Amirali Mohseni

Clustering

1. Load the standard Iris dataset, select the first two features, and ignore the class (or target) variables. Plot the data and see for yourself how “clustered” you think it looks. Include the plot, say how many clusters you think exist, and briefly explain why. (There are multiple reasonable answers to this question.) (5 points)

In [1]:

```
# First we need to inlcude all the libraries required for this a
ssignment!

from __future__ import division
import scipy.linalg
import numpy as np
np.random.seed(0)
import mltools as ml
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:

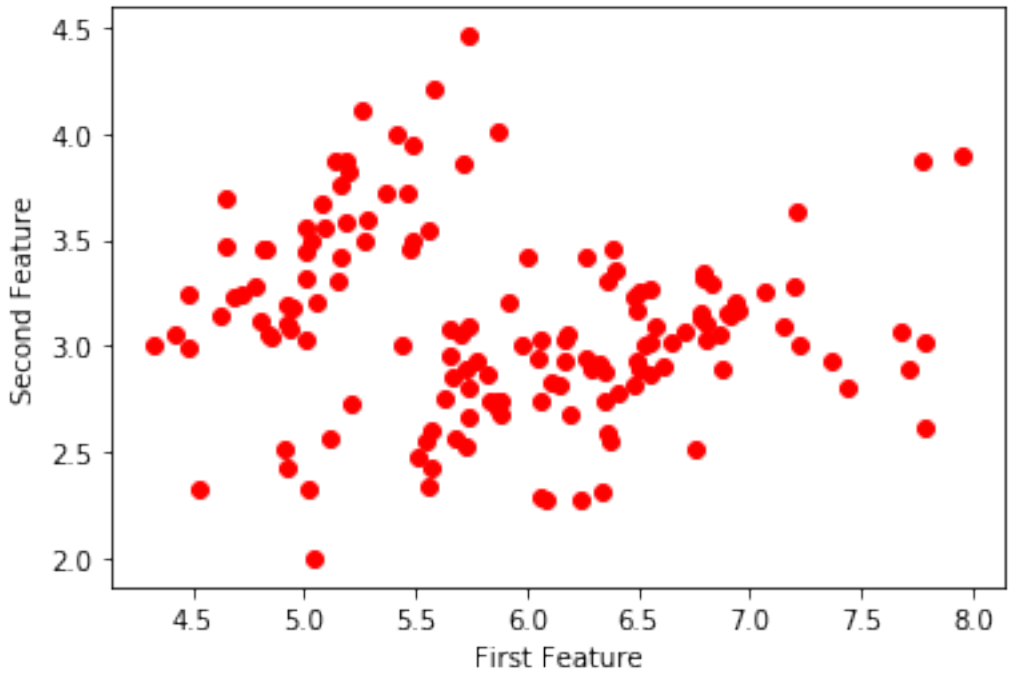
```
iris = np.genfromtxt('data/iris.txt', delimiter=None)
```

In [3]:

```
# first two features are asked in the question!
X = iris[:, 0:2]
Y = iris[:,-1]
```

In [4]:

```
plt.scatter(X[:,0],X[:,1],color='r')
plt.xlabel('First Feature')
plt.ylabel('Second Feature')
plt.show()
```



In [5]:

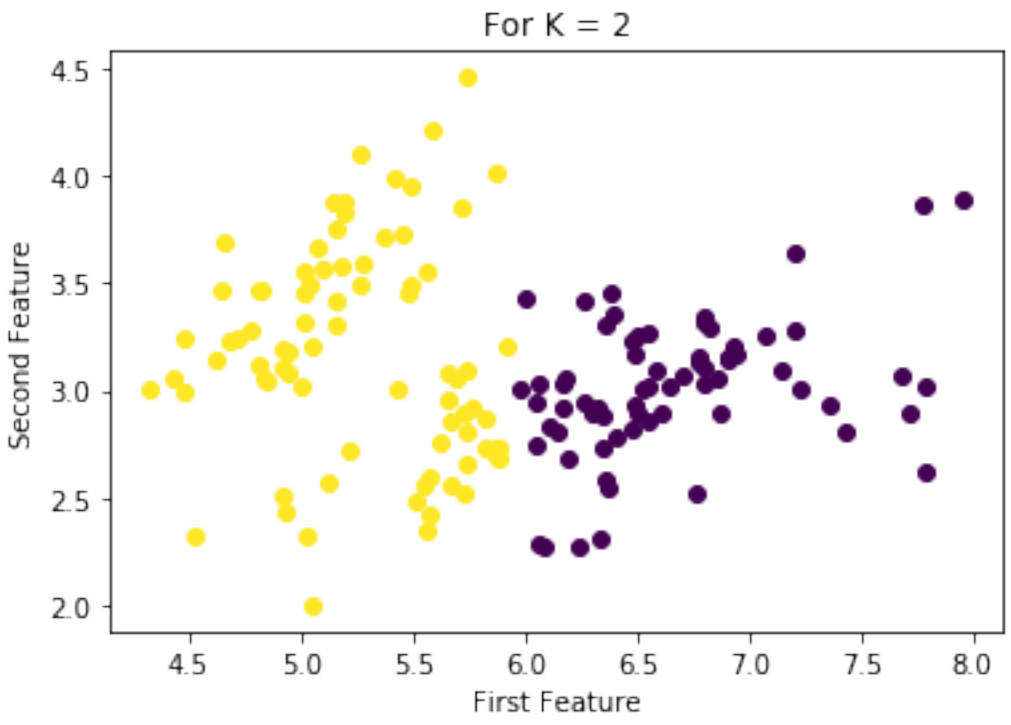
```
print("How many clusters you think exist, and briefly explain wh
y : \n")

print("I think that there is two clusters, and the main reason i
s that if we draw \
a fine line between the two parts that can be easily seperated,
we see that \
there exists two parts")
```

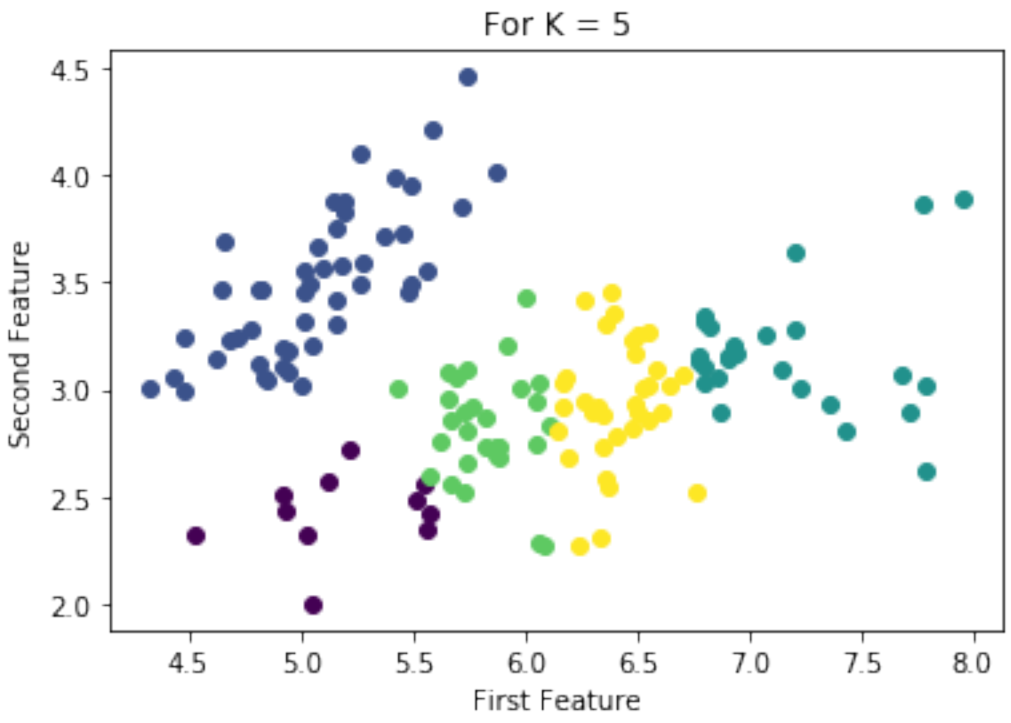
```
I think that there is two clusters, and the main rea
son is that if we draw a fine line between the two p
arts that can be easily seperated, we see that there
exists two parts
```

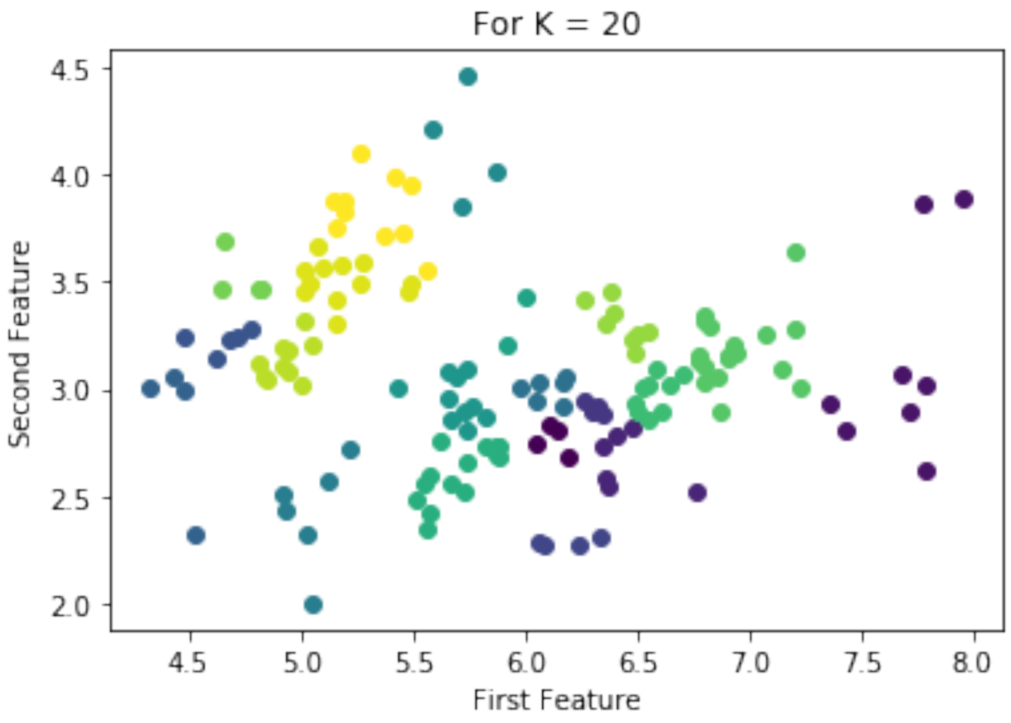
1. Run k-means on the first two features of the Iris data, for k = 2, k = 5, and k = 20. Try multiple (at least 5) different initializations for each k, and check to see whether they find the same solution; if not, pick the one with the best score. For the best clustering for each candidate k, create a plot with the data colored by assignment, and the cluster centers. You can plot the points colored by cluster assignments z using ml.plotClassify2D(None,X,z) . (You will need to also plot the cluster centers yourself.) (15 points)

In [6]:

```
a, b, c = ml.cluster.kmeans(X,2)
ml.plotClassify2D(None, X, a)
plt.xlabel('First Feature')
plt.title('For K = 2')
plt.ylabel('Second Feature')
plt.show()

a, b, c = ml.cluster.kmeans(X,5)
ml.plotClassify2D(None, X, a)
plt.xlabel('First Feature')
plt.title('For K = 5')
plt.ylabel('Second Feature')
plt.show()

a, b, c = ml.cluster.kmeans(X,20)
ml.plotClassify2D(None, X, a)
plt.xlabel('First Feature')
plt.title('For K = 20')
plt.ylabel('Second Feature')
plt.show()
```



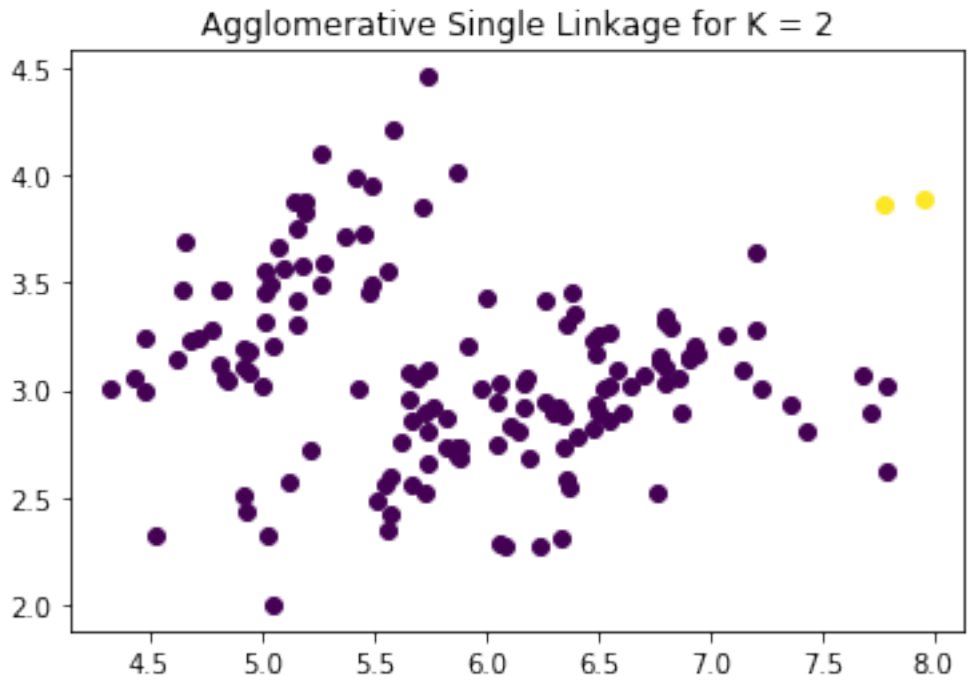


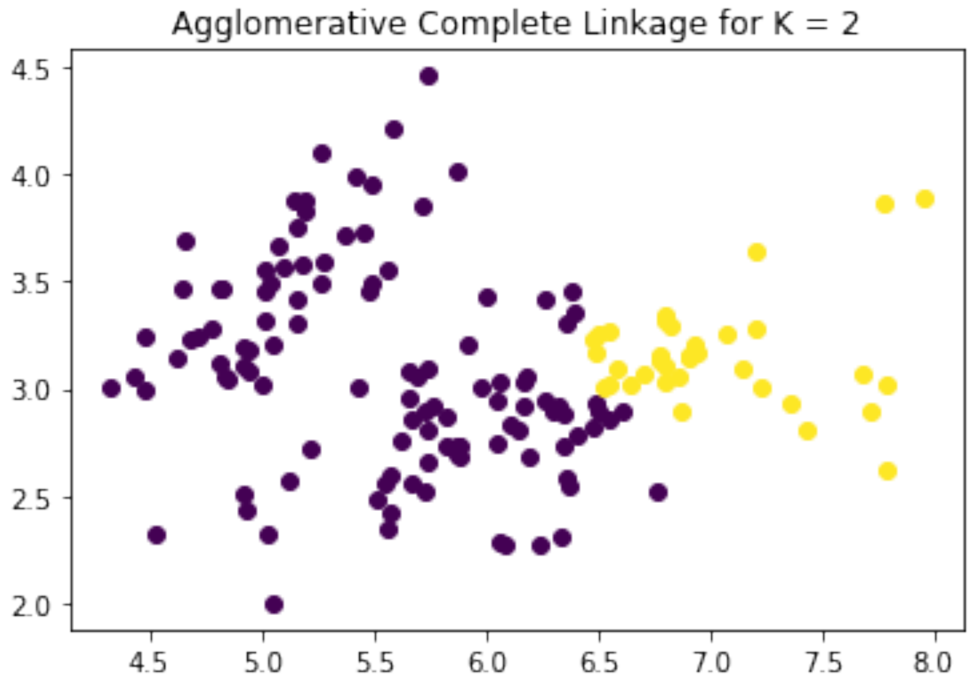


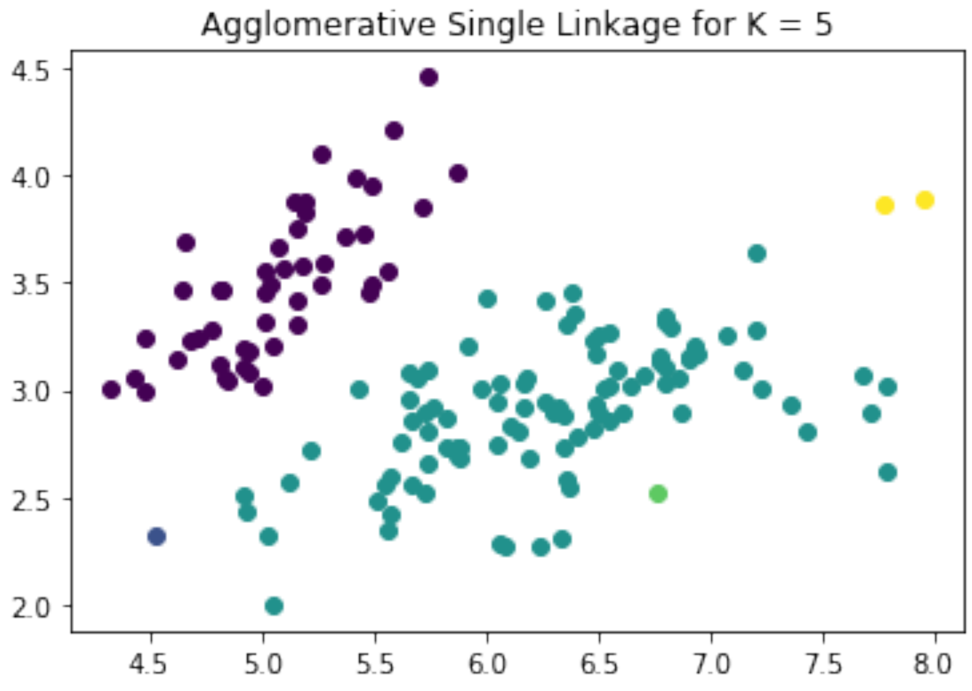
1. Run agglomerative clustering on the first two features of the Iris data, first using single linkage and then again using complete linkage, using the algorithms implemented in ml.cluster.agglomerative from cluster.py ). For each linkage criterion, plot the data colored by their assignments to 2, 5, and 20 clusters. (Agglomerative clustering does not require an initialization, so there is no need to run methods multiple times.) (15 points)

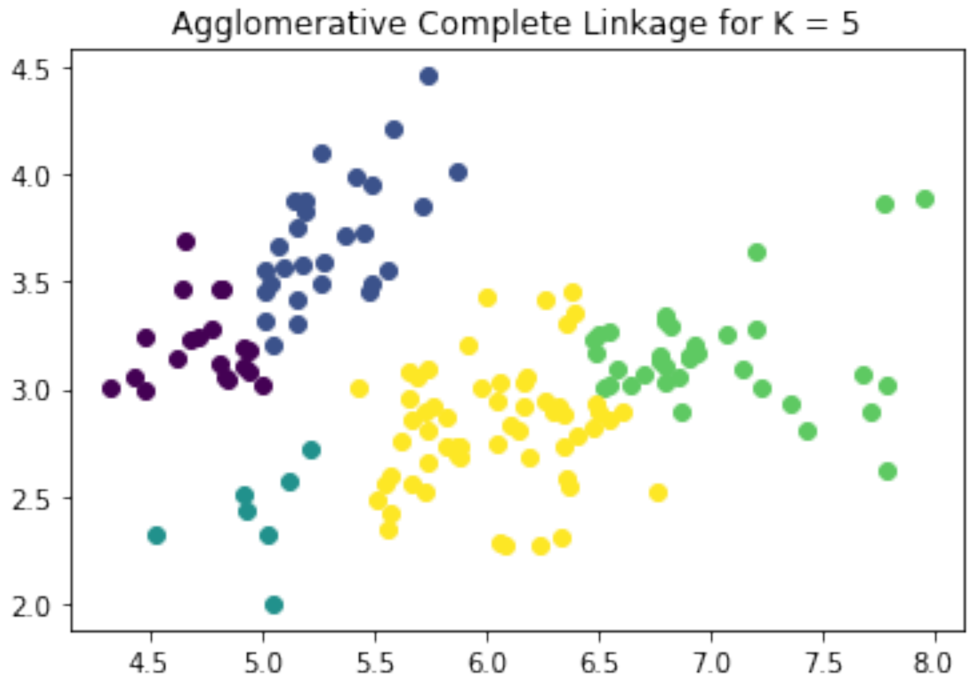
In [7]:

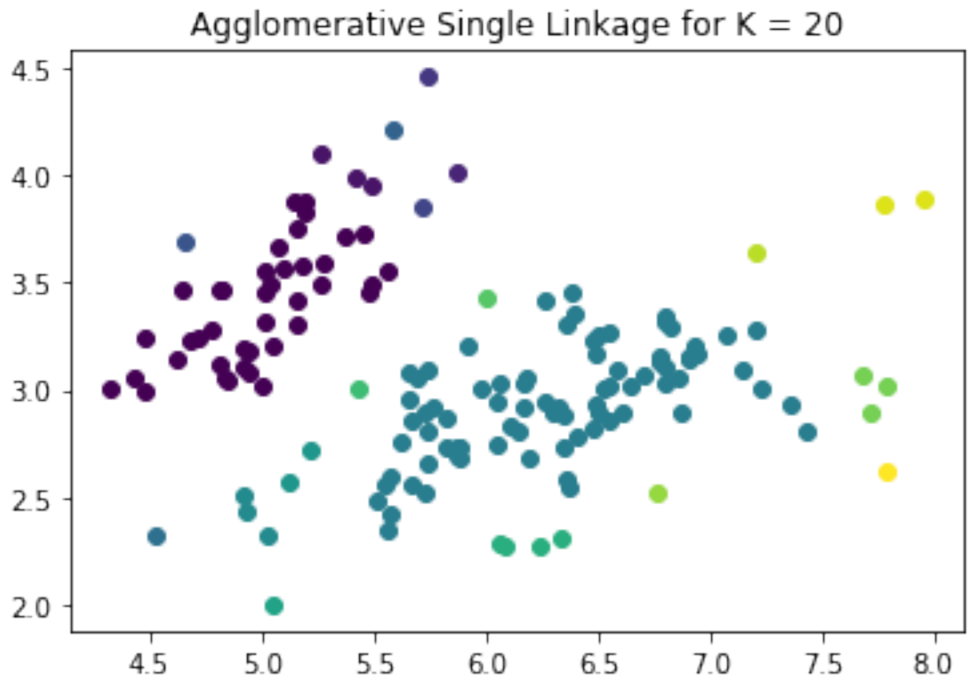
```
a, b = ml.cluster.agglomerative(X, 2, method='min')
plt.title("Agglomerative Single Linkage for K = 2");
ml.plotClassify2D(None, X, a);
plt.show()

a, b = ml.cluster.agglomerative(X, 2, method='max')
plt.title("Agglomerative Complete Linkage for K = 2");
ml.plotClassify2D(None, X, a);
plt.show()


a, b = ml.cluster.agglomerative(X, 5, method='min')
plt.title("Agglomerative Single Linkage for K = 5");
ml.plotClassify2D(None, X, a);
plt.show()

a, b = ml.cluster.agglomerative(X, 5, method='max')
plt.title("Agglomerative Complete Linkage for K = 5");
ml.plotClassify2D(None, X, a);
plt.show()

a, b = ml.cluster.agglomerative(X, 20, method='min')
plt.title("Agglomerative Single Linkage for K = 20");
ml.plotClassify2D(None, X, a);
plt.show()

a, b = ml.cluster.agglomerative(X, 20, method='max')
plt.title("Agglomerative Complete Linkage for K = 20");
ml.plotClassify2D(None, X, a);
plt.show()
```

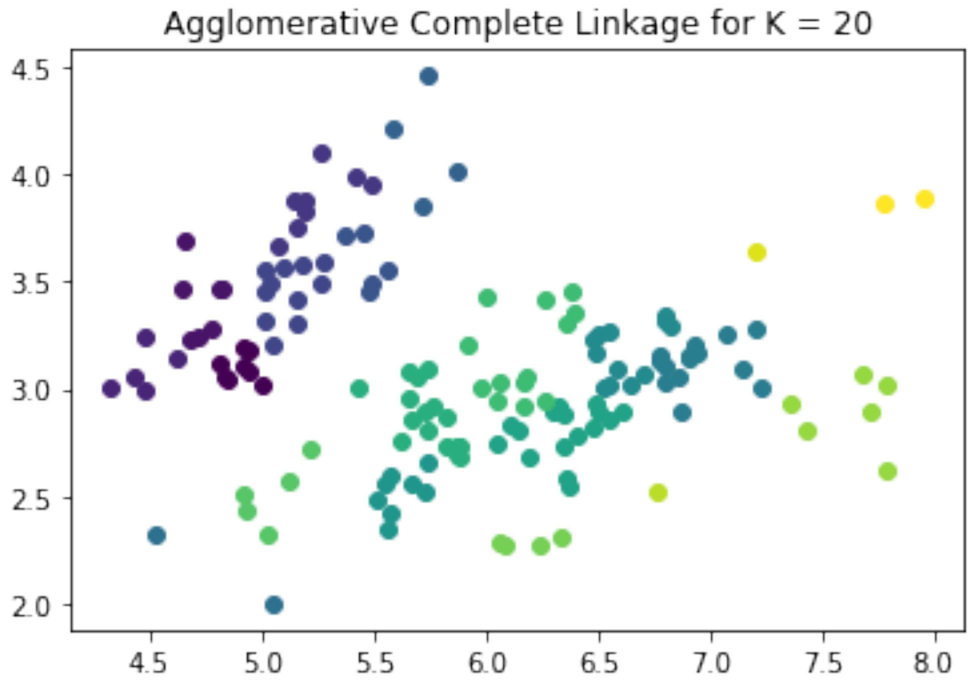

Agglomerative Single Linkage for K = 2
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


Agglomerative Complete Linkage for K = 2
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


Agglomerative Single Linkage for K = 5
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


Agglomerative Complete Linkage for K = 5
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0





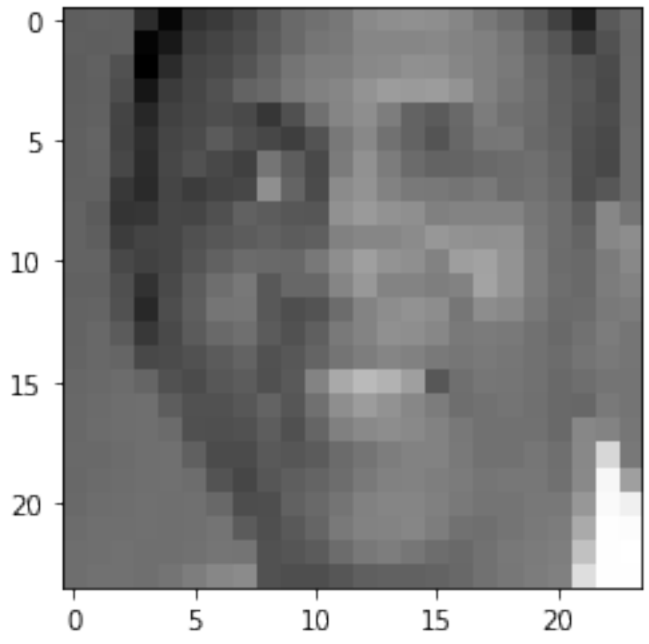
1. Briefly discuss similarities and differences in the outputs of the agglomerative clustering and k-means algorithms. (5 points)

One of the main differences between k-means and agglomerative clusters is that in case of agglomerative clusters, we see the dendograms if we look at the results we recieved previuosly. The important thing is that we can get min spanning tree if we utilize min distance between clusters. However, a maximum spanning tree avoids alongated clusters. As we have been shown previously, in the complete and in the single linkage for each, we can see that the in the case of single linkage, we have much less, in fact few, clusters that will take the majority of space and the most of them. However, the rest of them that are not part of this majority are much smaller and they might include single nodes. It is very important to know that k-means has a certain center point which will be chosen based on the look of the clusters, this can be based on distance or it can even be randomly chosen.

## EigenFaces

In [8]:

```
X = np.genfromtxt("data/faces.txt", delimiter=None) # load face
dataset plt.figure()
# pick a data point i for display
img = np.reshape(X[8,:],(24,24)) # convert vectorized data to 24
x24 image patches
plt.imshow( img.T , cmap="gray") # display image patch; you may
have to squint
plt.show()
```



1. Subtract the mean of the face images (X0 = X – μ) to make your data zero-mean. (The mean should be of the same dimension as a face, 576 pixels.) Plot the mean face as an image. (5 points)

In [9]:

```
mean = np.mean(X)
X0 = X-mean
print("X0 = ",X0)
```

```
X0 =  [[ -23.26595979  -19.26595979  -25.26595979 ..
.  -37.26595979
  -102.26595979 -110.26595979]
 [ -97.26595979 -100.26595979  -97.26595979 ...  -38
.26595979
   -75.26595979 -113.26595979]
 [-113.26595979 -111.26595979 -110.26595979 ...  -54
.26595979
   -56.26595979  -56.26595979]
 ...
 [ -99.26595979  -99.26595979 -100.26595979 ...  -91
.26595979
   -89.26595979  -84.26595979]
 [ -54.26595979  -52.26595979  -51.26595979 ...  -88
.26595979
   -88.26595979  -88.26595979]
 [ -62.26595979  -65.26595979  -64.26595979 ...  133
.73404021
   132.73404021  133.73404021]]
```

1. Use scipy.linalg.svd to take the SVD of the data, so that X0 =U·diag(S)·Vh Since the number of faces is larger than the dimension of each face, there are at most 576 non-zero singular values; use the full_matrices=False argument to avoid using a lot of memory. As in the slides, then compute W = U.dot( np.diag(S) ) so that X0 ≈ W · Vh. Print the shapes of W and Vh. (10 points)

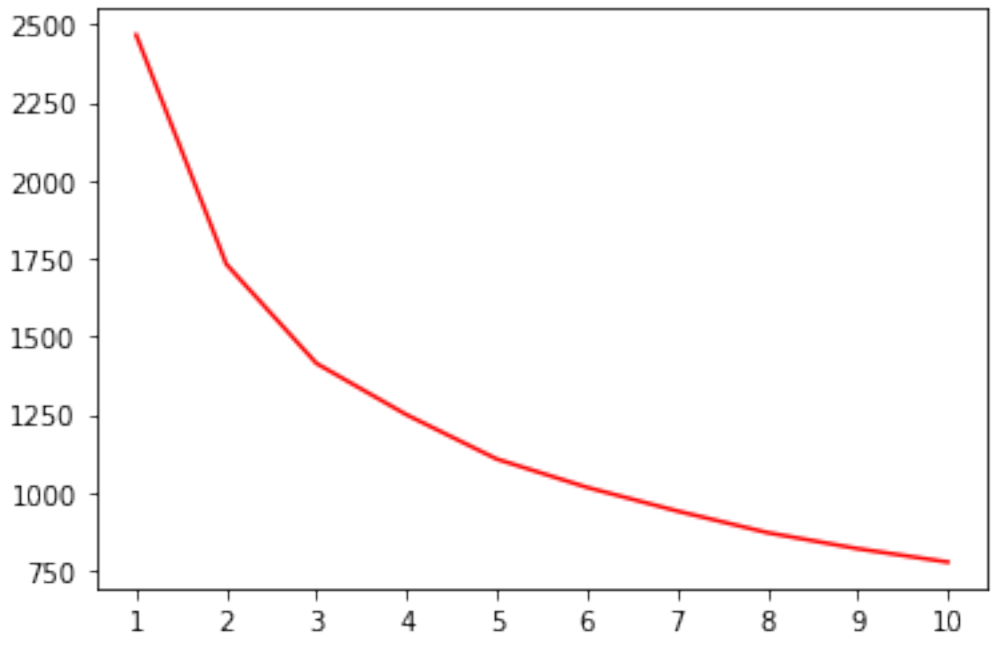
In [10]:

```
U, S, V = scipy.linalg.svd(X0, full_matrices=False)
W = U.dot(np.diag(S))
```

1. For K = 1,...,10, compute the approximation to X0 given by the first K eigenvectors (or eigenfaces): ˆ X0 = W[:,: K] · Vh[: K,:]. For each K, compute the mean squared error in the SVD's approximation, . Plot these MSE values as a function of K. (10 points)

In [11]:

```
mse = []
for k in range(1, 11):
    X0hat = W[:, :k].dot(V[:k,:])

    mse.append(np.mean((X0 - X0hat)**2))

_new, axis = plt.subplots()

axis.plot(range(1,11), mse, c='red')

axis.set_xticks(range(1,11))

plt.show()
```



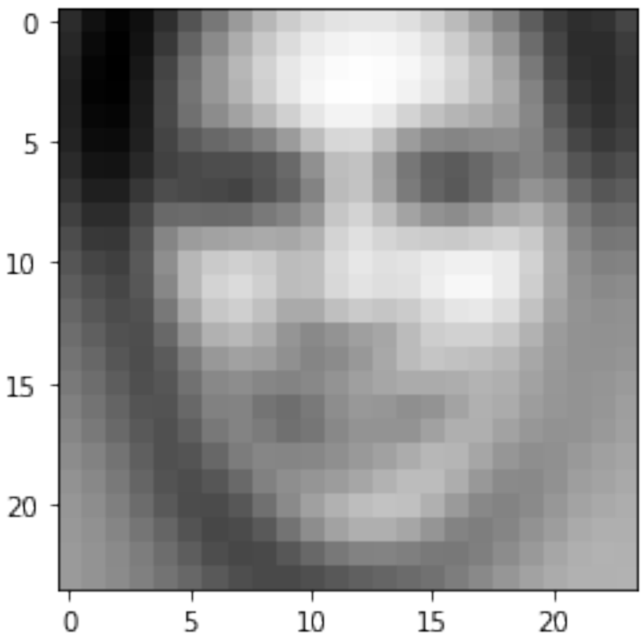
1. Display the first three principal directions of the data, by computing μ+α V[j,:] and μ-α V[j,:], where α is a scale factor (we suggest setting α to 2*np.median(np.abs(W[:,j])) , to match the scale of the data). These should be vectors of length 242 = 576, so you can reshape them and view them as “face images” just like the original data. They should be similar to the images in lecture. (10 points)

1. Choose any two faces and reconstruct them using the first K principal directions, for K = 5, 10, 50, 100. Plot the reconstructed faces as images. (5 points)

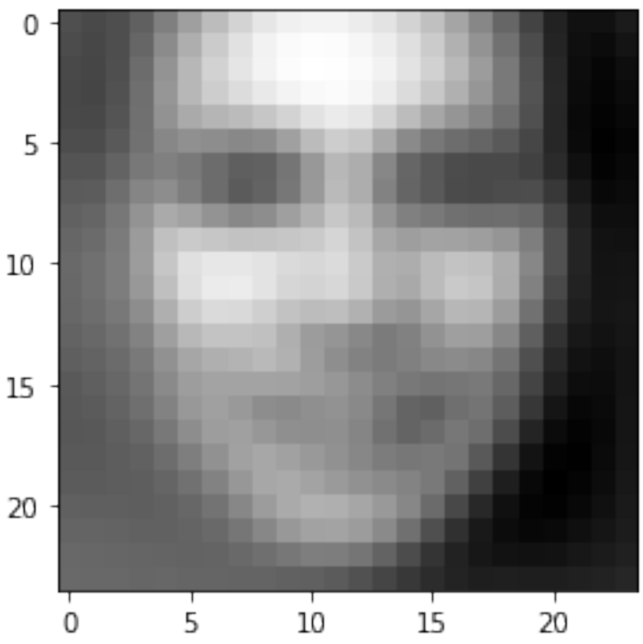
In [12]:

```
K = [5, 10,50,100]
# for each k that we have
for k in K:
    X0hat = W[:, :k].dot(V[:k,:])

    f1 = X0hat[5,:]
    f2 = X0hat[6,:]

    img = np.reshape(f1,(24,24))
    plt.imshow(img.T, cmap="gray")

    plt.title("Face 1 for K = " + str(k))
    plt.show()
    img = np.reshape(f2, (24,24))
    plt.imshow(img.T, cmap="gray")
    plt.title("Face 2 for K = " + str(k))
    plt.show()
```

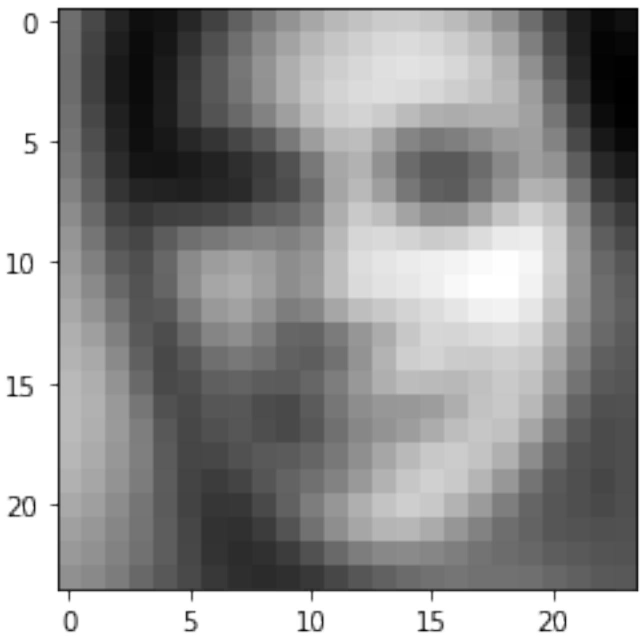
Face 1 for K = 5



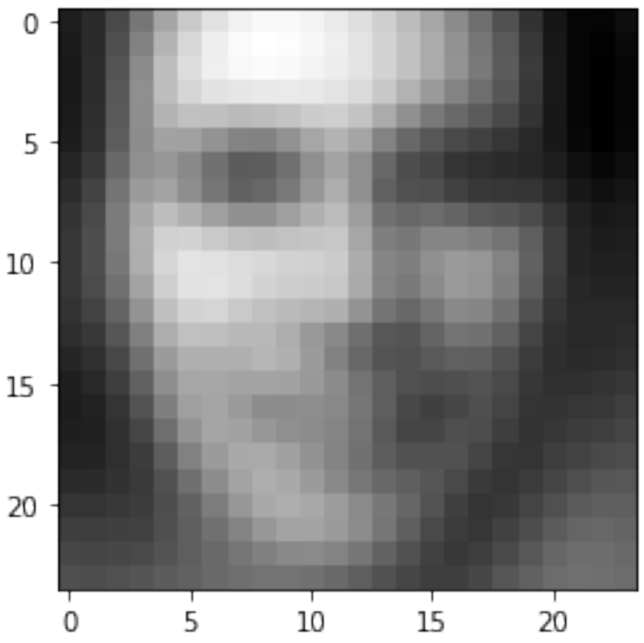
Face 2 for K = 5



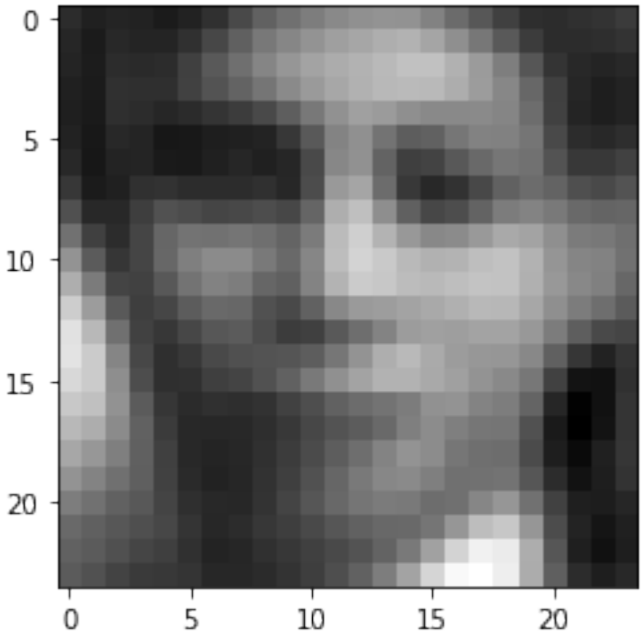
Face 1 for K = 10



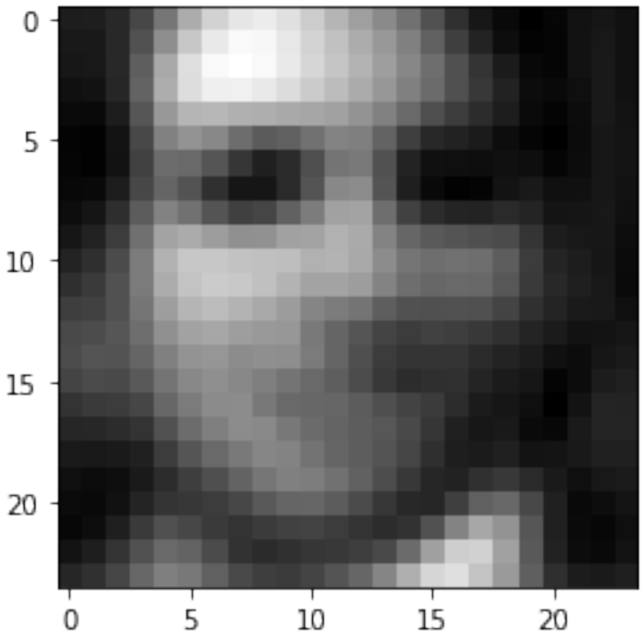
Face 2 for K = 10

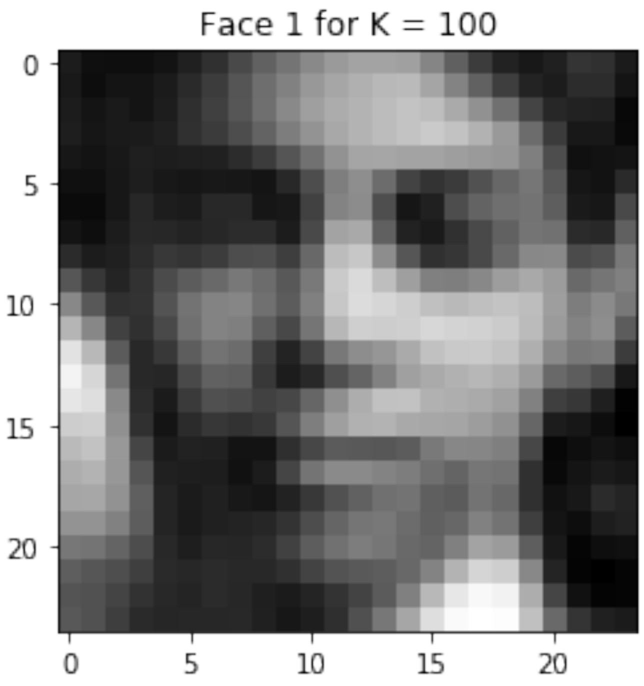


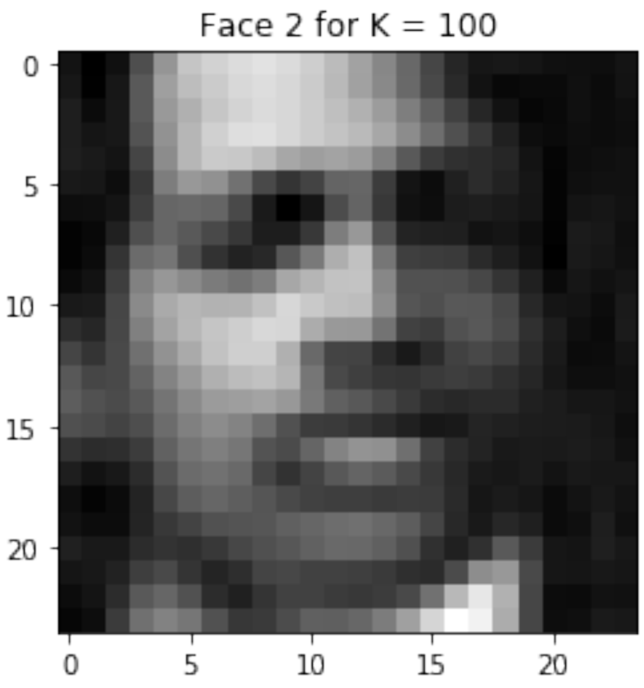
Face 1 for K = 50



Face 2 for K = 50





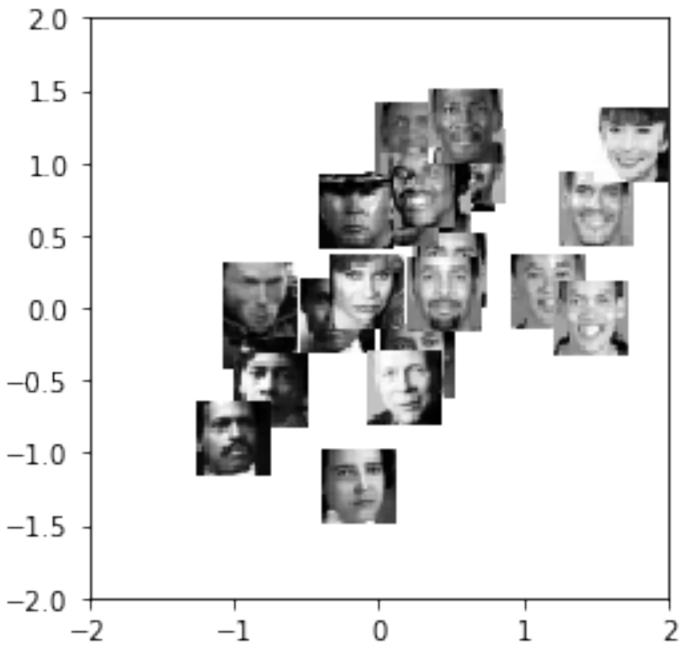


1. Methods like PCA are often called “latent space” methods, as the coefficients can be interpreted as a new geometric space in which the data are represented. To visualize this, choose 25 of the faces, and display them as images with the coordinates given by their coefficients on the first two principal components:

In [13]:

```
idx = [i for i in range(25)] # pick some data (randomly or other
wise); an array of integer indices
print(idx)
import mltools.transforms
coord,params = ml.transforms.rescale( W[:,0:2] ) # normalize sca
le of "W" locations
plt.figure();
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( X[i,:], (24,24) ) # reshape to square
    plt.imshow( img.T , cmap="gray", extent=loc ) # draw each im
age
    plt.axis( (-2,2,-2,2) ) # set axis to a reasonable scale
plt.show()
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1
5, 16, 17, 18, 19, 20, 21, 22, 23, 24]
```


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2
−1
0
1
2