In [5]:
```
import numpy as np #numerical computing module
import matplotlib.pyplot as plt #plotting module
#when using a notebook, the line below will display any plots directly in it:
%matplotlib inline
from __future__ import division
from PIL import Image
import numpy.fft as fft
from scipy import signal
import scipy.io
from mpl_toolkits.mplot3d import Axes3D
#from kmeans import *
```

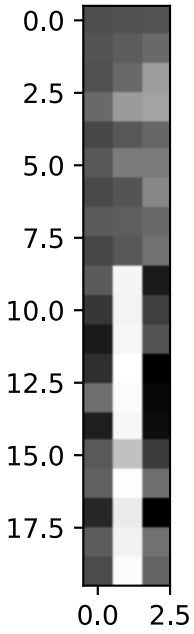
In [15]:
```
from __future__ import print_function
import numpy as np

def kmeans(X, k, init=None):
    """ KMEANS implements the k-means algorithm.

    [CLUSTERS, CENTROIDS] = KMEANS(X, K) partitions the data points
    in the N-by-P data matrix X into K distinct clusters, using Euclidean
    distance. This is a simple implementation of the k-means algorithm with
    random initialization.

    Optionally, it takes the argument INIT, a K-by-P matrix with a fixed
    initial position for the cluster centroids.

    MYKMEANS returns an N-by-1 vector CLUSTERS containing the cluster
    indices of each data point, as well as CENTROIDS, a K-by-P matrix with
    the final cluster centroids' locations.
    """



    n, p = X.shape

    if init is None:
      #choose initial centroids by picking k points at random from X
      init = X[np.random.randint(n, size=k), :]

    #centroids is a k-by-p random matrix
    #its i^th row contains the coordinates of the cluster with index i
    centroids = init

    #initialize cluster assignment array
    clusters = np.zeros(n)

    MAXITER = 1000

    for iter in range(MAXITER):

        #create a new clusters vector to fill in with updated assignments
        new_clusters = np.zeros(n)


        #for each data point x_i
        for i in range(n):

            x_i = X[i,:]

            #find closest cluster
            closest = findClosestCluster(x_i,centroids)###IMPLEMENT THIS FUNCT
ION AT THE END OF THIS FILE

            #reassign x_i to the index of the closest centroid found
            new_clusters[i] = closest


        if hasConverged(clusters,new_clusters):###IMPLEMENT THIS FUNCTION AT T
HE END OF THIS FILE
```

```
            #exit loop
            break


        #otherwise, update assignment
        clusters = new_clusters
        #and recompute centroids
        centroids = recomputeCentroids(X,clusters,k)###IMPLEMENT THIS FUNCTION
AT THE END OF THIS FILE



    if iter == (MAXITER-1):
        print('Maximum number of iterations reached!')

    return clusters, centroids


def findClosestCluster(x_i,centroids):
    # Compute Euclidean distance from x_i to each cluster centroid and return
    # the index of the closest one (an integer).
    # NOTE: use of numpy/scipy.linalg.norm function is NOT allowed here.

    ### Replace the following line with your own code
    dists = np.sum(((x_i - centroids)**2), axis=1)
    closest = np.argmin(dists)

    return closest



def hasConverged(old_assignment, new_assignment):
    # Check if algorithm has converged, i.e., cluster assignments haven't
    # changed since last iteration. Return a boolean.

    ### Replace the following line with your own code

    return np.array_equal(old_assignment, new_assignment)



def recomputeCentroids(X,clusters,k):
    # Recompute centroids based on current cluster assignment.
    # Return a k-by-p array where each row is a centroid.
    n, p = X.shape
    ### Replace the following line with your own code
    centroids = np.zeros((k,p))
    # initialize centroids to 0 and then check each data point and then divide
by number of points

    for cl in range(k):
        centroids[cl] = X[clusters == cl].sum(0) / (clusters == cl).sum()


    # for i in range(k):
    #     w, u = np.shape(X[clusters == i,:])
    #     centroids[i] = np.divide(X[clusters == i,:].sum(0), w)
```

```
    return centroids
```

In [16]:
```
images = scipy.io.loadmat('tinypics.mat')['data']
plt.imshow(images, cmap='gray')
plt.axis('image')
images
```

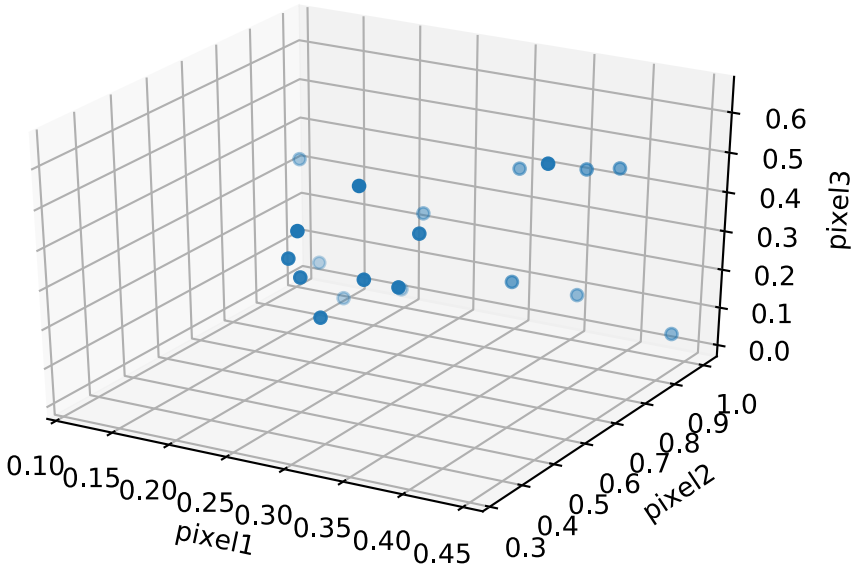
Out[16]:
```
array([[0.31828322, 0.32790572, 0.34036871],
       [0.34106572, 0.37444091, 0.42427947],
       [0.32392653, 0.42208296, 0.61935186],
       [0.42554799, 0.61195239, 0.64189935],
       [0.29444769, 0.35322588, 0.41348794],
       [0.3548847 , 0.48795843, 0.49113724],
       [0.29696706, 0.33869869, 0.53448631],
       [0.36867035, 0.37751132, 0.4196667 ],
       [0.28420632, 0.35499289, 0.45314331],
       [0.36516543, 0.95785251, 0.11864179],
       [0.22942441, 0.95468057, 0.26056792],
       [0.11579719, 0.96240601, 0.33956777],
       [0.19775761, 0.9987982 , 0.01886943],
       [0.442584  , 0.97961839, 0.04935614],
       [0.13093559, 0.9679728 , 0.06827657],
       [0.36061375, 0.75676186, 0.24708697],
       [0.38952586, 0.99037206, 0.44546125],
       [0.16708153, 0.91978098, 0.01527047],
       [0.37203713, 0.94799221, 0.45236112],
       [0.30493332, 0.98594423, 0.40274471]])
```



1a.

In the first nine images, the pixels are much more closely related in color as compared to the last 11. In the last 11, the middle of the images are very light as compared to the edges which are super dark.

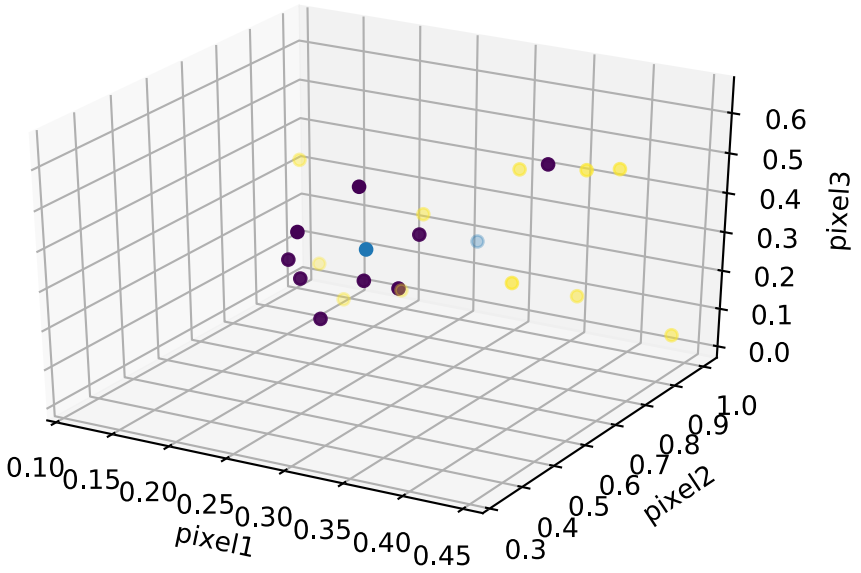
In [17]:
```
fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')
x = np.array(images).T[0]
y = np.array(images).T[1]
z = np.array(images).T[2]

ax.scatter(x, y, z)

ax.set_xlabel('pixel1')
ax.set_ylabel('pixel2')
ax.set_zlabel('pixel3')

plt.show()
```



1b.

There is one obvious cluster in the foreground and then there are two hard-to-assign dots that reside in the middle between that obvious cluster which mostly lines up on the lower values of axis pixel2 (around .3) and the plane that could run through the background pixels in the high values of axis pixel 2 (between .9 and 1). I would group those background pixels together as well as a cluster because they line up pretty perfectly along a plane even though they aren't really clumped. So, I'd say there are two clusters total, though the second one is spread out much more (though in a very uniform way)

In [20]:
```
clusters, centroids = kmeans(images, 2)

fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')
x = np.array(images).T[0]
y = np.array(images).T[1]
z = np.array(images).T[2]

xx = np.array(centroids).T[0]
yy = np.array(centroids).T[1]
zz = np.array(centroids).T[2]

ax.scatter(x, y, z, c = clusters)
ax.scatter(xx, yy, zz)

ax.set_xlabel('pixel1')
ax.set_ylabel('pixel2')
ax.set_zlabel('pixel3')

plt.show()
```

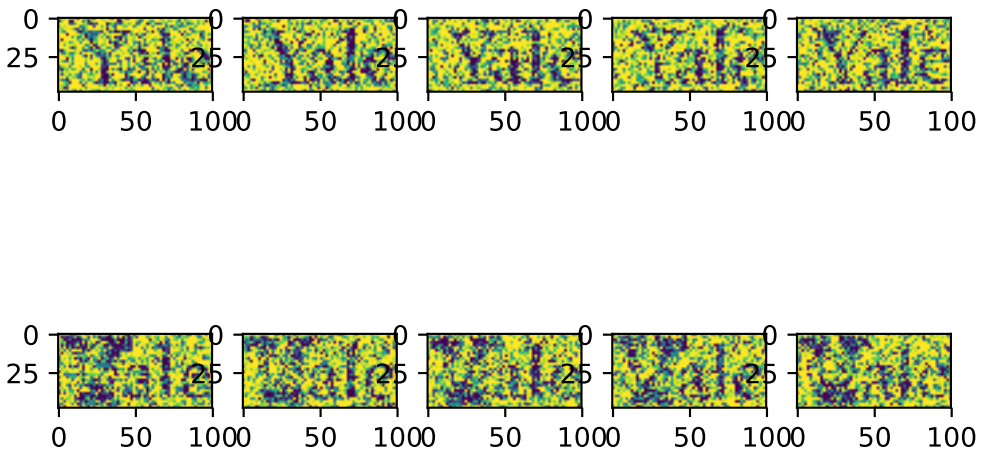


1d.

Yes, these are the clusters that I expected based on visual inspection in part a. In the foreground, there is the purple cluster, and in the background, there is the yellow cluster. The only difference is that it included the two dots (one in each of the clusters) that I said were free floating. Besides the color changing, I cannot see any difference when I rerun k-means. I think this is because the data set is so small and pretty obviously delineated so there is not much room for variability, besides which cluster comes first in the array.

In [23]:
```
data = scipy.io.loadmat('logos.mat')['data']

f, axarr = plt.subplots(2,5)
axarr[0,0].imshow(data[:,:,0])
axarr[0,1].imshow(data[:,:,1])
axarr[0,2].imshow(data[:,:,2])
axarr[0,3].imshow(data[:,:,3])
axarr[0,4].imshow(data[:,:,4])
axarr[1,0].imshow(data[:,:,5])
axarr[1,1].imshow(data[:,:,6])
axarr[1,2].imshow(data[:,:,7])
axarr[1,3].imshow(data[:,:,8])
axarr[1,4].imshow(data[:,:,9])
```

Out[23]:
```
<matplotlib.image.AxesImage at 0xc16ea88>
```



2a.

The two classes are the Yale logo without the bulldog in the Y, and the Yale logo with the bulldog in the Y

In [24]:
```
X = np.empty([10, 4800])
for i in range(10):
    X[i] = data[:,:,i].flatten()
X[0].shape
```

Out[24]:
```
(4800L,)
```

2b.

The dimentionality of each data point is (4800,) when I run .shape. This means it is basically a 1 x 4800 array since it was flattened. I do not believe it is possible to make a scatter plot analogous to the one from 1-b because it has too many dimensions (4800 vs 3)

In [38]:
```
clusterz, centroidz = kmeans(X, 2)
clusterz
```

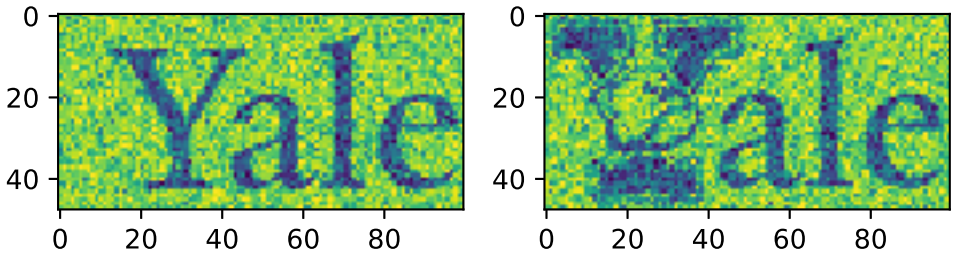
Out[38]:
```
array([0., 0., 0., 0., 0., 1., 1., 1., 1., 1.])
```

2c.

The first 5 elements of clusterz are 0 while the last 5 are 1 which is what I expect based on what the images looked like. When plotting the resulting centroids side by side (see cell below), I see that the first centroid is comprised of the Yale logo and the second centroid is comprised of the Yale logo with the bulldog in the Y. However, running kmeans multiple times only sometimes gets this distinction right. The first time I ran it, both images looked identical with neither Y showing the bulldog. On successive runs, it sometimes the Y in one image would be rather blue while the other was normal looking. However, most of the time, kmeans returned the correct clusters. I think this imperfection is due to the noise in the images which is confusing it.

In [131]:
```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax1.imshow(centroidz[0].reshape((48, 100)))
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(centroidz[1].reshape((48, 100)))
```

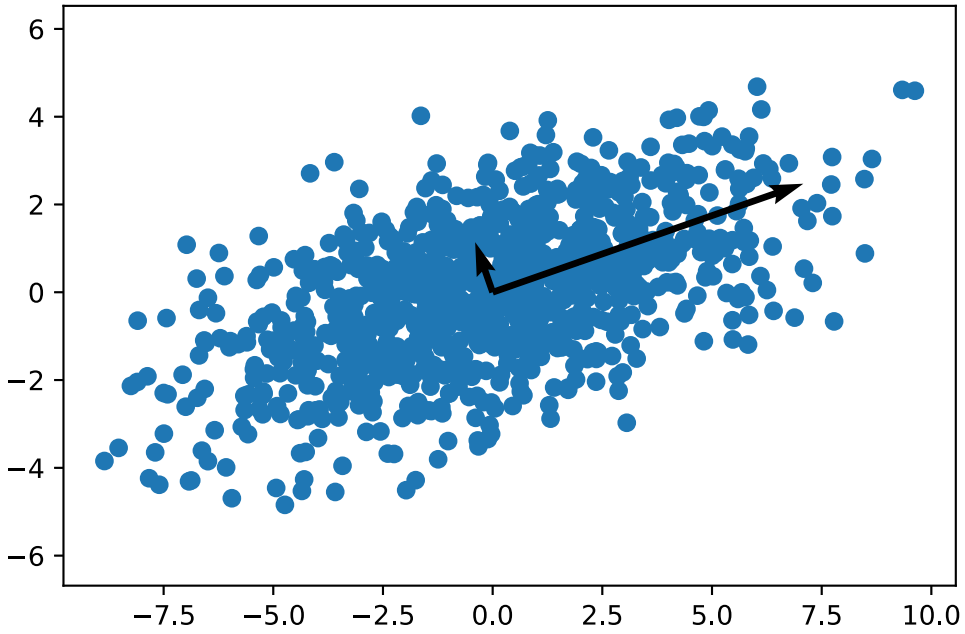
Out[131]:
```
<matplotlib.image.AxesImage at 0x11162c88>
```



In [42]:
```
# 3a.
def pca(X, m):
    rows, columns = X.shape
    X = X - np.mean(X, axis=0)
    C = (1/(rows - 1)) * np.dot(X.T, X)
    evalues, evectors = scipy.sparse.linalg.eigs(C, k=m)
    ind = evalues.argsort()
    variances = evalues[ind[::-1]]
    components = evectors[:,ind[::-1]]
    output = [np.asarray(components), np.asarray(evalues)]
    return output
```

In [44]:
```
# 3b.
gaussian = scipy.io.loadmat('gaussian.mat')['gaussian']
plt.scatter(gaussian[:,0], gaussian[:,1])
result = pca(gaussian, 2)
PC1_x = result[0][0][0]
PC1_y = result[0][1][0]
PC2_x = result[0][0][1]
PC2_y = result[0][1][1]
lambda_1 = result[1][0]
lambda_2 = result[1][1]
plt.quiver([0,0], [0,0], [lambda_1 * PC1_x, lambda_2 * PC2_x], [lambda_1 * PC1
_y, lambda_2 * PC2_y], scale=30)
plt.axis('equal')
```

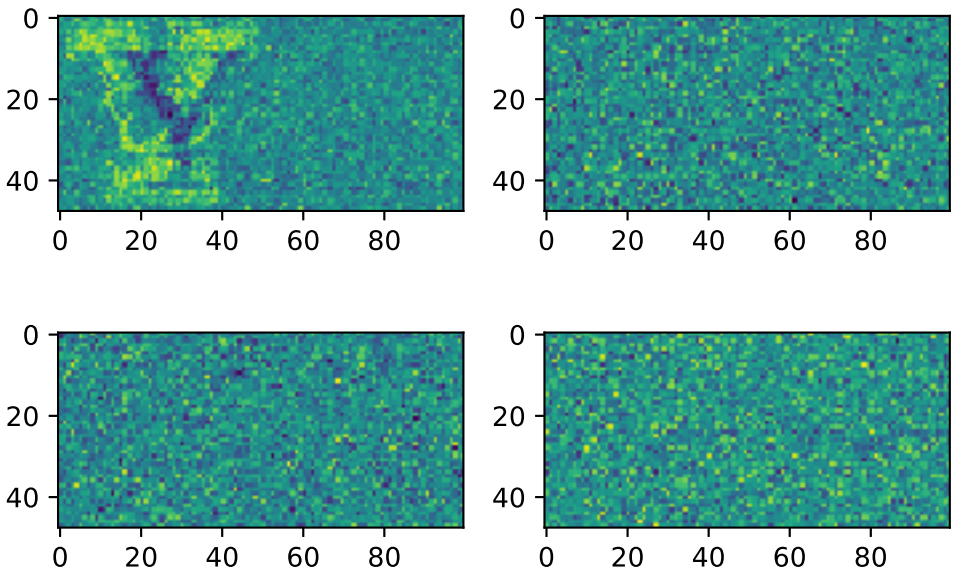
Out[44]:
```
(-9.782499112120043, 10.557136014917834, -5.335042518868506, 5.17565528542925
8)
```



In [45]:
```
images = [data[:,:,i] for i in range(10)]
images_flattened = [im.flatten() for im in images]
x = np.array(images_flattened)
result = pca(x, 4)
```

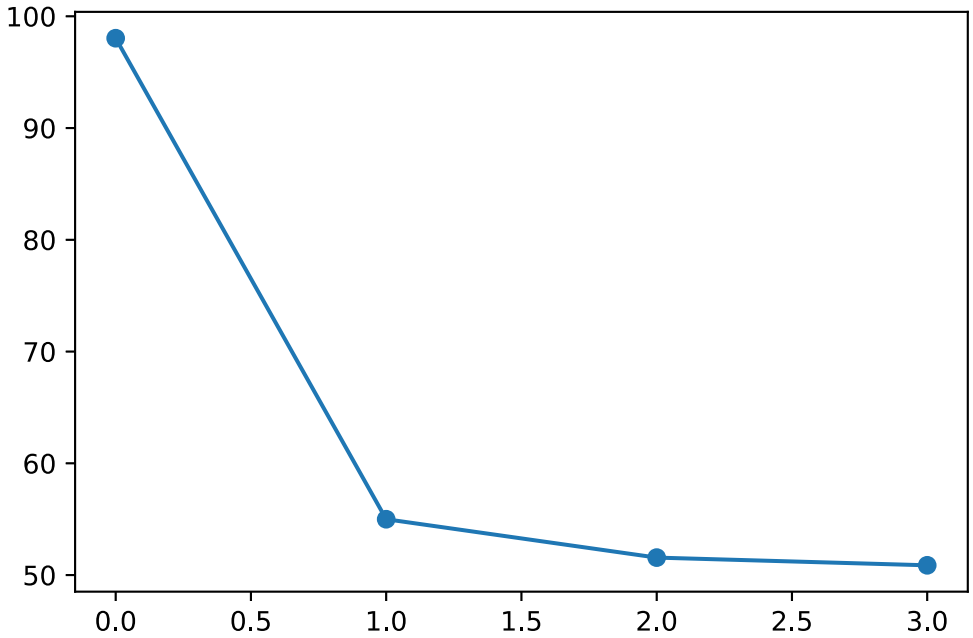
In [46]:
```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax1.imshow(result[0][:,0].real.reshape([48,100]))
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(result[0][:,1].real.reshape([48,100]))
ax3 = fig.add_subplot(2,2,3)
ax3.imshow(result[0][:,2].real.reshape([48,100]))
ax4 = fig.add_subplot(2,2,4)
ax4.imshow(result[0][:,3].real.reshape([48,100]))
```

Out[46]: <matplotlib.image.AxesImage at 0x51998c8>



In [38]:
```
plt.plot(result[1].real, '-o')
```

Out[38]: [<matplotlib.lines.Line2D at 0xf377888>]



4a.

For the principle components, only the first one shows anything of value, which is the contrast between the two different Y's, and the other three just show noise. In the graph of the variances, the first varience is drastically different than the next three, which are of very similar value. Therefore, using just one dimension would suffice to represent the data set. This is the result that I expected since there is only one defining feature between the two sets of images.

In [47]:
```
# result[0] is the components from pca()
# x is a (10, 4800) matrix which is each image flattened
norm = np.sqrt(sum(result[0][:,0].real**2))
projected = [(np.dot(x[i], result[0][:,0].real)/norm**2) for i in range(10)]
projection = np.array(projected)
projection.shape
```

Out[47]:
```
(10L,)
```

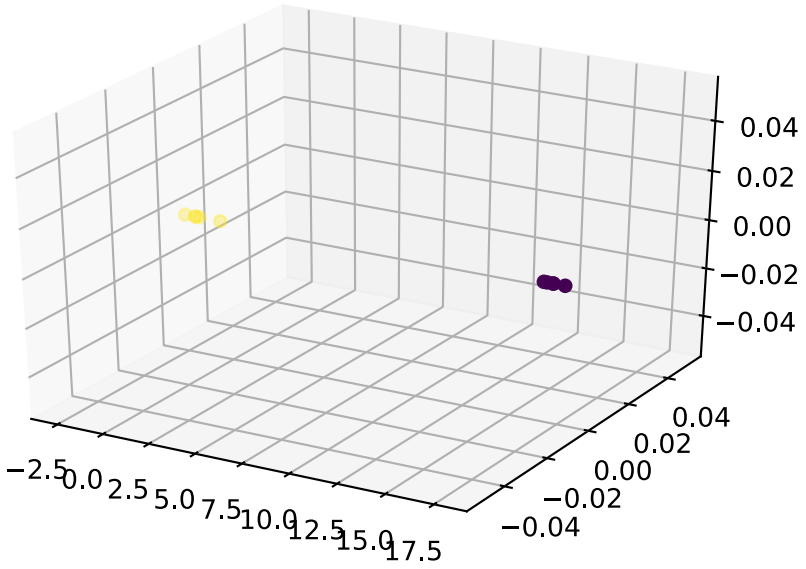
4b.

The data is now 1 dimensional

In [55]:
```
clusters, centroids = kmeans(projection.reshape(10,1), 2)
clusters
```

Out[55]:
```
array([0., 0., 0., 0., 0., 1., 1., 1., 1., 1.])
```

In [117]:
```
fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')
x = projection
y = np.zeros(10)

ax.scatter(x, y, c=clusters)

plt.show()
```

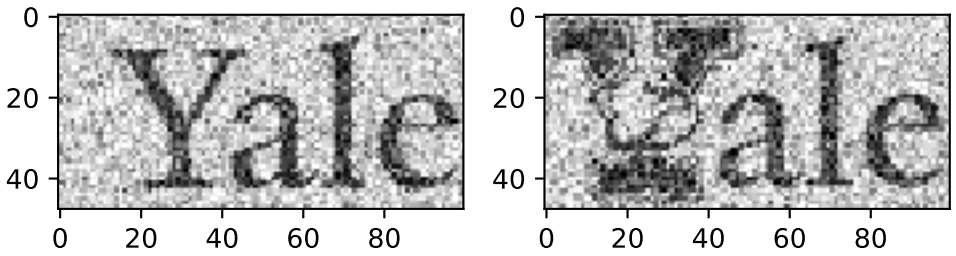


4c.

Yes, the resulting assignment is what I expected because there are five points in one cluster and 5 points in the other cluster which is how it should be since 5 of the images have the bulldog Y and the other five images don't. Running the K-means over and over shows that this version is much more accurately able to compute the right centroids each time. It rarely did it wrong.

In [118]:
```
# displaying centroid images side by side. Printed out the cluster assignments
earlier in order see which ones to average
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax1.imshow(np.mean(data[:,:,0:4], axis=(2)), cmap='gray')
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(np.mean(data[:,:,5:9], axis=(2)), cmap='gray')
```

Out[118]: <matplotlib.image.AxesImage at 0x10e8f188>



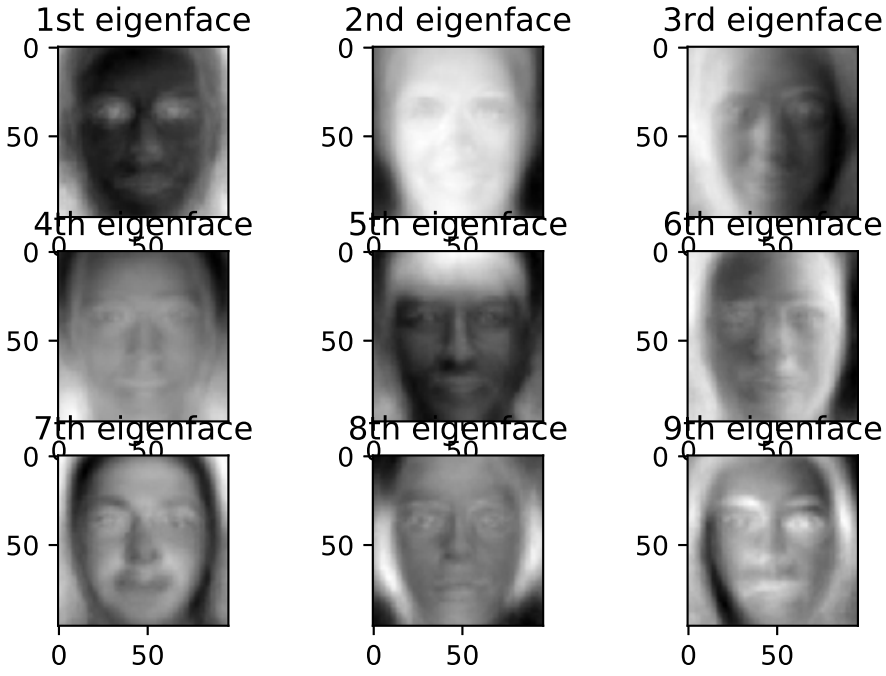
4d.

The same approach worked better in this case because the dimension reduction reduced the impact of the noise in the images. Because the kmeans algorithm is susceptible to noise, in part 2 of this assignment, I had to run the kmeans algorithm a couple times before it spat out the correct cluster assignment whereas in 4c, it got it right almost all of the time.

In [66]:
```
faces = scipy.io.loadmat('faces.mat')['Data']
#faces = np.double(faces) / 255
faces = np.double(faces)
faces /= faces.max(1, keepdims=True)
```

In [71]:
```
r = pca(faces, 9)
```

In [76]:
```
fig = plt.figure()
ax1 = fig.add_subplot(3, 3, 1)
ax1.imshow(r[0][:,0].real.reshape([96,96]), cmap='gray')
ax2 = fig.add_subplot(3,3,2)
ax2.imshow(r[0][:,1].real.reshape([96,96]), cmap='gray')
ax3 = fig.add_subplot(3,3,3)
ax3.imshow(r[0][:,2].real.reshape([96,96]), cmap='gray')
ax4 = fig.add_subplot(3,3,4)
ax4.imshow(r[0][:,3].real.reshape([96,96]), cmap='gray')
ax5 = fig.add_subplot(3,3,5)
ax5.imshow(r[0][:,4].real.reshape([96,96]), cmap='gray')
ax6 = fig.add_subplot(3,3,6)
ax6.imshow(r[0][:,5].real.reshape([96,96]), cmap='gray')
ax7 = fig.add_subplot(3,3,7)
ax7.imshow(r[0][:,6].real.reshape([96,96]), cmap='gray')
ax8 = fig.add_subplot(3,3,8)
ax8.imshow(r[0][:,7].real.reshape([96,96]), cmap='gray')
ax9 = fig.add_subplot(3,3,9)
ax9.imshow(r[0][:,8].real.reshape([96,96]), cmap='gray')
ax1.title.set_text('1st eigenface')
ax2.title.set_text('2nd eigenface')
ax3.title.set_text('3rd eigenface')
ax4.title.set_text('4th eigenface')
ax5.title.set_text('5th eigenface')
ax6.title.set_text('6th eigenface')
ax7.title.set_text('7th eigenface')
ax8.title.set_text('8th eigenface')
ax9.title.set_text('9th eigenface')
```



5a.

The eigenfaces highlight the areas of greatest variablity between the faces. The relevant features when comparing the face images are the eyes, nose and mouth because those are the areas that change the most from person to person.