Lab 5 - Principal Component Analysis Solution

Name: Shamith Achanta (shamith2)

Due September 30, 2019 11:59 PM

Logistics and Lab Submission

See the course website. This is the last lab for this section of the course. Make sure to be up to date for the policies of the second part of the course. **You will have another lab next week and a different TA (who is not familiar with this lab), so it is in your best interests to finish this lab before next week's lab session.**

There will be office hours on Wednesdays, as usual.

What You Will Need To Know For This Lab:

- Eigendecomposition
- Singular Value Decomposition
- Principal Component Analysis

Preamble (Don't change this):

In [1]:
```
%pylab inline
import numpy as np
from sklearn import neighbors
from mpl_toolkits.mplot3d import Axes3D
import random
from sklearn.decomposition import PCA
from PIL import Image
from sklearn.cluster import KMeans
import scipy.spatial.distance as dist
from matplotlib.colors import ListedColormap
```

```
Populating the interactive namespace from numpy and matplotlib
```

Enable Interactive Plots

In [2]:
```
enable_interactive=False # If you want to rotate plots, set this to True.
# When submitting your notebook, enable_interactive=False and run the whole notebook.
# The interactive stuff can be a bit glitchy, so if you're having trouble, turn them off.
if enable_interactive:
    # These packages allow us to rotate plots and what not.
    from IPython.display import display
    from IPython.html.widgets import interact
```

Problem 1: Visualizing Principal Components (50 points)

In this problem, you will be implementing PCA, visualizing the principal components and using it to perform dimensionality reduction.

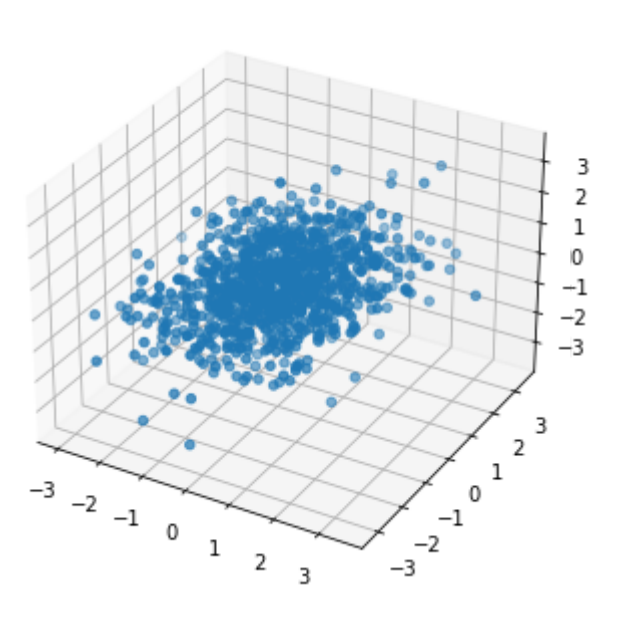
Do not use a pre-written implementation of PCA for this problem (e.g. sklearn.decomposition.PCA). You should assume that the input data has been appropriately pre-processed to have zero-mean features.

In [3]:
```
# We will generate some data.
numpy.random.seed(seed=2232017)
true_cov = np.array([[1,.5,.2],[.5,1,.3],[.2,.3,1]]) #This is the true covariance matrix
                                                     #of the data. Do not use it in your code!
data=(np.random.randn(1000,3)).dot(np.linalg.cholesky(true_cov).T)
```

First, we visualize the data using a 3D scatterplot.

Our data is stored in a variable called `data` where each row is a feature vector (with three features).

In [4]:
```
fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(data[:,0],data[:,1],data[:,2])
if enable_interactive:
    @interact(elev=(-90, 90), azim=(0, 360))
    def view(elev, azim):
        ax.view_init(elev, azim)
        display(ax.figure)
```



Write a function which implements PCA via the eigendecomposition. **(10 points)**

You will be given as input:

- A (N,d) numpy array of data (with each row as a feature vector)

Your function should return a tuple consisting of the PCA transformation matrix (which is (d,d)), and a vector consisting of the amount of variance explained in the data by each PCA feature. **Note that the PCA features are ordered in decreasing amount of variance explained, by convention.**

Hints:

- The function numpy.linalg.eigh will be useful. Note that it returns its eigenvalues in *ascending* order. numpy.fliplr or similar may be useful as well.
- You can calculate the covariance matrix of the data by multiplying the data matrix with its transpose in the appropriate order, and scaling it.
- Do not use numpy.cov -- we are assuming the data has zero mean beforehand, so the number of degrees of freedom is different (since the covariance estimate knows the mean in our case).

In [5]:
```
def pcaeig(data):
    #Put your code here
    cov = (1/data.shape[0])*(data.T).dot(data)
    eigen_val, eigen_vect = np.linalg.eigh(cov)
    eigen_vect = np.fliplr(eigen_vect).T
    eigen_val = eigen_val[::-1]

    return eigen_val, eigen_vect
```

Now, run PCA on your data. Store your PCA transformation in a variable called `W`, and the amount of variance explained by each PCA feature in a variable called `s`. Print out **on each line** the principal components (i.e. the rows of `W`) along with the corresponding amount of variance explained. **(5 points)**

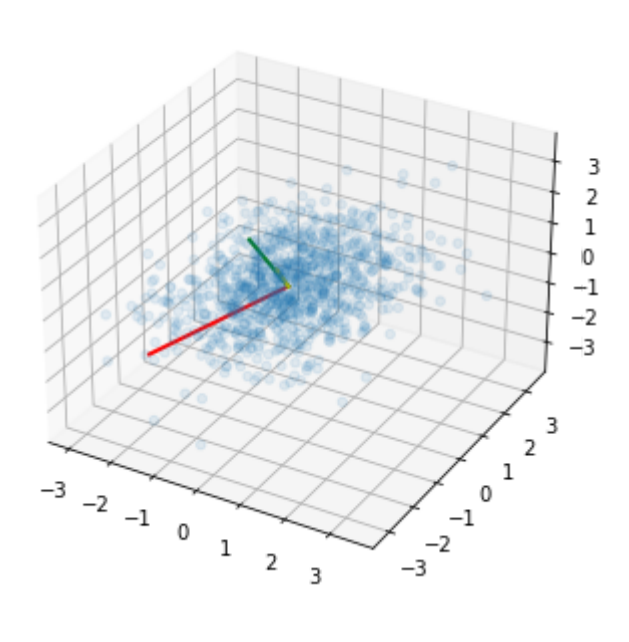
In [6]:
```
# Put your code here
s, W = pcaeig(data)
print("PCA Transformation: \n {} \n".format(W))
print("Variance: \n {}".format(s))
```

```
PCA Transformation:
 [[-0.58522166 -0.68258448 -0.43771457]
 [-0.43983688 -0.18627811  0.87854652]
 [-0.68121886  0.70666746 -0.19121184]]

Variance:
 [1.64258547 0.81658979 0.48669754]
```

We can visualize the principal components on top of our data. The first principal component is in red, and captures the most variance. The second principal component is in green, while the last principal component is in yellow.

We generated our data from am *elliptical distribution*, so it should be easy to visualize these components as the axes of the data (which looks like an ellipsoid).

In [7]:
```
figb = plt.figure()
axb = Axes3D(figb)
axb.scatter(data[:,0],data[:,1],data[:,2],alpha=0.1)
c=['r-','g-','y-']
for var, pc,color in zip(s, W,c):
    axb.plot([0, 2*var*pc[0]], [0, 2*var*pc[1]], [0, 2*var*pc[2]], color, lw=2)
if enable_interactive:
    @interact(elev=(-90, 90), azim=(0, 360))
    def view(elev, azim):
        axb.view_init(elev, azim)
        display(axb.figure)
```



If done correctly, the red line should be longer than the green line which should be longer than the yellow line.

Now, you will implement functions to generate PCA features.

Write a function which implements dimension reduction via PCA. It takes in three inputs:

- A (N,d) numpy array, `data`, with each row as a feature vector
- A (d,d) numpy array, `W`, the PCA transformation matrix (e.g. generated from `pcaeig` or `pcasvd`)
- A number `k`, which is the number of PCA features to retain

It should return a (N,k) numpy array, where the i-th row contains the PCA features corresponding to the i-th input feature vector. **(5 points)**

In [8]:
```
def pcadimreduce(data,W,k):
    # Put your code here
    w = W[0:k,:]
    feat = np.zeros((data.shape[0], k))

    for n in range(data.shape[0]):
        feat[n] = w.dot(data[n])

    return feat
```

Write a function which reconstructs the original features from the PCA features. It takes in three inputs:

- A (N,k) numpy array, `pcadata`, with each row as a PCA feature vector (e.g. generated from `pcadimreduce`)
- A (d,d) numpy array, `W`, the PCA transformation matrix (e.g. generated from `pcaeig` or `pcasvd`)
- A number `k`, which is the number of PCA features

It should return a (N,d) numpy array, where the i-th row contains the reconstruction of the original i-th input feature vector (in `data`) based on the PCA features contained in `pcadata`. **(5 points)**

In [9]:
```
def pcareconstruct(pcadata,W,k):
    # Put your code here
    w  = W[0:k,:].T
    org = np.zeros((pcadata.shape[0],W.shape[0]))

    for i in range(pcadata.shape[0]):
        org[i] = w.dot(pcadata[i])

    return org
```

As a sanity check, if you take $k=3$, perform dimensionality reduction then reconstruction, you should get the original data back:

In [10]:
```
# Reconstructed data using all the principal components
reduced_data=pcadimreduce(data,W,3)
reconstructed_data=pcareconstruct(reduced_data,W,3)

print ("This should be small:",np.max(np.abs(data-reconstructed_data)))
```

```
This should be small: 3.9968028886505635e-15
```

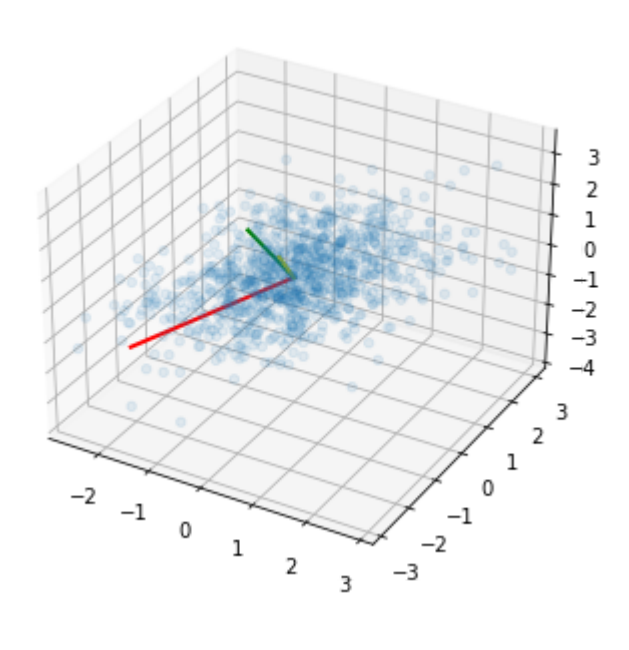
One use of PCA is to help visualize data. The 3-D plots above are a bit hard to read on a 2-D computer screen or when printed out.

Use PCA to to reduce the data to two dimensions. Visualize the first two PCA features with a scatter plot. Also, construct an approximation of the original features using the first two principal components into a (N,d) array called `reconstructed_data`.**(10 points)**

In [11]:
```
#Put your code here
reduced_data = pcadimreduce(data,W,2)
reconstructed_data = pcareconstruct(reduced_data,W,2)
```

We can now visualize the data using two principal components in the original feature space.

In [12]:
```
figc = plt.figure()
axc = Axes3D(figc)
axc.scatter(reconstructed_data[:,0],reconstructed_data[:,1],reconstructed_data[:,2],alpha=0.1)
c=['r-','g-','y-']
for var, pc,color in zip(s, W,c):
    axc.plot([0, 2*var*pc[0]], [0, 2*var*pc[1]], [0, 2*var*pc[2]], color, lw=2)

if enable_interactive:
    @interact(elev=(-90, 90), azim=(0, 360))
    def view(elev, azim):
        axc.view_init(elev, azim)
        display(axc.figure)
```



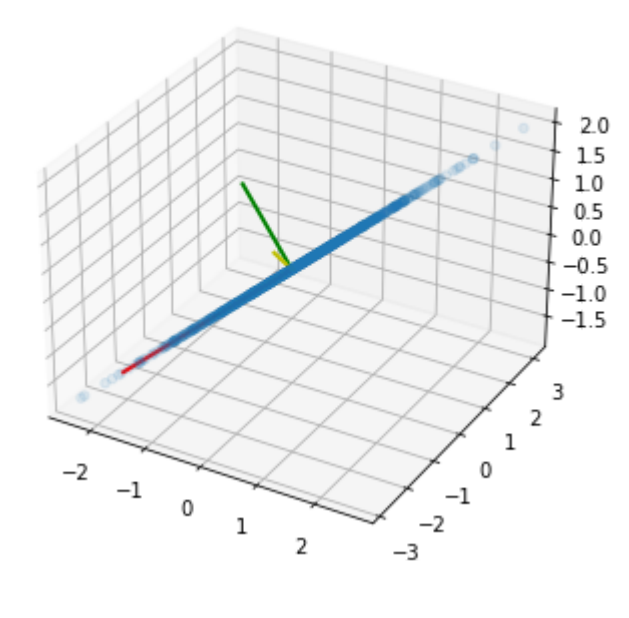
If done correctly, you should see no component of the data along the third principal direction, and the data should lie in a plane. This may be easier to see with the Interactive Mode on.

Use PCA to reduce the data to one dimension and store the one dimensional PCA feature in `reduced_data_1`. Construct an approximation of the original features using the first principal component into a (N,d) array called `reconstructed_data_1`. **(5 points)**

In [13]:
```
#Put your code here
reduced_data_1 = pcadimreduce(data, W, 1)
reconstructed_data_1 = pcareconstruct(reduced_data_1, W, 1)
```

We can now visualize this in the original feature space.

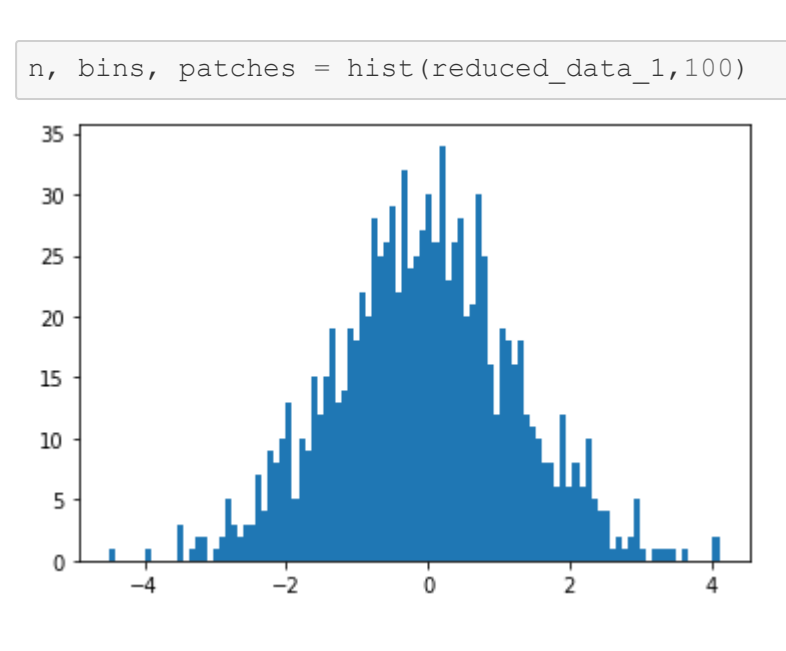
In [14]:
```
figd = plt.figure()
axd = Axes3D(figd)
axd.scatter(reconstructed_data_1[:,0],reconstructed_data_1[:,1],reconstructed_data_1[:,2],alpha=0.1)
c=['r-','g-','y-']
for var, pc,color in zip(s, W,c):
    axd.plot([0, 2*var*pc[0]], [0, 2*var*pc[1]], [0, 2*var*pc[2]], color, lw=2)

if enable_interactive:
    @interact(elev=(-90, 90), azim=(0, 360))
    def view(elev, azim):
        axd.view_init(elev, azim)
        display(axd.figure)
```



If done correctly, you should see no component of the data along the second and third principal direction, and the data should lie along a line. This may be easier with the Interactive Mode on.

We can also visualize the PCA feature as a histogram:

In [15]:
```
n, bins, patches = hist(reduced_data_1,100)
```



Finally, you will implement PCA via the SVD. **(5 points)**

You will be given as input:

- A (N,d) numpy array of data (with each row as a feature vector)

Your function should return a tuple consisting of the PCA transformation matrix, and a vector consisting of the amount of variance explained in the data by each PCA feature. **Note that the PCA features are ordered in decreasing amount of variance explained.**

Hints:

- The function numpy.linalg.svd will be useful. Use the full SVD (default).
- Be careful with how the SVD is returned in `numpy.linalg.svd` (`V` in numpy is the transpose of what is in the notes).

In [16]:
```
def pcasvd(data):
    #Put your code here
    U, S, W = linalg.svd(data)
    D = (1.0 / data.shape[0]) * (np.square(np.diag(S)))

    return D, W
```

If your PCA implementation via the SVD is correct (and your Eigendecomposition implementation is correct), principal components should match between the SVD and PCA implementations (up to sign, i.e. the i-th principal component may be the negative of the i-th principal component from the eigendecomposition approach).

Verify this by printing out the principal components and the corresponding amount of variance explained. **You will not get any credit if the principal components (up to sign) and variances do not match the eigendecomposition. (5 points)**

In [17]:
```
# Put your code here
D, W = pcasvd(data)

print("D:\n {}".format(D))
print("W:\n {}".format(W))
```

```
D:
 [[1.64258547 0.         0.        ]
 [0.         0.81658979 0.        ]
 [0.         0.         0.48669754]]
W:
 [[ 0.58522166  0.68258448  0.43771457]
 [-0.43983688 -0.18627811  0.87854652]
 [ 0.68121886 -0.70666746  0.19121184]]
```

Problem 2: PCA for Data Compression (30 points)

In class, you saw an example application of PCA to create eigenfaces. In this part of the lab, we will look at eigenfaces for compression using the Olivetti faces dataset.

In [18]:
```
# First, we load the Olivetti dataset
from sklearn.datasets import fetch_olivetti_faces


oli = fetch_olivetti_faces()
# Height and Width of Images are in h,w. You will need to reshape them to this size to display them.
h=64
w=64
X = oli.data

X_t=X[20]
X=X[:-1]

#This centering is unnecessary. it just makes the pictures a bit more readable.

X_m=np.mean(X,axis=0)
X=X-X_m # center them
X_t=X_t-X_m

# In the following, X represents the entire dataset. X_t represents some specific image to compress.
```

We can visualize the Olivetti Faces:

We will be making use of Scikit-Learn's PCA functionality.

Three functions will be useful for this problem :

- PCA.fit : Finds the requested number of principal components.
- PCA.transform : Apply dimensionality reduction (returns the PCA features)
- PCA.inverse_transform : Go from PCA features to the original features (Useful for visualizing)

You will also find the following useful:

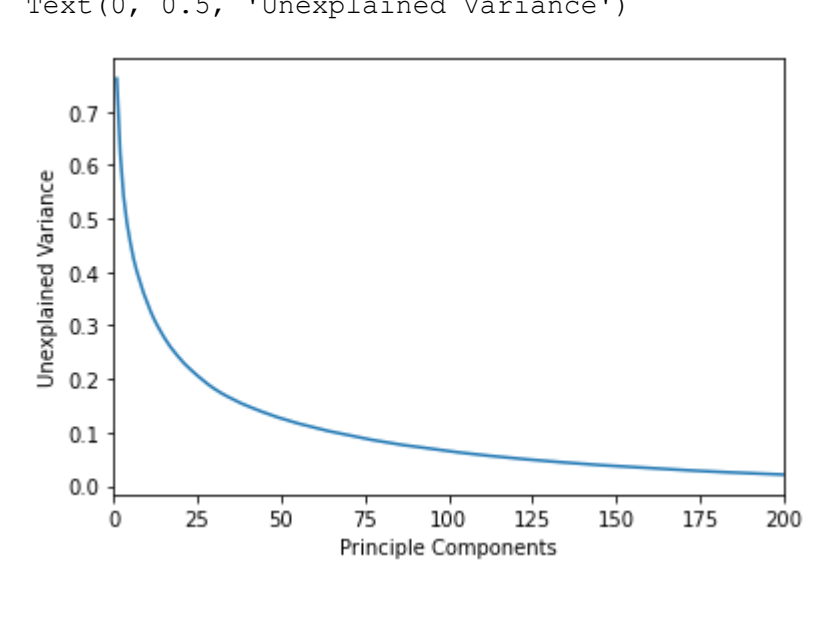
- PCA.explained_variance_ratio_ : Percentage of variance explained by each of the principal components

Plot the fraction of **unexplained** variance on `X` by PCA retaining the first k principal components, where $k=1,\ldots,200$. Note that this is a *scree plot* (normalized by the total variance).

`numpy.cumsum` may be useful for this. **(10 points)**

In [19]:
```
# Put your code here
unexpVar = zeros(200)
val = np.arange(1,201)

for k in val:
    model = PCA(n_components = k)
    model.fit(X)

    reconstructed_X = model.inverse_transform(model.transform(X))

    unexpVar[k-1] = 1 - np.sum(model.explained_variance_ratio_)

pylab.plot(val, unexpVar)
pylab.xlim(0,200)

pylab.xlabel('Principle Components')
pylab.ylabel('Unexplained Variance')
```

Out[19]:
```
Text(0, 0.5, 'Unexplained Variance')
```



Based on the Scree plot, propose a reasonable number of principal components to keep, in order to perform dimensionality reduction. Justify your choice. There is a range of correct answers (but you need to justify yours!). (5 points)

[I would choose the value to be somewhere between the first 50 and 75 Principle Components because these Principle Components have an unexplained variance between 10% and 20%, which implies they have an explained variance between 80% to 90%]

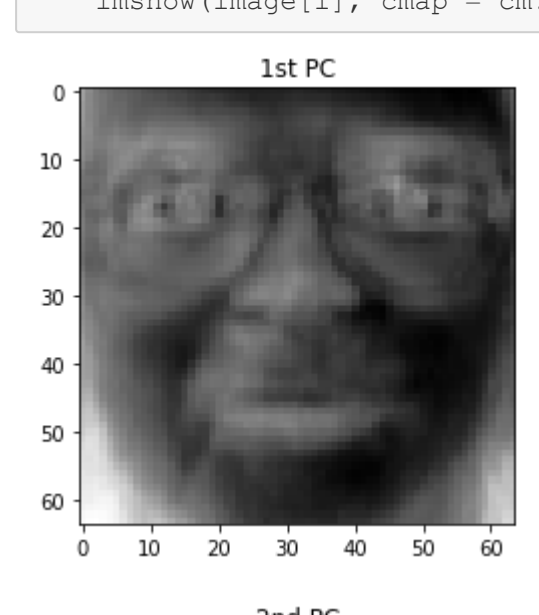
Visualize the first 5 principal components as well as the 30th, 50th and 100th principal components, which are called *eigenfaces* in this context. Assuming your PCA object is called `pca`, the eigenfaces are contained in `pca.components_`, where each row is a principal component.

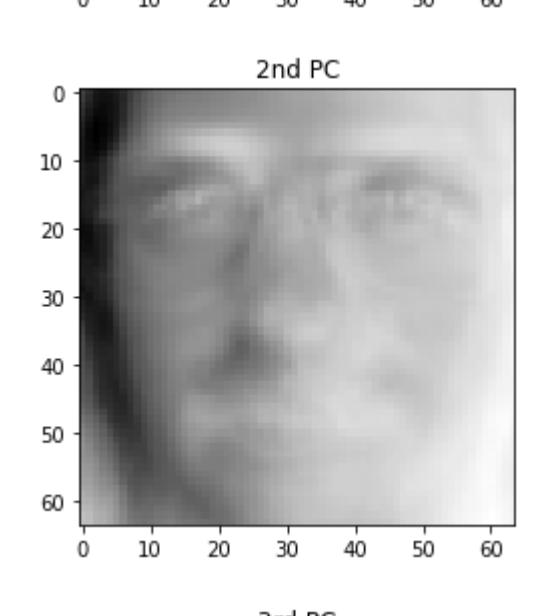
The following code from Lab 4 may be useful:

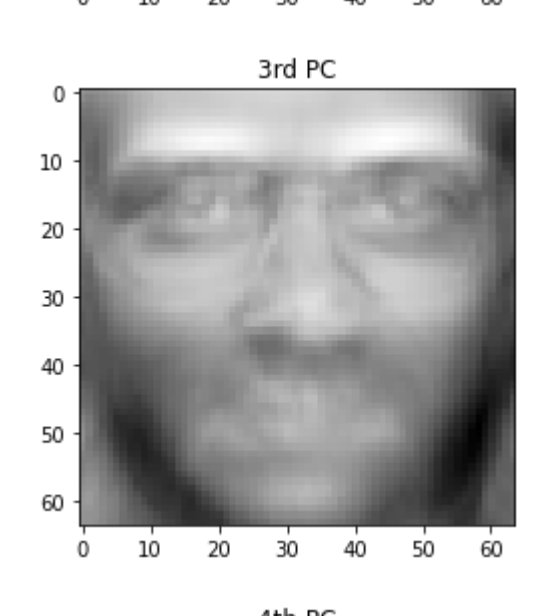
```
figure()
imshow( image , cmap = cm.Greys_r)
```

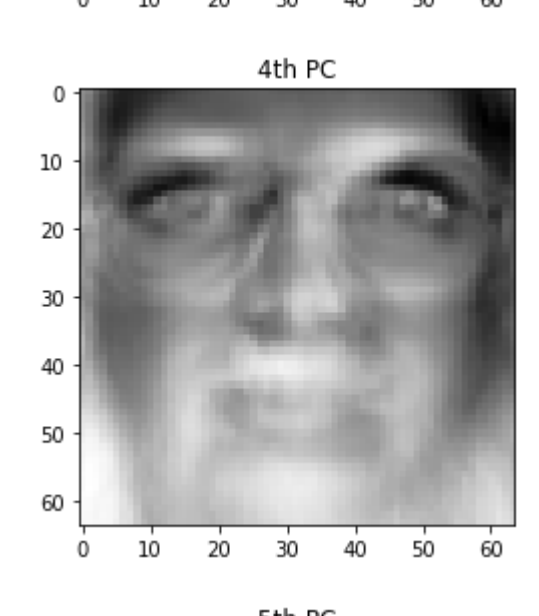
where image is the appropriately reshaped principal component (to `h` rows and `w` columns). **(5 points)**

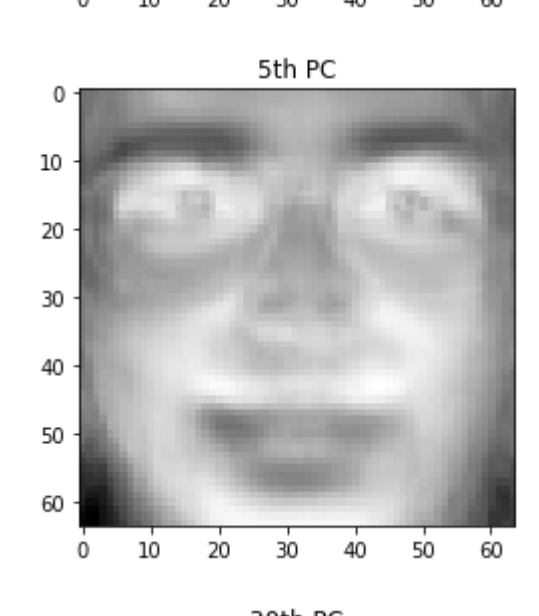
In [20]:
```
# Put your code here
comp = list(range(0, 5))

for i in [29, 49, 99]:
    comp.append(i)

image = np.zeros((8,64,64))

for i in range(64):
    for j, cmp in enumerate(comp):
        image[j][i] = model.components_[cmp][i*64:(i+1)*64]

titles = ['1st PC', '2nd PC', '3rd PC', '4th PC', '5th PC', '30th PC', '50th PC', '100th PC']
for i in range(len(titles)):
    figure()
    plt.title(titles[i])
    imshow(image[i], cmap = cm.Greys_r)
```

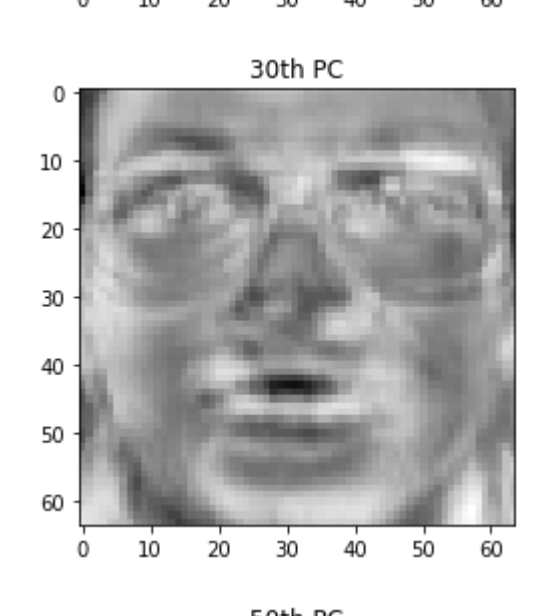


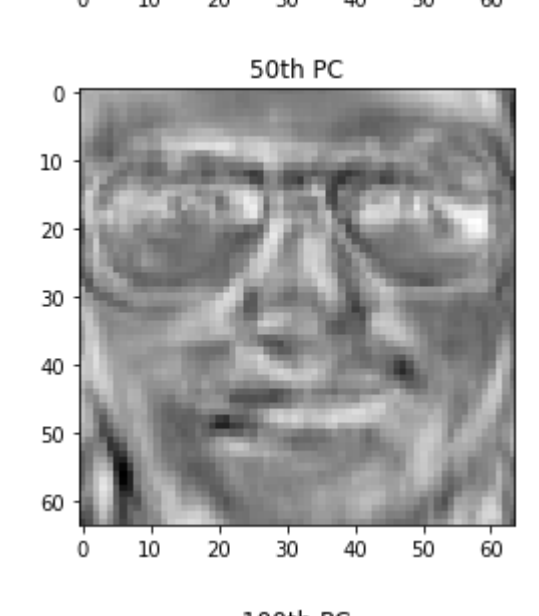


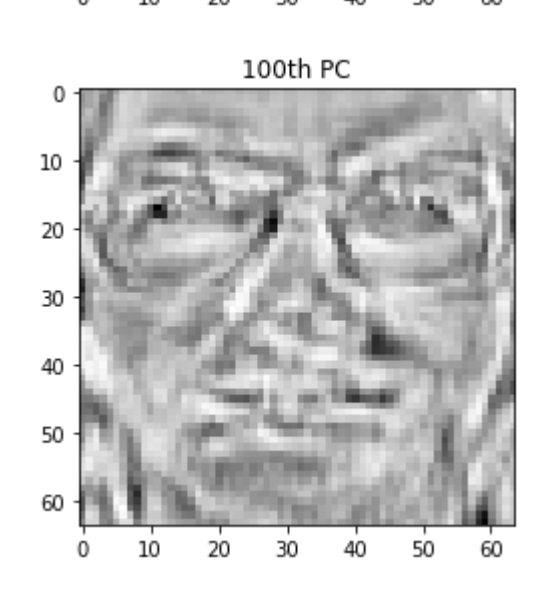










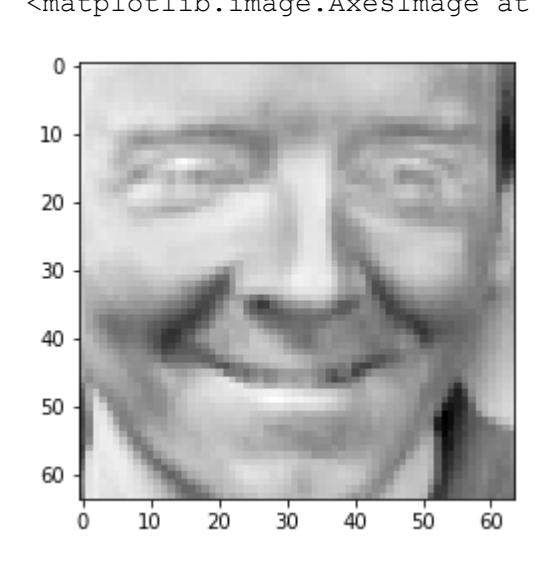


As you can see, later eigenfaces capture more detail as compared to earlier ones (e.g. they're specific to some guy).

Now, you will compress an image, `X_t`, using PCA.

In [21]:
```
# This is what X_t looks like:
imshow((X_t).reshape((h,w)),cmap=cm.Greys_r)
```

Out[21]:
```
<matplotlib.image.AxesImage at 0x29d2d462508>
```



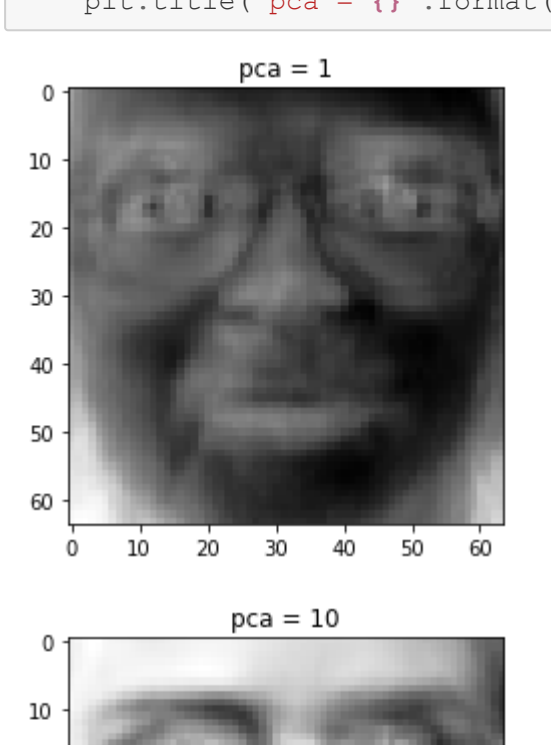
Display the image in `X_t`'s approximation using the first `i` principal components (learned from `X`) where i=1,10,20,...,100 (i.e. in increments of 10), then 120,140,160,180,200 (i.e. in increments of 20).

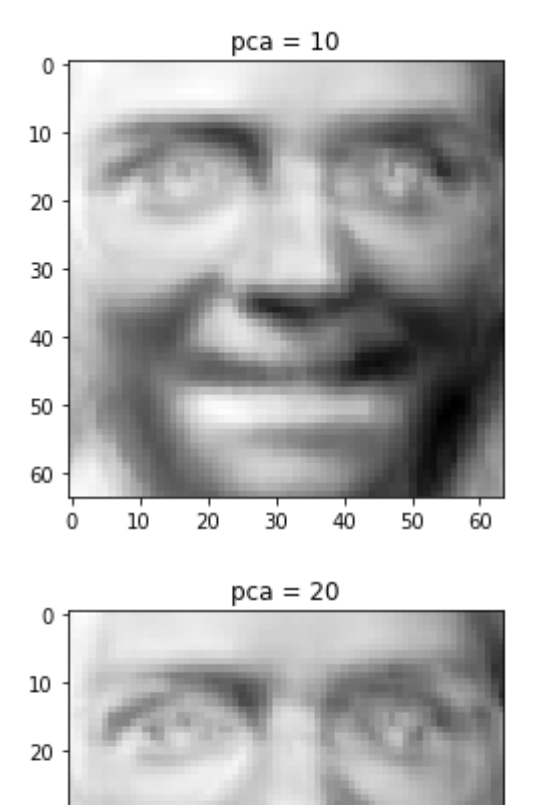
Do this by the following procedure:

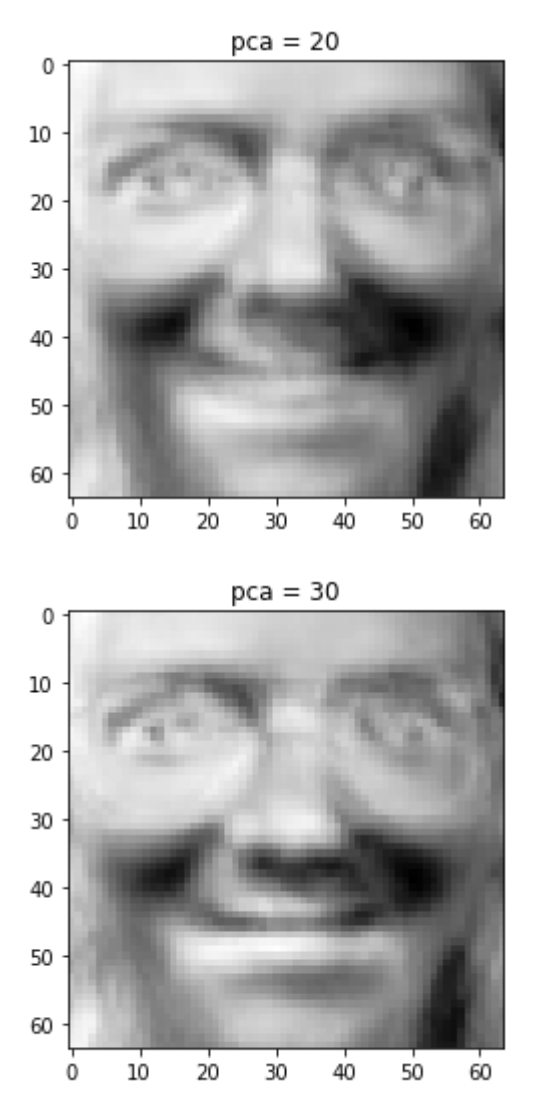
1. Determine the PCA transformation (i.e. fit) on `X`.
2. Transform `X_t` to the PCA features determined by `X`.
3. Retain the first `i` PCA features of the transformed `X_t` (set the others to zero).
4. Transform the result of step 3 back to the original feature space.

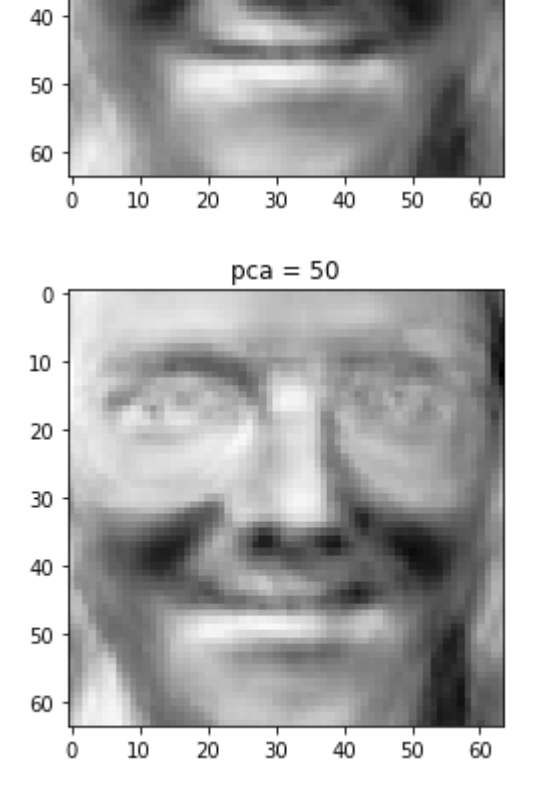
(5 points)

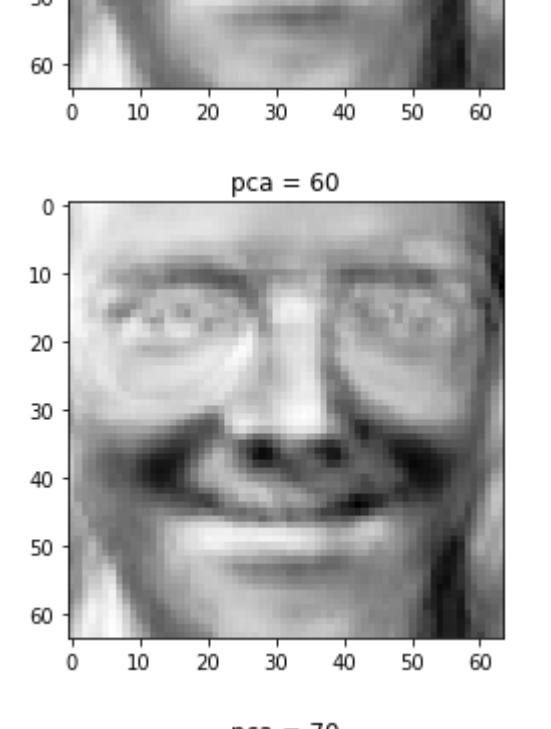
In [22]:
```python
# Put your code here
# constants
begin_1 = 10
begin_2 = 120

end_1 = 110
end_2 = 220

step_1 = 10
step_2 = 20

val = np.concatenate((np.array([1]), np.arange(begin_1, end_1, step_1), np.arange(begin_2, end_2, step_
2)))

for k, i in enumerate(val):
    model = PCA(n_components = i)
    model.fit(X)

    restored = model.inverse_transform(model.transform(X_t.reshape(1, -1)))

    figure()
    imshow(restored.reshape((h, w)), cmap=cm.Greys_r)
    plt.title("pca = {}".format(i))
```

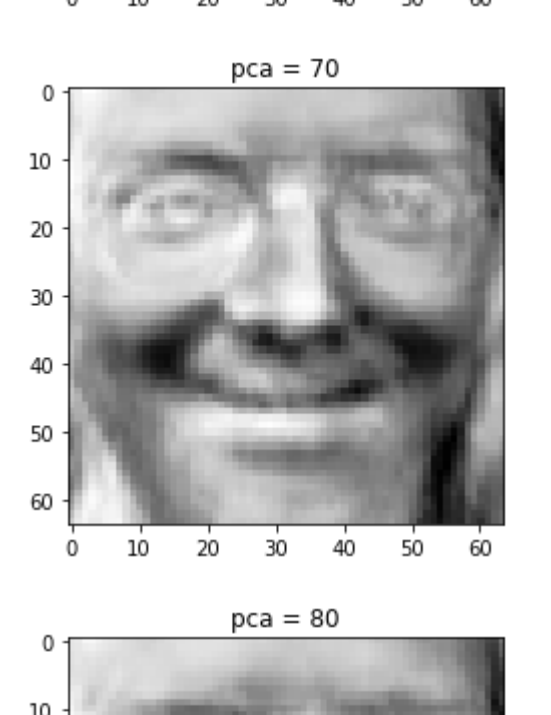


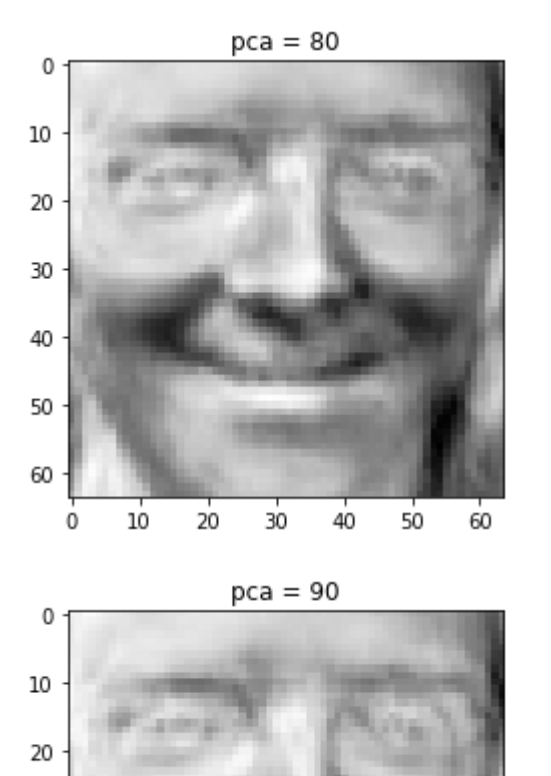


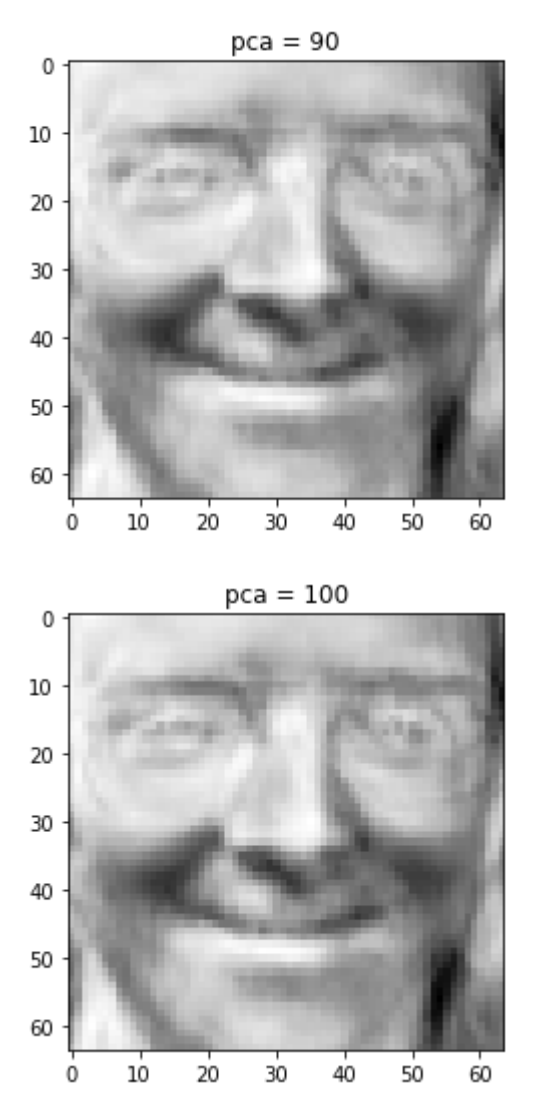


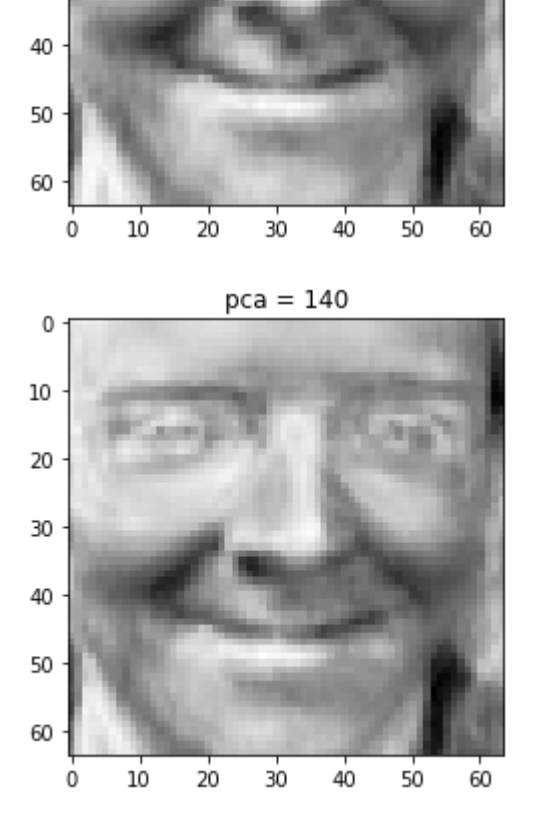


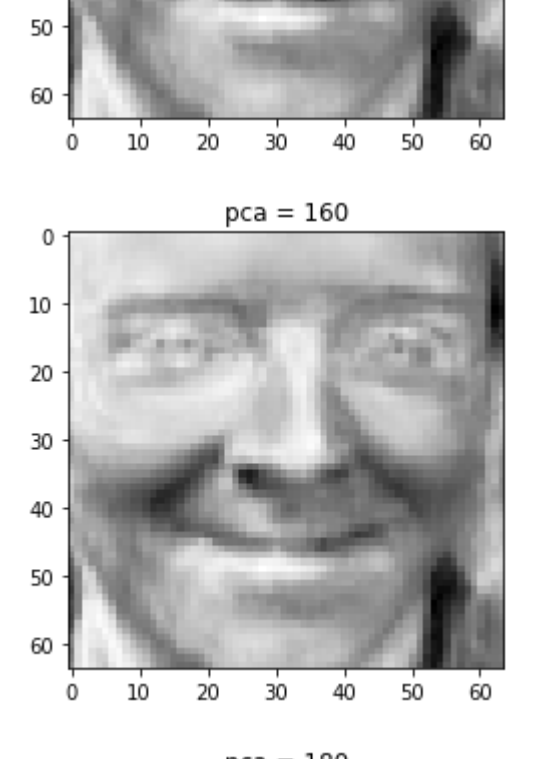


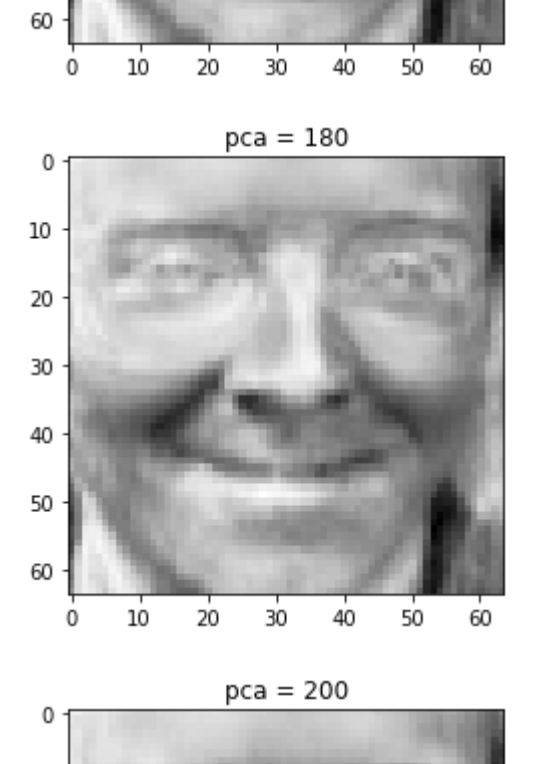


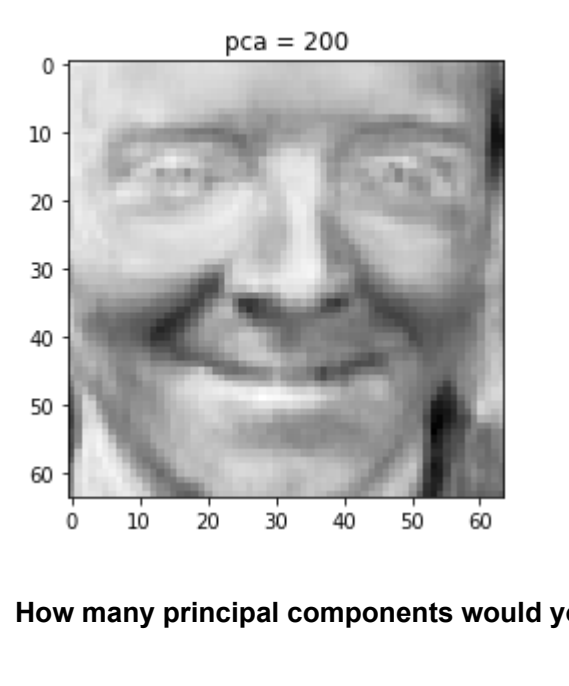


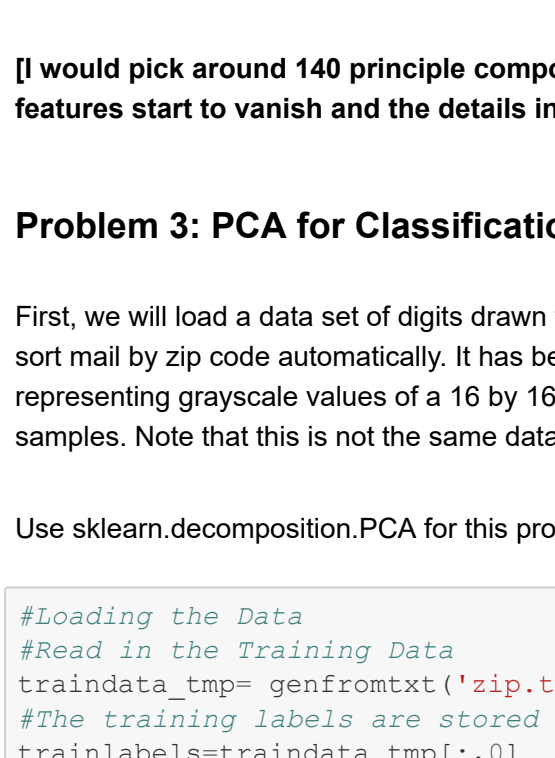


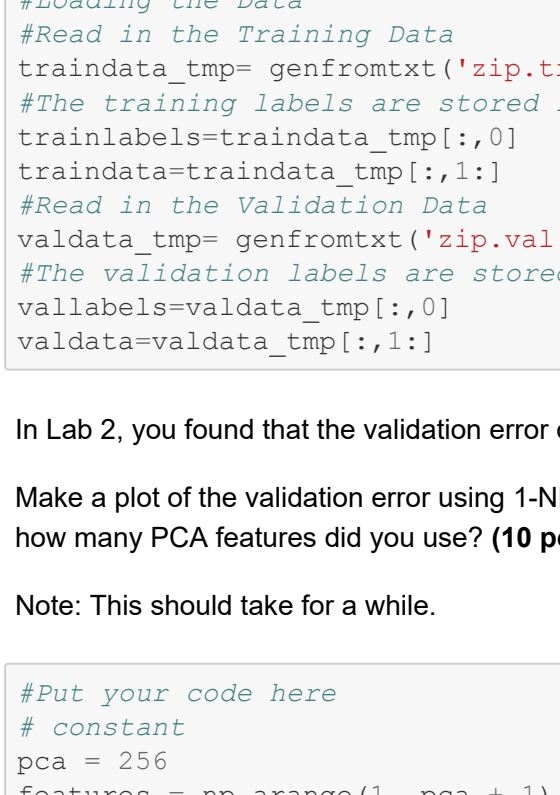












How many principal components would you keep to compress the image? Why? (You may be qualitative or quantitative) (5 points)

[I would pick around 140 principle components to compress the image because, around 140 priciple components, the noisier features start to vanish and the details in the image start to emerge and image resembles the original image]

Problem 3: PCA for Classification (20 points)

First, we will load a data set of digits drawn from zip codes written on US mail. This data set was designed to help get good algorithms to sort mail by zip code automatically. It has been preprocessed a bit, with details given here. Each feature vector consists of real values representing grayscale values of a 16 by 16 image of a digit. The training data has 7291 samples, while the validation data has 2007 samples. Note that this is not the same dataset built into scikit- learn -- it is much larger.

Use sklearn.decomposition.PCA for this problem.

In [23]:
```python
#Loading the Data
#Read in the Training Data
traindata_tmp= genfromtxt('zip.train', delimiter=' ')
#The training labels are stored in "trainlabels", training features in "traindata"
trainlabels=traindata_tmp[:,0]
traindata=traindata_tmp[:,1:]
#Read in the Validation Data
valdata_tmp= genfromtxt('zip.val', delimiter=' ')
#The validation labels are stored in "vallabels", validation features in "valdata"
vallabels=valdata_tmp[:,0]
valdata=valdata_tmp[:,1:]
```

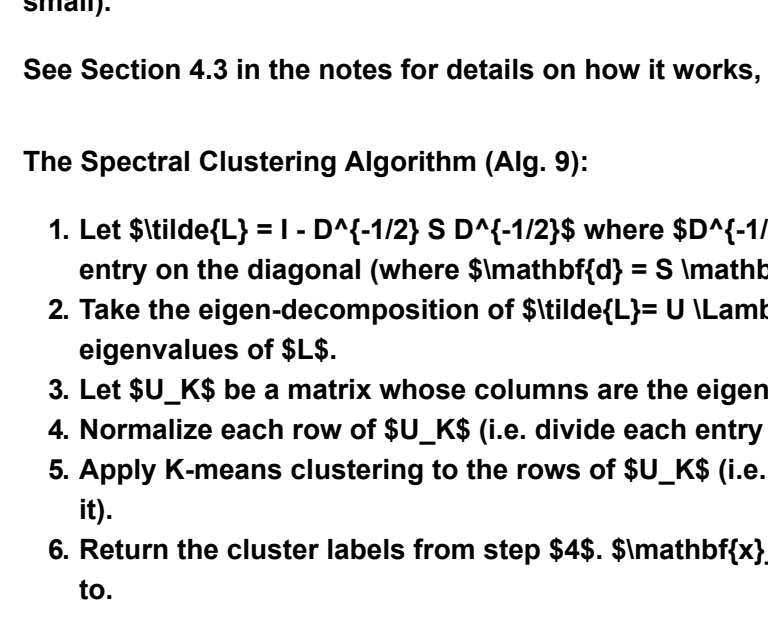
In Lab 2, you found that the validation error on this data set was 0.056 for 1-NN.

Make a plot of the validation error using 1-NN on the PCA features using 1,2,...,256 PCA features. What is the minimum validation error, and how many PCA features did you use? **(10 points)**

Note: This should take for a while.

In [24]:
```python
#Put your code here
# constant
pca = 256
features = np.arange(1, pca + 1)
err = np.zeros(pca)

for feature in features:
    modelPCA = PCA(n_components = feature)
    modelPCA.fit(traindata)

    reduced_train = modelPCA.transform(traindata)
    reduced_val = modelPCA.transform(valdata)

    modelNN = neighbors.KNeighborsClassifier(n_neighbors = 1)
    modelNN.fit(reduced_train, trainlabels)

    err[feature - 1] = 1 - modelNN.score(reduced_val, vallabels)

pylab.plot(features, err)
pylab.xlim(1, pca)

min_err =  (np.argmin(err) + 1)
modelPCA = PCA(n_components = min_err)

modelPCA.fit(traindata)

reduced_train = modelPCA.transform(traindata)
reduced_val = modelPCA.transform(valdata)

modelNN = neighbors.KNeighborsClassifier(n_neighbors = 1)
modelNN.fit(reduced_train, trainlabels)

# print results
print("Minimum Validation Error: {}".format(np.min(err)))
print("Used PCA features: {}".format(min_err))
```

```
Minimum Validation Error: 0.04882909815645242
Used PCA features: 39
```



[Minimum Validation Error: 4.933% Used PCA features: 38]

Does PCA+NN give better or worse performance than just NN? Why do you think this is? Which would you choose (and if using PCA, with how many features)? Take into account both computational costs and error in your answer. (10 points)

[PCA + NN gave a better performance than NN. This is because to PCA is robust to data loss when performing large reductions in dimensionality. One would choose PCA + NN with around 39 components because the algorithm preserves the data while compressing. However, one drawback is that optimizing the number of components seems to be extremely expensive in terms of time and space, so optimizing the algorithm (by compliling with CPython or Numba, or taking a multithreaded approach) to speed up computation is required]

Problem 4 (Optional): Spectral Clustering (20 bonus points)

In this problem, you will implement a powerful clustering algorithm known as spectral clustering. It can separate data that in some cases, K-means cannot (as you will see in this problem).

Spectral clustering works by forming a graph based on similarities between data vectors, and looking for cluster of data vectors such that the similarity between them is high, but the similarity to vectors outisde the clusters is low (and the clusters aren't too small).

See Section 4.3 in the notes for details on how it works, or this tutorial.

The Spectral Clustering Algorithm (Alg. 9):

1. **Let $\tilde{L} = I - D^{-1/2} S D^{-1/2}$ where $D^{-1/2}$ is a square diagonal matrix with $\frac{1}{\sqrt{d_i}}$ as the i-th entry on the diagonal (where $\mathbf{d} = S \mathbf{1}$).**
2. **Take the eigen-decomposition of $\tilde{L}= U \Lambda U^\top$ where Λ is a diagonal matrix containing the eigenvalues of L.**
3. **Let U_K be a matrix whose columns are the eigenvectors corresponding to the K-smallest eigenvalues of L.**
4. **Normalize each row of U_K (i.e. divide each entry on the i-th row by the norm of the i-th row)**
5. **Apply K-means clustering to the rows of U_K (i.e. treat each row of U_K as a K-dimensional feature vector and cluster it).**
6. **Return the cluster labels from step 4. $\mathbf{x}_i$ is assigned to the cluster which the i-th row of U_K was assigned to.**

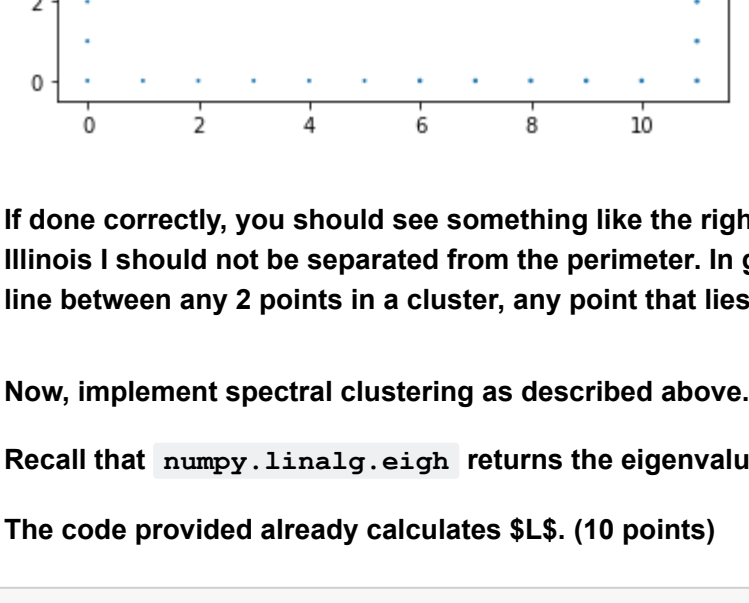
L is known as the normalized Laplacian of the similarity graph, and has many nice properties for analyzing the similarity graph, most of which are beyond the scope of the course.

Note: You don't really need to know why spectral clustering works to do this problem (though it would be nice) -- you just need to be able to implement the algorithm.

First, I'll make a data set based on the Illinois logo.

In [25]:
```python
tmp=np.nonzero(np.asarray(
[[1,1,1,1,1,1,1,1,1,1,1,1],
[1,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,1,1,1,1,1,1,0,0,1],
[1,0,0,0,0,1,1,0,0,0,0,1],
[1,0,0,0,0,1,1,0,0,0,0,1],
[1,0,0,0,0,1,1,0,0,0,0,1],
[1,0,0,1,1,1,1,1,1,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,1],
[1,1,1,1,1,1,1,1,1,1,1,1]]))

illcmap=ListedColormap(['#131F33','#FA6300'])

data=np.c_[tmp[1],tmp[0]]
figure()
scatter(data[:,0],data[:,1])
```

Out[25]: <matplotlib.collections.PathCollection at 0x29d2c6bce88>



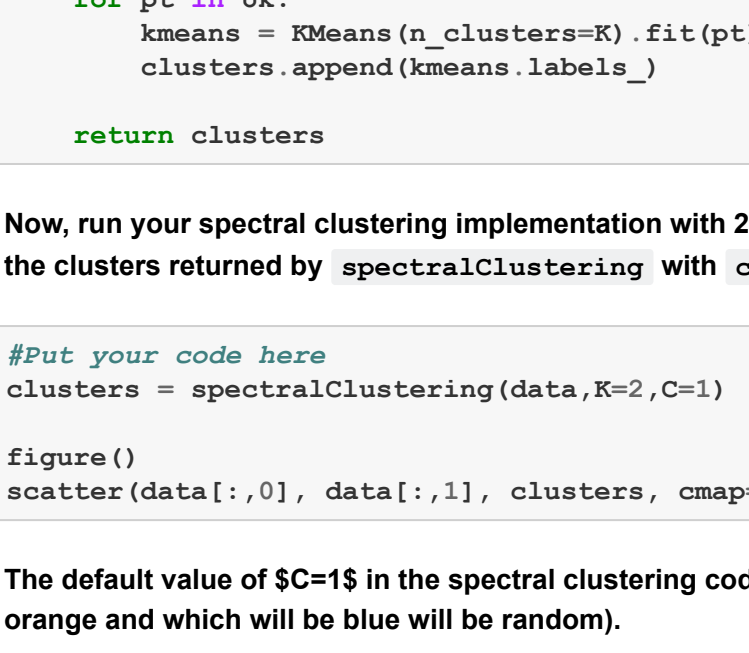
Let us first see what happens if we try to cluster these points using K-means to get 2 clusters.

Use `sklearn.cluster.KMeans` to cluster these points into two clusters.

Plot the clusters using the colors as the labels you get from K-means clustering as a scatter plot, with `cmap=illcmap`. (5 points)

In [26]:
```python
#Put your code here
kmeans = KMeans(n_clusters=2).fit(data)

figure()
scatter(data[:,0], data[:,1], kmeans.labels_ + 1, cmap=illcmap)
```

Out[26]: <matplotlib.collections.PathCollection at 0x29d2c63a7c8>



If done correctly, you should see something like the right half of the points are in one cluster, and the left half are in the other. The Illinois I should not be separated from the perimeter. In general, K-means cannot produce non-convex clusters (i.e. if you draw a line between any 2 points in a cluster, any point that lies on that line is in that cluster), so it cannot separate the I from the border.

Now, implement spectral clustering as described above.

Recall that `numpy.linalg.eigh` returns the eigenvalues of a matrix in *ascending* order.

The code provided already calculates L. (10 points)

In [27]:
```python
def spectralClustering(data,K,C=1):
    W=np.exp(-dist.cdist(data,data,'sqeuclidean')/C)
    W=W-np.diag(np.diag(W))
    Dinv5=np.diag( (W.dot(np.ones(W.shape[0])))**(-0.5) )
    L=np.eye(W.shape[0])-Dinv5.dot(W).dot(Dinv5)
    # Put your code here
    clusters = []
    eigen_vals, eigen_vect = numpy.linalg.eigh(L)
    Uk = eigen_vect.T[:, : K]
    Uk = Uk / Uk.sum(axis=1).reshape((Uk.shape[0], 1))[:, np.newaxis]

    for pt in Uk:
        kmeans = KMeans(n_clusters=K).fit(pt)
        clusters.append(kmeans.labels_)

    return clusters
```

Now, run your spectral clustering implementation with 2 clusters on the data in `data`, and plot the data with the colors given by the clusters returned by `spectralClustering` with `cmap=illcmap`. (5 points)

In [ ]:
```python
#Put your code here
clusters = spectralClustering(data,K=2,C=1)

figure()
scatter(data[:,0], data[:,1], clusters, cmap=illcmap)
```

The default value of $C=1$ in the spectral clustering code should separate the I from the border (though which will be colored orange and which will be blue will be random).

You can find a demo of Spectral Clustering applied to image segmentation based on

Shi, Jianbo, and Jitendra Malik. "Normalized cuts and image segmentation." IEEE Transactions on pattern analysis and machine intelligence 22.8 (2000): 888-905.

at https://scikit-learn.org/0.18/auto_examples/cluster/plot_face_segmentation.html.

And this concludes the Machine Learning section of the course! Good luck with your future endeavors!