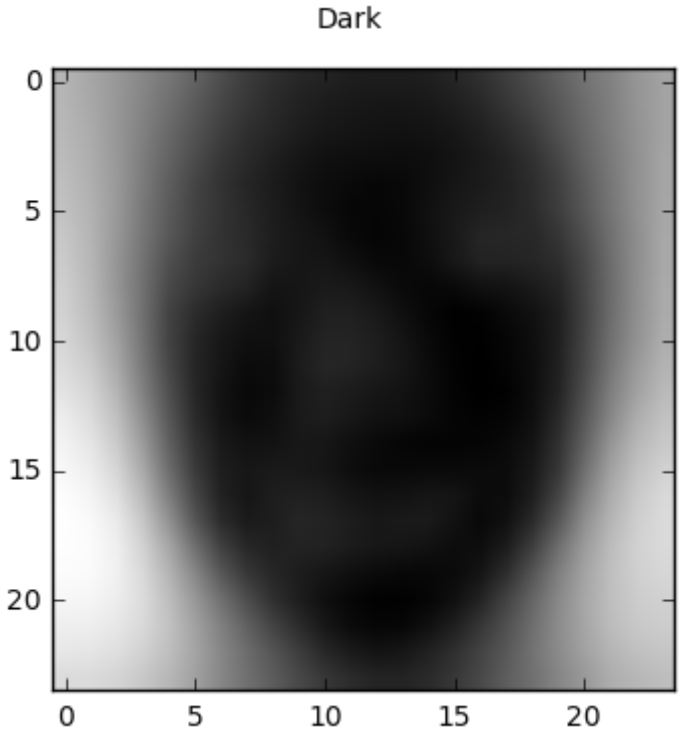
In [38]:
```
K_eigendir = []
for x in range(1,11):
    Xhat = U[:,0:x].dot( np.diag(S[0:x]) ).dot( Vh[0:x,:] )
    K_eigendir.append(np.mean( (X0 - Xhat)**2))
plt.plot(K_eigendir)
plt.ylabel('MSE')
plt.xlabel('K-eigendir')
plt.show()
```

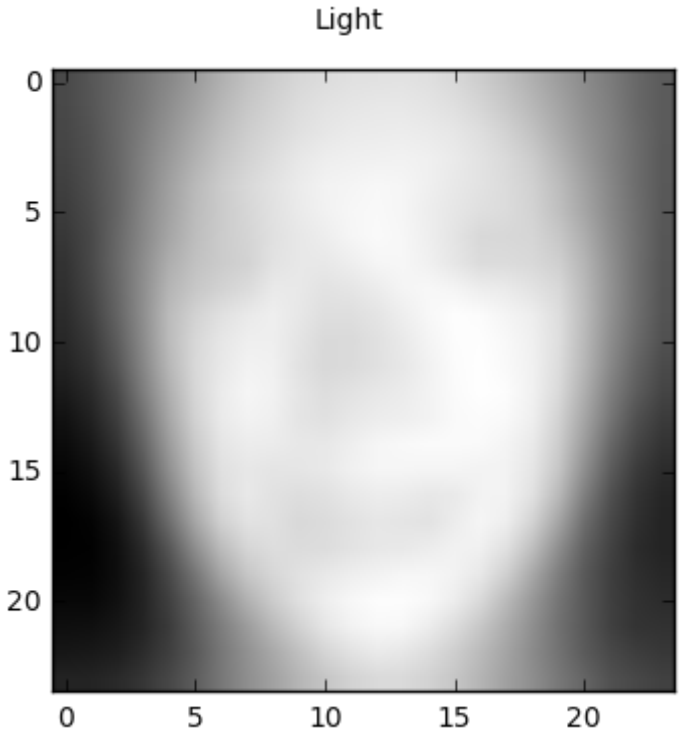


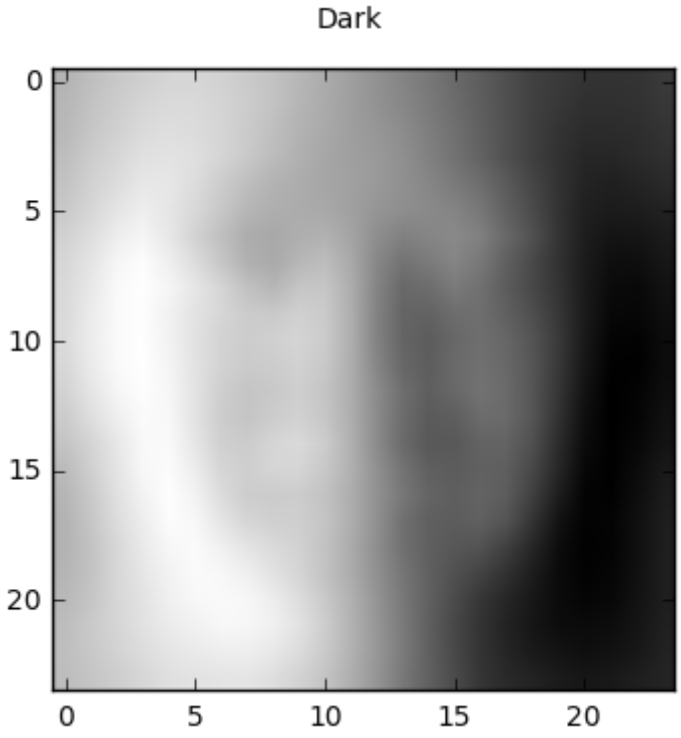
From the graph, you can see that as the k increases, the mean squared error decreases. k=0 to k=1 had the largest decrease in error. It seems like with this set of data, we can go a bit lower to reduce the MSE and it might start to plateu around 700.

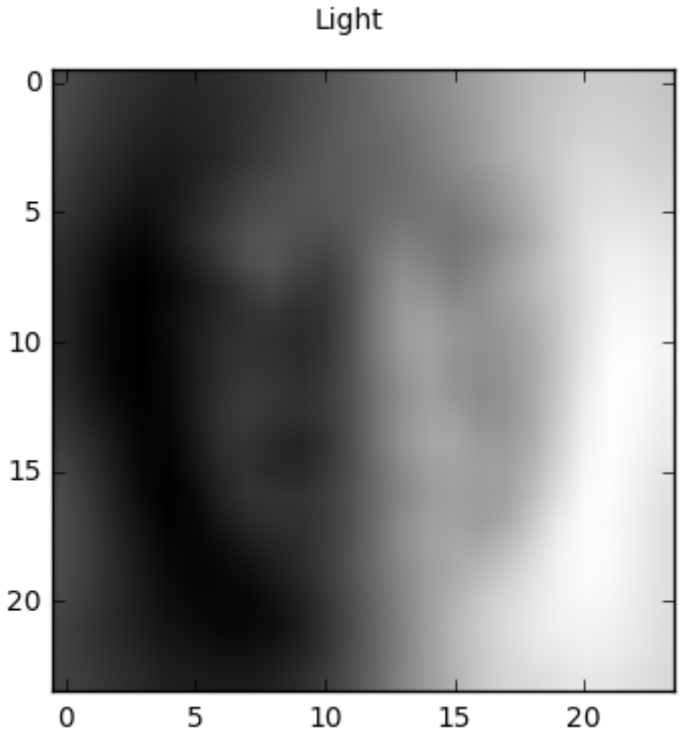
(d)

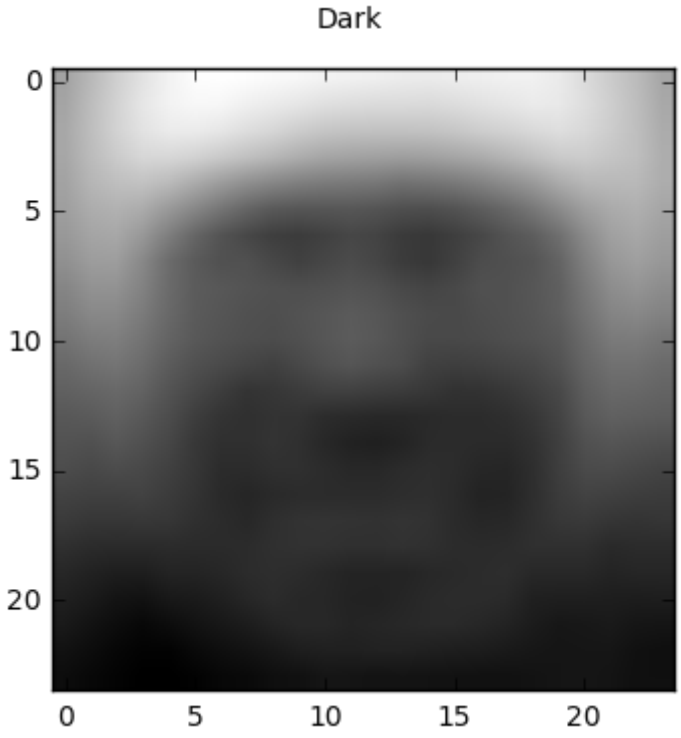
In [46]:
```
for i in range(1,4):
    alpha = 2*np.median(np.abs(W[:,i]))
    plus = U + (alpha * Vh[i,:])
    minus = U - (alpha * Vh[i,:])
    plt.figure()
    img = np.reshape(plus[i,:],(24,24))
    plt.imshow( img.T , cmap="gray")
    plt.suptitle("Dark")
    plt.show()
    img = np.reshape(minus[i,:],(24,24))
    plt.imshow( img.T , cmap="gray")
    plt.suptitle("Light")
    plt.show()
```

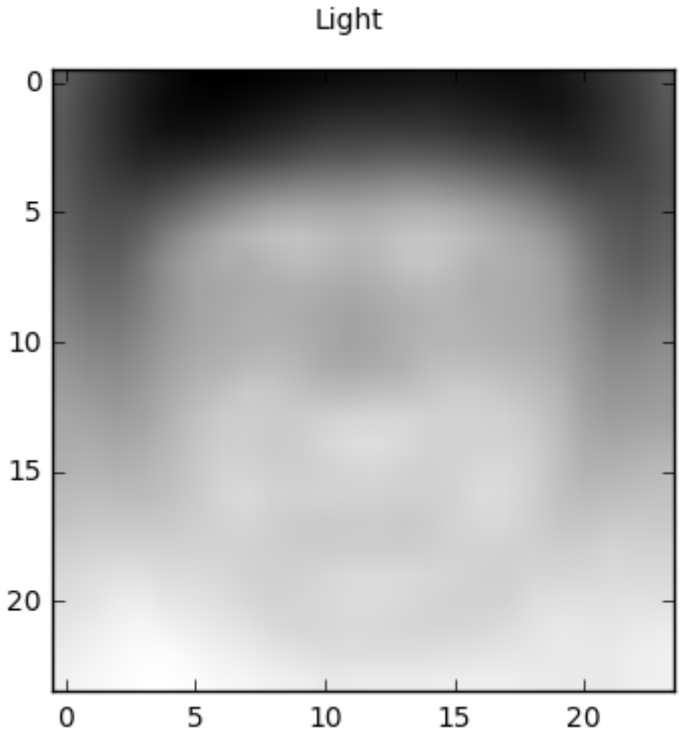

Dark
0
5
10
15
20
0
5
10
15
20


Light
0
5
10
15
20
0
5
10
15
20


Dark
0
5
10
15
20
0
5
10
15
20


Light
0
5
10
15
20
0
5
10
15
20



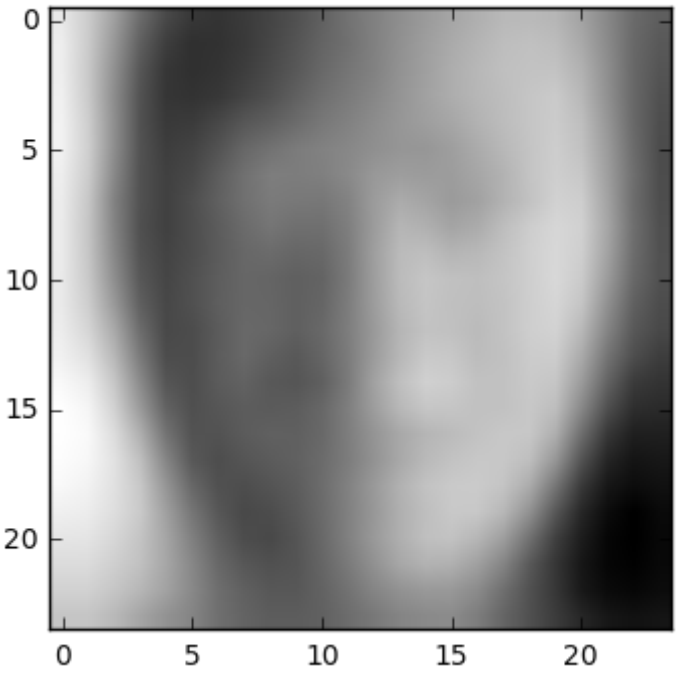


So taking the zero mean of the data allows us to control the darkness and brightness contrast of the images. Positive being darker faces, and negative meaning brighter faces. The value of alpha allows us to get rid of most of the different shades, turning the images into mostly black and white.

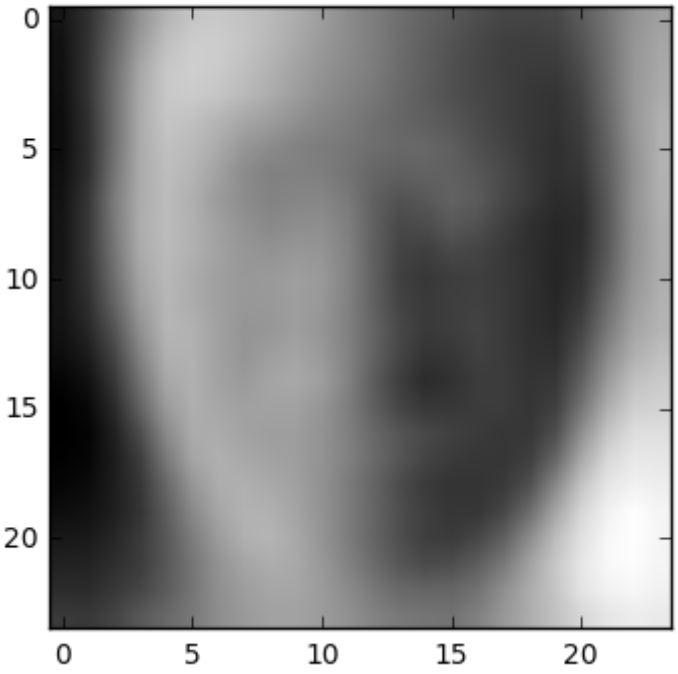
(e)

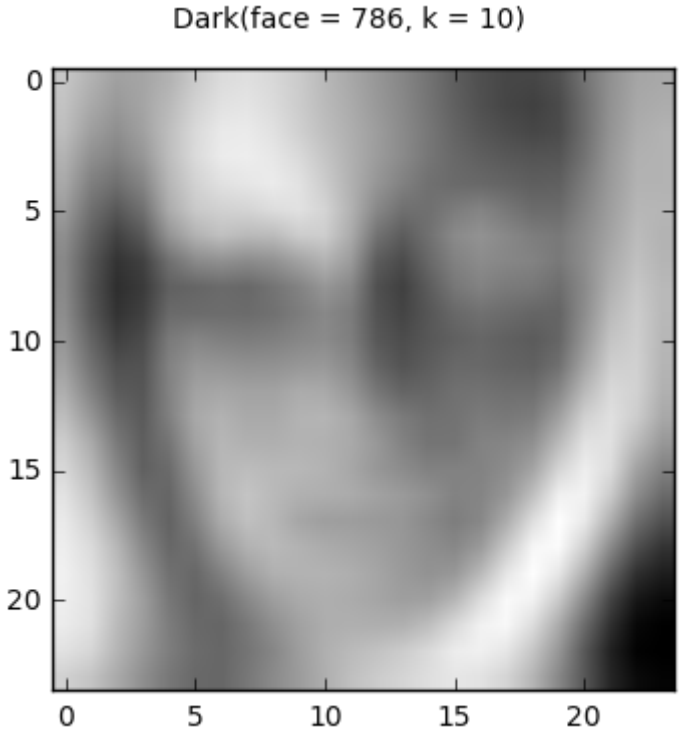
In [51]:
```
faces = [786, 356]
K_directions = [5,10,50,100]
for j in faces:
    for i in K_directions:
        alpha = 2*np.median(np.abs(W[:,i]))
        plus = U + (alpha * Vh[i,:])
        minus = U - (alpha * Vh[i,:])
        plt.figure()
        img = np.reshape(plus[j,:],(24,24))
        plt.imshow( img.T , cmap="gray")
        plt.suptitle("Dark(face = {}, k = {})".format(j, i))
        plt.show()
        img = np.reshape(minus[j,:],(24,24))
        plt.imshow( img.T , cmap="gray")
        plt.suptitle("Light(face = {}, k = {})".format(j, i))
        plt.show()
```

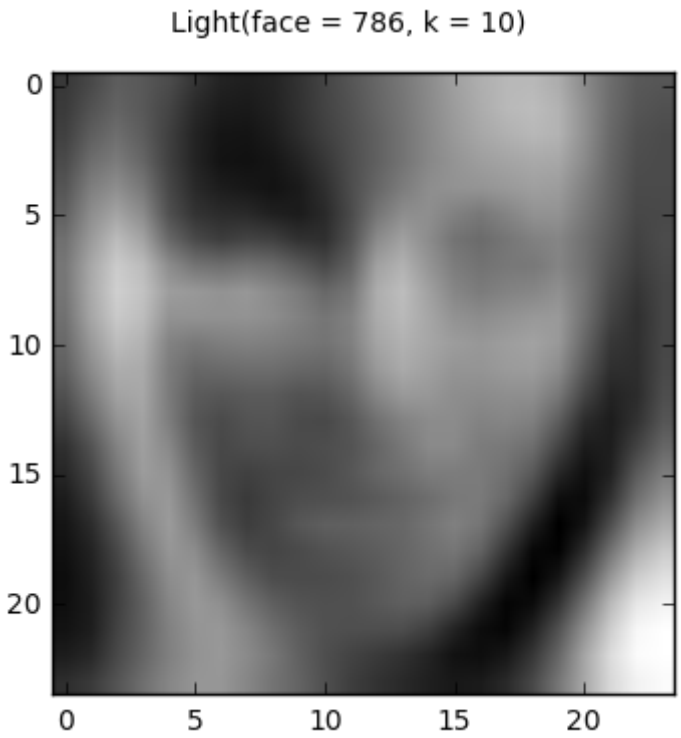
Dark(face = 786, k = 5)

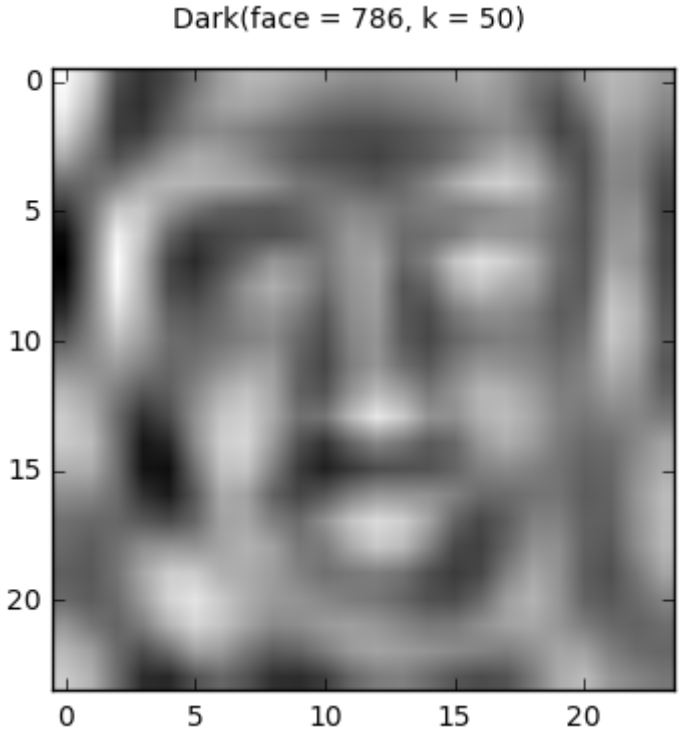


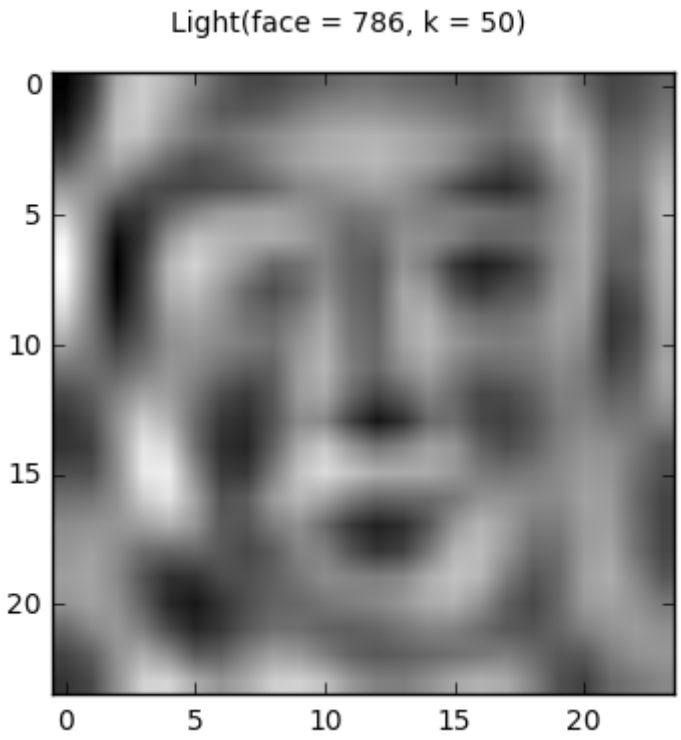
Light(face = 786, k = 5)

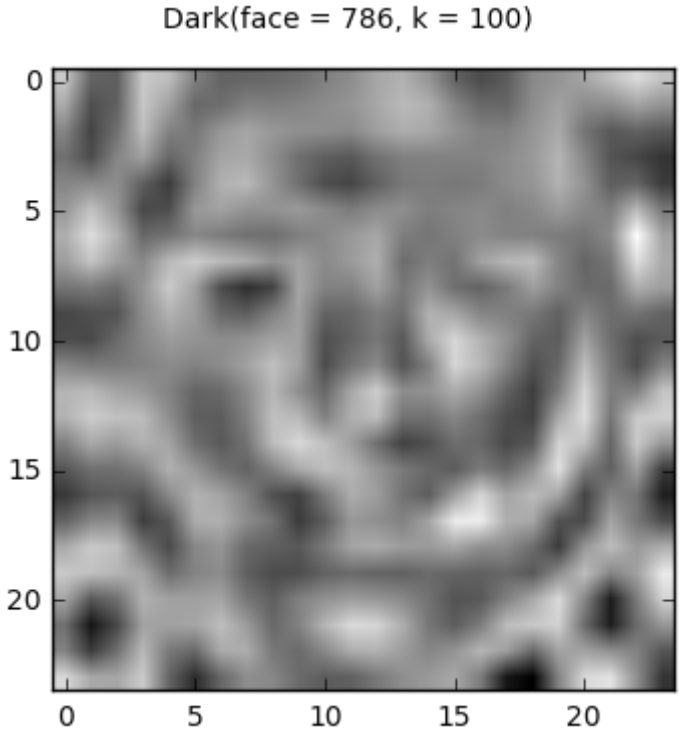


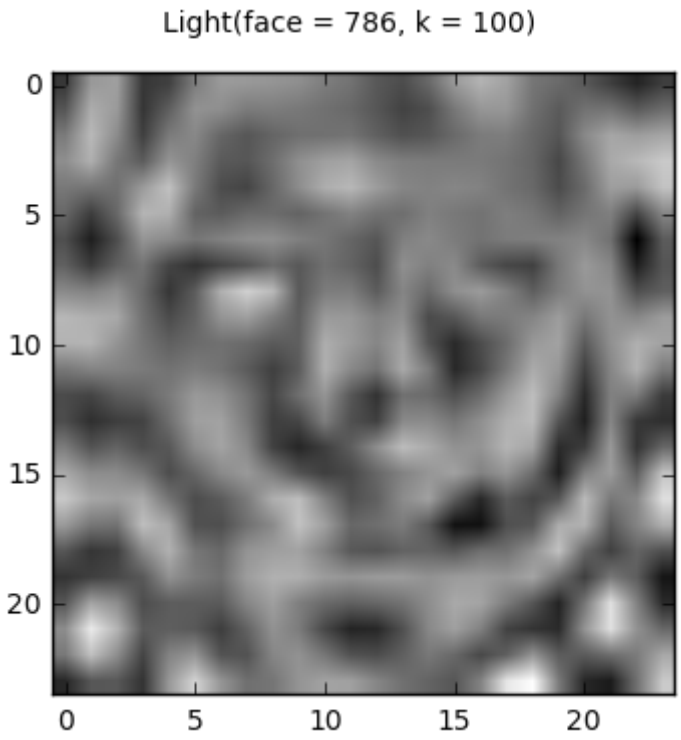

Dark(face = 786, k = 10)
0
5
10
15
20
0
5
10
15
20

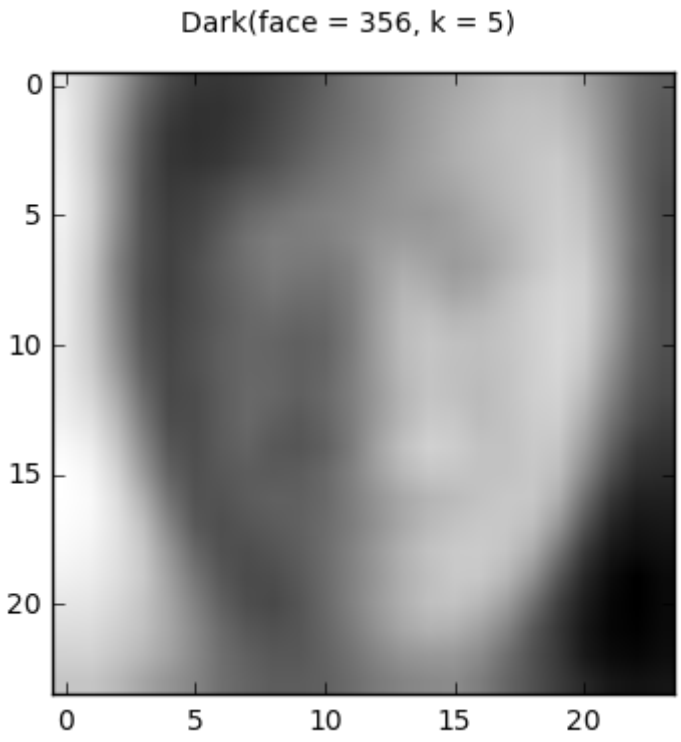

Light(face = 786, k = 10)
0
5
10
15
20
0
5
10
15
20

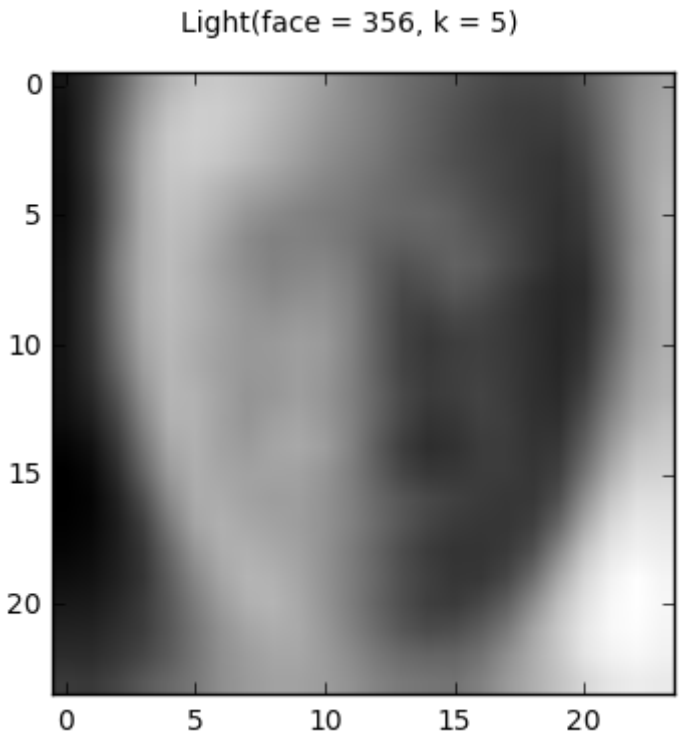

Dark(face = 786, k = 50)
0
5
10
15
20
0
5
10
15
20

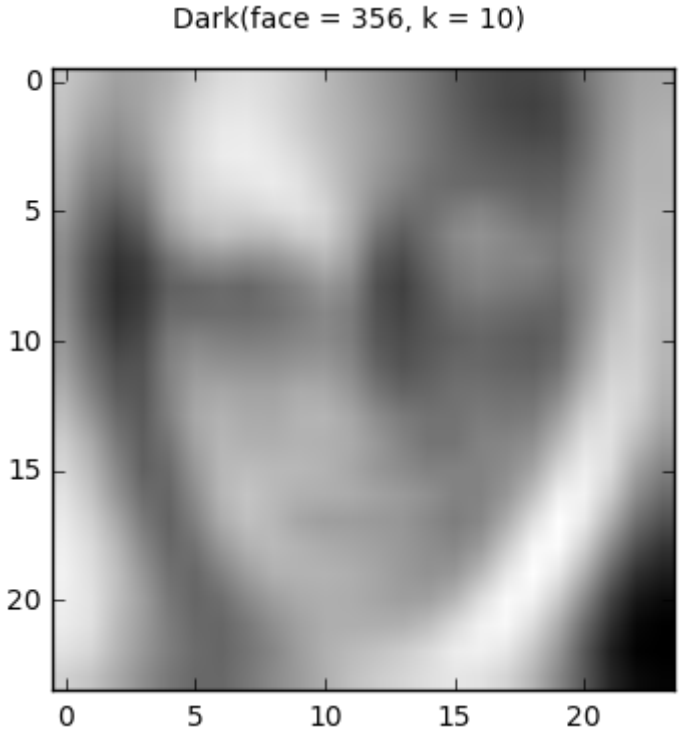

Light(face = 786, k = 50)
0
5
10
15
20
0
5
10
15
20

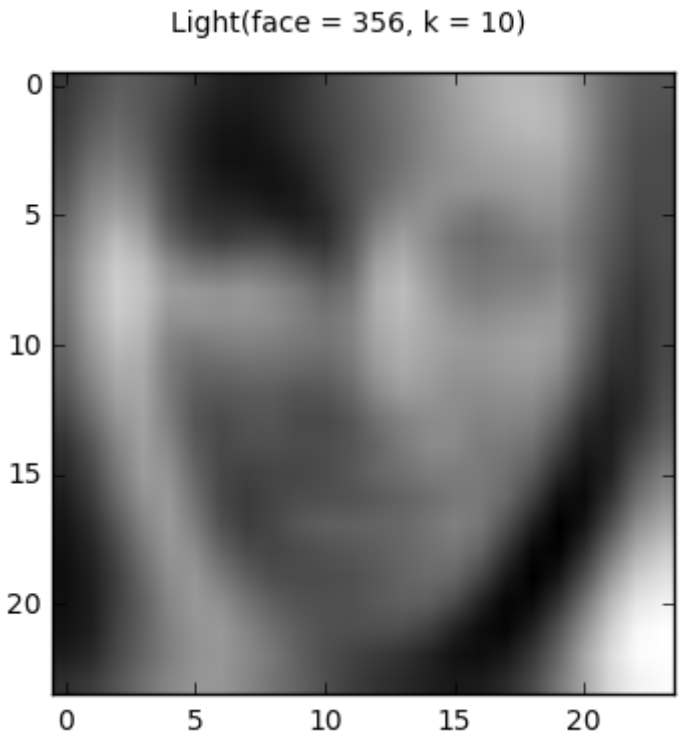

Dark(face = 786, k = 100)
0
5
10
15
20
0
5
10
15
20

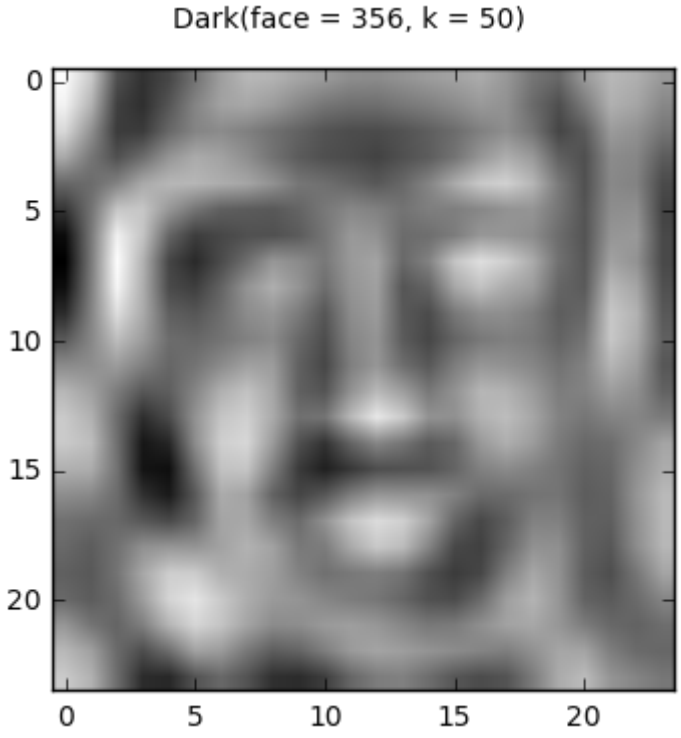

Light(face = 786, k = 100)
0
5
10
15
20
0
5
10
15
20

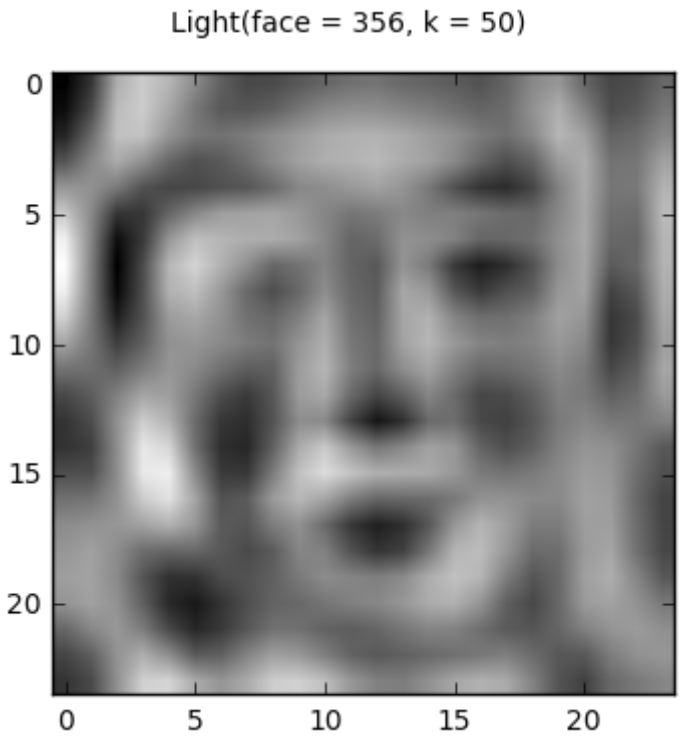

Dark(face = 356, k = 5)
0
5
10
15
20
0
5
10
15
20

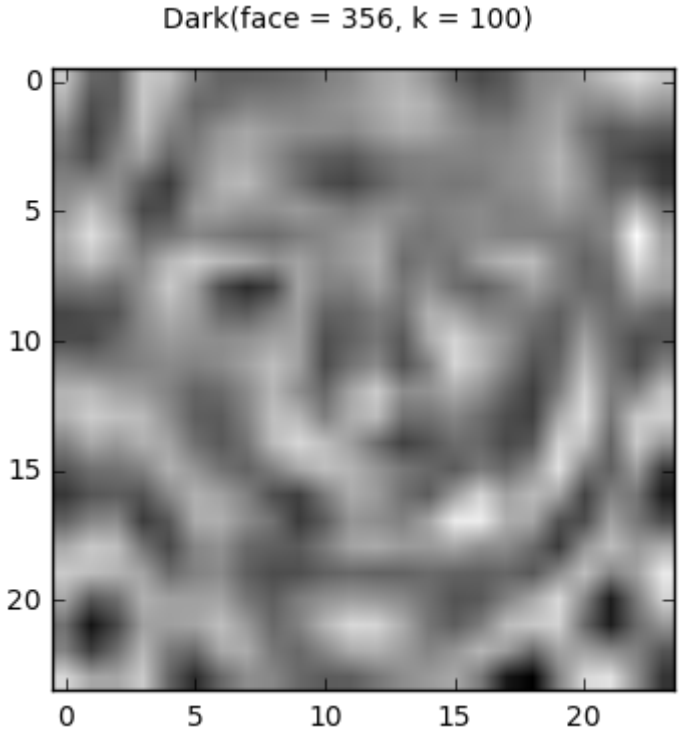

Light(face = 356, k = 5)
0
5
10
15
20
0
5
10
15
20

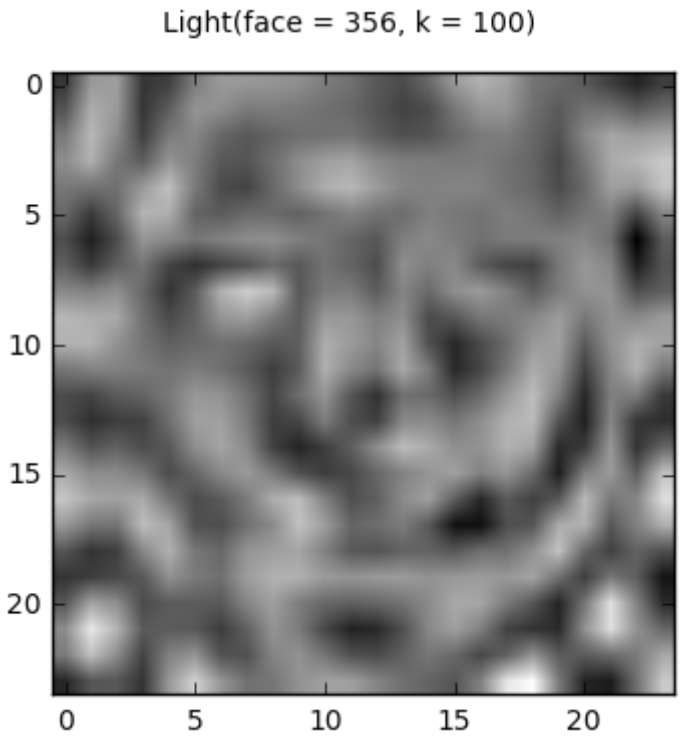

Dark(face = 356, k = 10)
0
5
10
15
20
0
5
10
15
20


Light(face = 356, k = 10)
0
5
10
15
20
0
5
10
15
20


Dark(face = 356, k = 50)
0
5
10
15
20
0
5
10
15
20


Light(face = 356, k = 50)
0
5
10
15
20
0
5
10
15
20

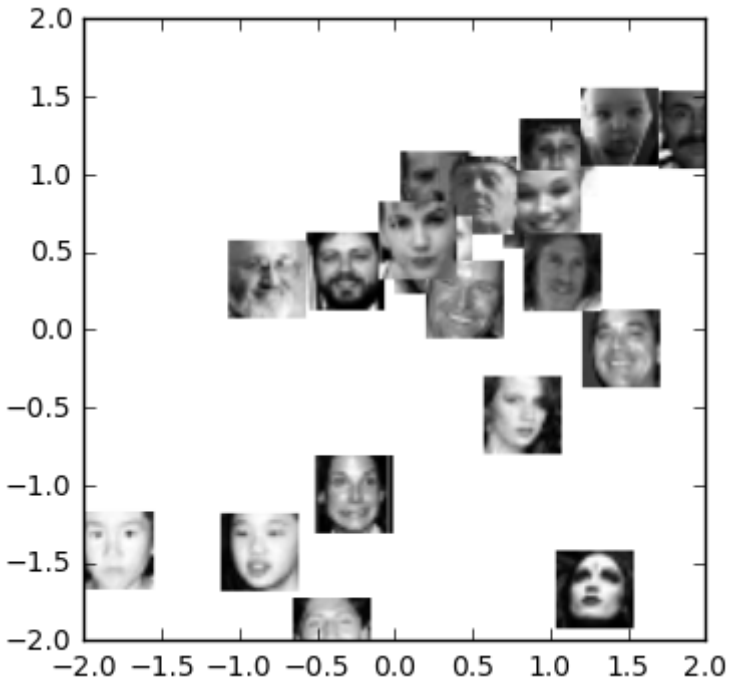




With hight k values, the MSE seems to have decreased, but the resulting face images have become heavily distorted. This is probably due to overfitting, when the K is at a very high value, then the eigen tries to make too many clusters. Thats why in the faces where k = 100, there hardly seems to be a face, instead clusters of white and black checker spots.

(f)

In [74]:
```
idx = [int(np.random.random() * 4916) for x in range(20) ]
import mltools.transforms
coord,params = ml.transforms.rescale( W[:,0:2] ) # normalize scale of "W" loca
tions
plt.figure(); plt.hold(True); # you may need this for pyplot
for i in idx:
    # compute where to place image (scaled W values) & size
    loc = (coord[i,0],coord[i,0]+0.5, coord[i,1],coord[i,1]+0.5)
    img = np.reshape( Xe[i,:], (24,24) ) # reshape to square
    plt.imshow( img.T , cmap="gray", extent=loc ) # draw each image
    plt.axis( (-2,2,-2,2) ) # set axis to reasonable visual scale
plt.show()
```



It seems like the Latent representation is trying to plot faces on a scale where at the top right are the darkest images, and the bottom left are the lightest images. Images that fall within the center have a fair balance between dark and light.