Answers

December 14, 2017

In [5]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg
from numpy import matlib
%matplotlib inline
import pickle
from sklearn.manifold import TSNE
from sklearn import datasets
from matplotlib import offsetbox
from sklearn.metrics.pairwise import euclidean_distances
```

In [129]:
```
def circles_example():
    """
    an example function for generating and plotting synthesised data.
    """

    t = np.arange(0, 2 * np.pi, 0.03)
    length = np.shape(t)
    length = length[0]
    circle1 = np.matrix([np.cos(t) + 0.1 * np.random.randn(length),
                         np.sin(t) + 0.1 * np.random.randn(length)])
    circle2 = np.matrix([2 * np.cos(t) + 0.1 * np.random.randn(length),
                         2 * np.sin(t) + 0.1 * np.random.randn(length)])
    circle3 = np.matrix([3 * np.cos(t) + 0.1 * np.random.randn(length),
                         3 * np.sin(t) + 0.1 * np.random.randn(length)])
    circle4 = np.matrix([4 * np.cos(t) + 0.1 * np.random.randn(length),
                         4 * np.sin(t) + 0.1 * np.random.randn(length)])
    circles = np.concatenate((circle1, circle2, circle3, circle4), axis=1)

    plt.plot(circles[0,:], circles[1,:], '.k')
    plt.show()

    return circles


def apml_pic_example(path='APML_pic.pickle'):
    """
    an example function for loading and plotting the APML_pic data.
```

```
    :param path: the path to the APML_pic.pickle file
    """

    with open(path, 'rb') as f:
        apml = pickle.load(f)

    plt.plot(apml[:, 0], apml[:, 1], '.')
    plt.show()

    return apml



def euclid(X, Y):
    """
    return the pair-wise euclidean distance between two data matrices.
    :param X: NxD matrix.
    :param Y: MxD matrix.
    :return: NxM euclidean distance matrix.

    """

    # The forum said that we can use the built in function for euclidean distances,
    # but I'm not sure I understood that correctly.
    # My own code is below, but it didn't work on high dimensional data because
    # there wasnt enough memory to do the repmat.

    return euclidean_distances(X, Y)

#     N,D = X.shape
#     M = Y.shape[0]

#     print('M;', M)
#     np.repeat(X.transpose(), M)
#     X = np.reshape(np.repeat(X.transpose(), M), (D, N, M))
#     Y = np.reshape(np.repeat(Y.transpose(),N, axis = 0), (D, N, M))

#     return np.sqrt(np.sum((X-Y)**2, axis = 0))


def euclidean_centroid(X):
    """
    return the center of mass of data points of X.
    :param X: a sub-matrix of the NxD data matrix that defines a cluster.
    :return: the centroid of the cluster.
    """
```

```
    return np.sum(X, axis = 0) / X.shape[0]


def kmeans_pp_init(X, k, metric):
    """
    The initialization function of kmeans++, returning k centroids.
    :param X: The data matrix.
    :param k: The number of clusters.
    :param metric: a metric function like specified in the kmeans documentation.
    :return: kxD matrix with rows containing the centroids.
    """

    N,D = X.shape
    centers_indeces = []
    # chose first center at random
    centers_indeces.append(np.random.randint(low = 0, high = N))

    for i in range(1,k):
        dists = metric(X, np.array(X[centers_indeces]))
        weights = np.amin(dists, axis = 1) ** 2
        weight_probs = weights / np.sum(weights)
        centers_indeces.append(np.random.choice(np.arange(N),
                                    replace = False, p = weight_probs))

    return X[centers_indeces]


def cost_function(X, clusterings, centroids):
    """
    Returns the cost for thr 'elbow' method.
    X: The NxD data matrix.
    clustering: A list of N-dimensional vectors, each representing the
                clustering of one of the iterations of K-means.
    centroids: A list of kxD centroid matrices, one for each iteration.
    """

    cost = 0
    for i in range(len(centroids)):
        cost += np.sum(np.linalg.norm(X[clusterings == i] - centroids[i])**2)

    return cost


def silhouette(X, clusterings, centroids):
    """
    Given results from clustering with K-means, return the silhouette measure of
    the clustering.
    :param X: The NxD data matrix.
```

```
    :param clustering: A list of N-dimensional vectors, each representing the
                clustering of one of the iterations of K-means.
    :param centroids: A list of kxD centroid matrices, one for each iteration.
    :return: The Silhouette statistic, for k selection.
    """

    N,D = X.shape
    dists = euclid(X, X)
    k = len(centroids)

    normalized_dists = np.zeros((k, N))
    for i in range(k):
        normalized_dists[i] = np.sum(dists[clusterings == i], axis = 0)
        / np.count_nonzero(clusterings == i)

    indexes = np.ones((k,N)).astype(bool)
    indexes[clusterings, range(N)] = False
    A = normalized_dists[clusterings, range(N)]
    B = np.amin(np.reshape(normalized_dists.transpose()[indexes.transpose()],
                           (N,-1)).transpose(), axis = 0)
    return np.sum(np.divide(B - A , np.maximum(A,B)))



def show_clustering(X, clustering, centroids):
    """
    Colors different clusters in different colors
    X: NxD matrix
    clustering: A list of N-dimensional vectors, each representing the
                clustering of one of the iterations of K-means.
    centroids: A list of kxD centroid matrices, one for each iteration.
    """
    k = len(centroids)
    colors = ['blue', 'yellow', 'green', 'pink', 'purple', 'orange', 'brown',
              'magenta', 'grey', 'cyan', 'blue', 'magenta', 'green', 'grey']
    fig,ax = plt.subplots()
    for i in range(k):
        Y = X[clustering==i]
        ax.scatter(Y[:,0], Y[:,1], color = colors[i])
    ax.scatter(centroids[:,0], centroids[:,1], color = 'black')



def kmeans(X, k, iterations=5, metric=euclid, center=euclidean_centroid,
           init=kmeans_pp_init, stat=silhouette):
    """
    The K-Means function, clustering the data X into k clusters.
    :param X: A NxD data matrix.
```

```
    :param k: The number of desired clusters.
    :param iterations: The number of iterations.
    :param metric: A function that accepts two data matrices and returns their
            pair-wise distance. For a NxD and KxD matrices for instance, return
            a NxK distance matrix.
    :param center: A function that accepts a sub-matrix of X where the rows are
            points in a cluster, and returns the cluster centroid.
    :param init: A function that accepts a data matrix and k, and returns k initial
    :param stat: A function for calculating the statistics we want to extract about
                the result (for K selection, for example).
    :return: a tuple of (clustering, centroids, statistics)
    clustering - A N-dimensional vector with indices from 0 to k-1, defining the clu
    centroids - The kxD centroid matrix.
    statistics - whatever data you choose to use for your statistics (silhouette by
    """

    N, D = X.shape
    centeroids = init(X, k, metric)
    for i in range(iterations):
        clustering = np.argmin(metric(X, centeroids), axis = 1)
        centeroids = np.zeros((k,D))
        for j in range(k):
            centeroids[j] = center(X[clustering == j])

    return clustering, centeroids, stat(X, clustering, centeroids)


def heat(X, sigma):
    """
    calculate the heat kernel similarity of the given data matrix.
    :param X: A NxD data matrix.
    :param sigma: The width of the Gaussian in the heat kernel.
    :return: NxN similarity matrix.
    """
    return np.exp(-(euclid(X, X)**2) / (2*(sigma**2)))


def mnn(X, m):
    """
    calculate the m nearest neighbors similarity of the given data matrix.
    :param X: A NxD data matrix.
    :param m: The number of nearest neighbors.
    :return: NxN similarity matrix.
    """

    N = np.shape(X)[0]

    distances = euclid(X, X)
```

```
    nearest = np.zeros((N,N))
    for i in range(N):
        sorted_indexes = np.argsort(distances[i])
        nearest[i,sorted_indexes[1:m+1]] = 1
    return nearest + nearest.transpose() - (nearest * nearest.transpose())


def spectral(X, k, similarity_param, similarity=heat):
    """
    Cluster the data into k clusters using the spectral clustering algorithm.
    :param X: A NxD data matrix.
    :param k: The number of desired clusters.
    :param similarity_param: m for mnn, sigma for the hear kernel.
    :param similarity: The similarity transformation of the data.
    :return: clustering, as in the kmeans implementation.
    """

    N = X.shape[0]
    W = similarity(X, similarity_param)
    D_sqrt = np.eye(N)*(1/np.sqrt(np.sum(W, axis = 1)))
    L = np.eye(N) - np.dot(np.dot(D_sqrt, W), D_sqrt)
    eigvals, eigvecs = np.linalg.eigh(L)

    clustering, centeroids, stats = kmeans(eigvecs[:,0:k], k)
    centeroids = np.zeros((k,X.shape[1]))
    for j in range(k):
        centeroids[j] = euclidean_centroid(X[clustering == j])

    return clustering, centeroids, eigvals



def produce_data(k, dim = 2, cov = 0.8, separation_param  = 7,
                 low = 50, high = 150):

    X = []
    means = []
    colors = ['blue', 'yellow', 'green', 'pink', 'purple', 'orange', 'brown',
              'magenta', 'grey', 'cyan', 'blue', 'magenta', 'green', 'grey']
    for i in range(k):
        mean = (np.random.choice(a = 5*k, size = dim))
        if len(means) > 0:
            dists = np.linalg.norm(np.array(means) - np.matlib.repmat(mean,
                                                    len(means), 1), axis = 1)
            while (np.any(dists < separation_param*cov)):
                mean = (np.random.choice(a = 5*k, size = dim))
                dists = np.linalg.norm(np.array(means) - np.matlib.repmat(mean,
                                                len(means), 1), axis = 1)
```

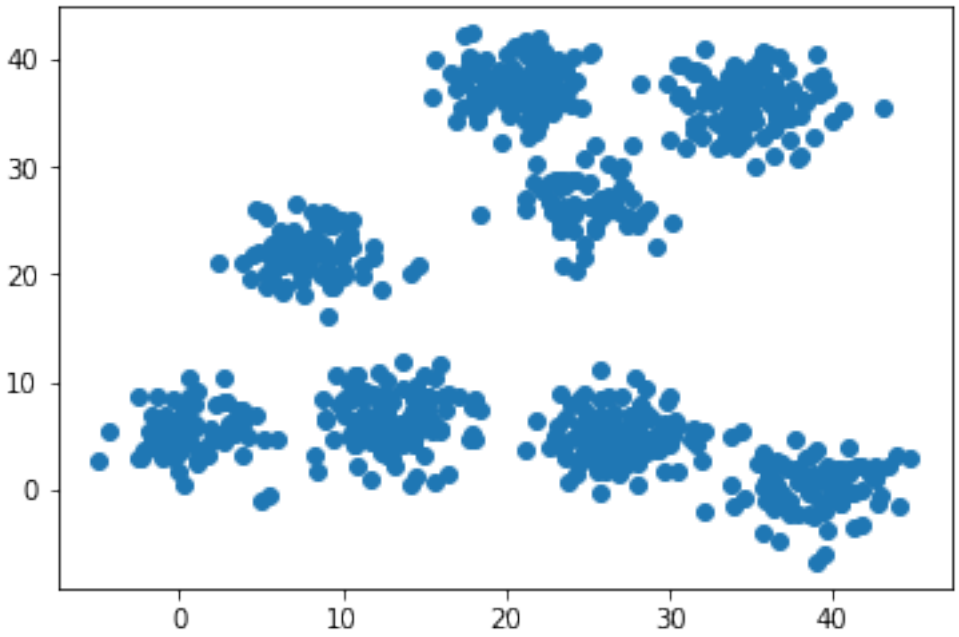
```
            means.append(mean)
            N = np.random.randint(low = low, high = high)
            Y = np.random.randn(dim,N)*cov + np.matlib.repmat(mean, N, 1).transpose()
            X.append(Y)

        X = np.concatenate(X, axis = 1)
        return X.transpose()

    def plot_embedding(X, title=None):
        x_min, x_max = np.min(X, 0), np.max(X, 0)
        X = (X - x_min) / (x_max - x_min)

        plt.figure()
        ax = plt.subplot(111)
        for i in range(X.shape[0]):
            plt.text(X[i, 0], X[i, 1], str(digits.target[i]),
                     color=plt.cm.Set1(y[i] / 10.),
                     fontdict={'weight': 'bold', 'size': 9})

        if hasattr(offsetbox, 'AnnotationBbox'):
            # only print thumbnails with matplotlib > 1.0
            shown_images = np.array([[1., 1.]])  # just something big
            for i in range(digits.data.shape[0]):
                dist = np.sum((X[i] - shown_images) ** 2, 1)
                if np.min(dist) < 4e-3:
                    # don't show points that are too close
                    continue
                shown_images = np.r_[shown_images, [X[i]]]
                imagebox = offsetbox.AnnotationBbox(
                    offsetbox.OffsetImage(digits.images[i], cmap=plt.cm.gray_r),
                    X[i])
                ax.add_artist(imagebox)
        plt.xticks([]), plt.yticks([])
        if title is not None:
            plt.title(title)


    def paramter_selection(params, func, data):
        """
        A function that iterates over list of parameters and plot the
        eigenvalues for each of the parameters.
        params - list of parameters
        func - a similarity function for spectral kmeans
        data - NxD data matrix
        """
        n = int(np.sqrt(len(params)))
        fig, ax = plt.subplots(nrows = n, ncols = n)
        fig.set_size_inches((12,9))
```

```
            for i in range(len(params)):
                stat = spectral(data, 2, params[i], similarity=func)[2]
                ax[int(i/n)][i%n].plot(stat[:20], 'o')
                ax[int(i/n)][i%n].set_title(('param:,', "%.2f" % params[i]))
```

1 Measures For k - Selection

```
In [59]: # produce some structured syntetic data with 8 clusters
         X = produce_data(8, cov = 2.3, separation_param=4.5)
         plt.scatter(X[:,0], X[:,1])
```

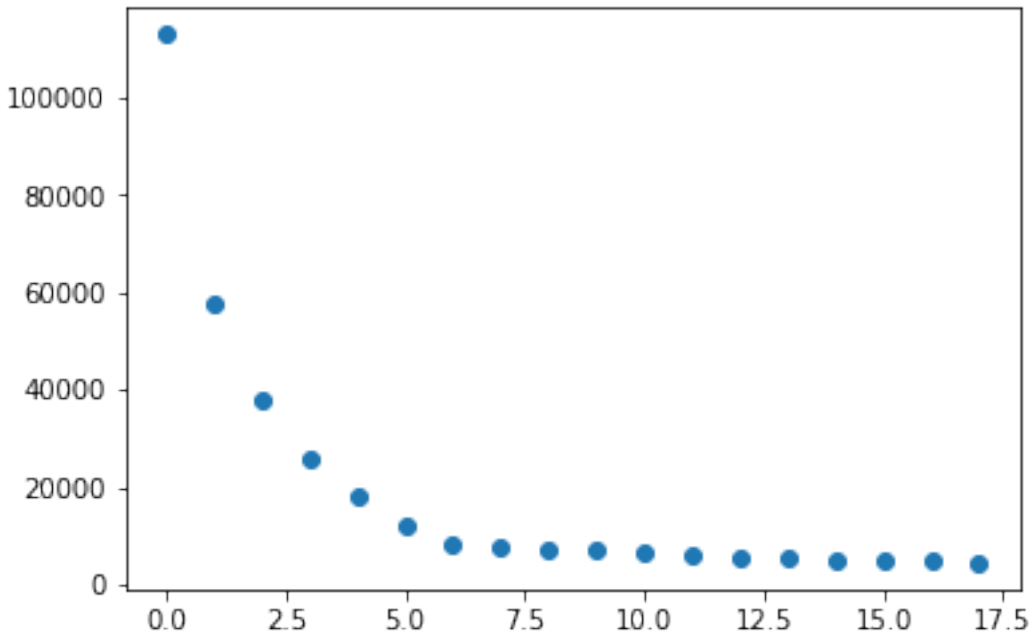
```
Out[59]: <matplotlib.collections.PathCollection at 0x113cbce4780>
```



The 'Elbow' method:

```
In [29]: best_cost = []
         for k in range(2,20):
             cost = []
             for i in range(10):
                 cost.append(kmeans(X, k, stat=cost_function)[2])
             best_cost.append(np.min(cost))
         plt.plot(best_cost, 'o')
```

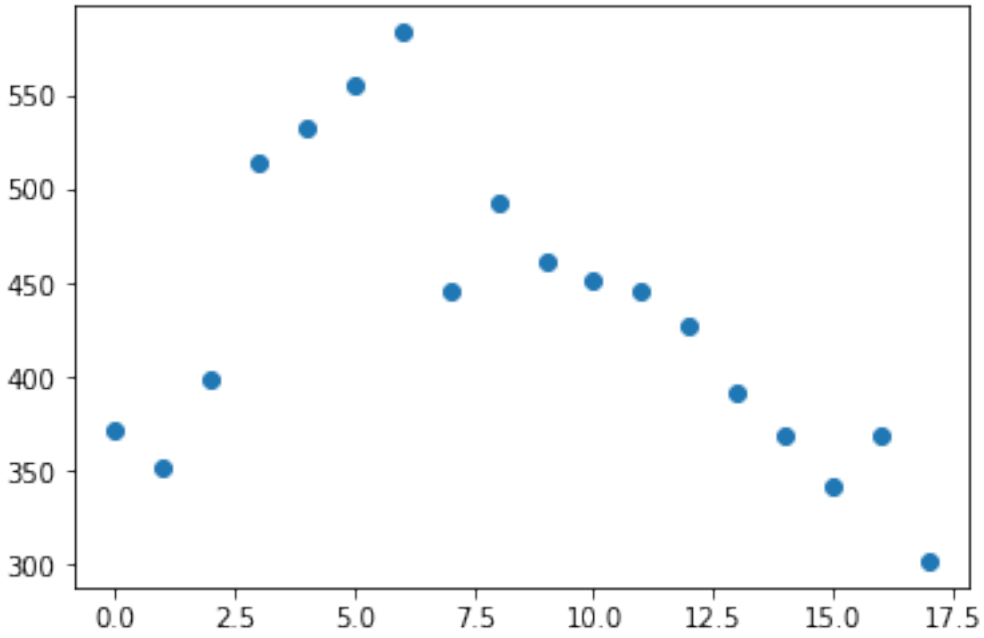
```
Out[29]: [<matplotlib.lines.Line2D at 0x113c8a1f710>]
```



As we can see, the 'elbow' indeed occurs at k=8 (the sixth point)- until that point the slope is steep but the steepness is degrading, and after reaching the break the cost function degrades slowly.

Silhouette:

```
In [43]: sil = []
         for k in range(2,20):
             sil.append(kmeans(X, k)[2])
         plt.plot(sil, 'o')

Out[43]: [<matplotlib.lines.Line2D at 0x113c8ed4668>]
```

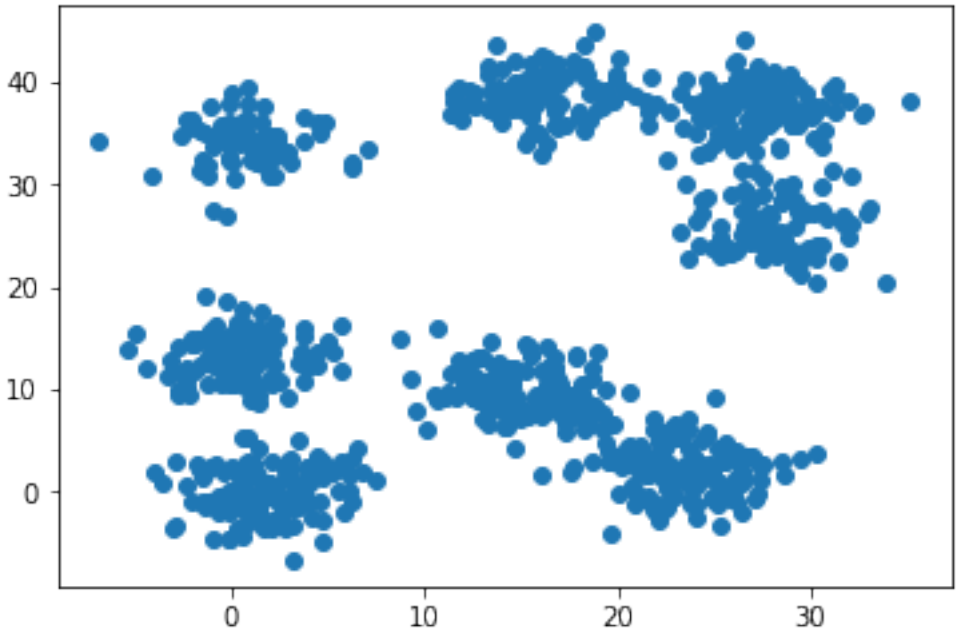


The clear peak is at k = 8 (6th point).

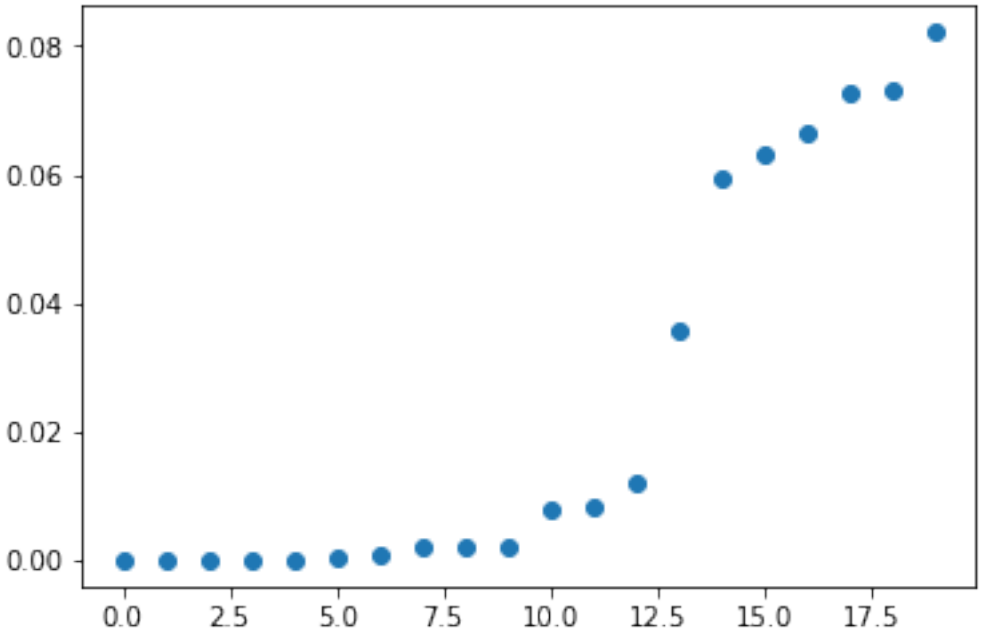
Eigen-gap: Now we will produce similar data with 8 clusters, but not as well separated.

```
In [61]: X = produce_data(8, cov = 2.3, separation_param=4)
         plt.scatter(X[:,0], X[:,1])
```

```
Out[61]: <matplotlib.collections.PathCollection at 0x113cbda5048>
```

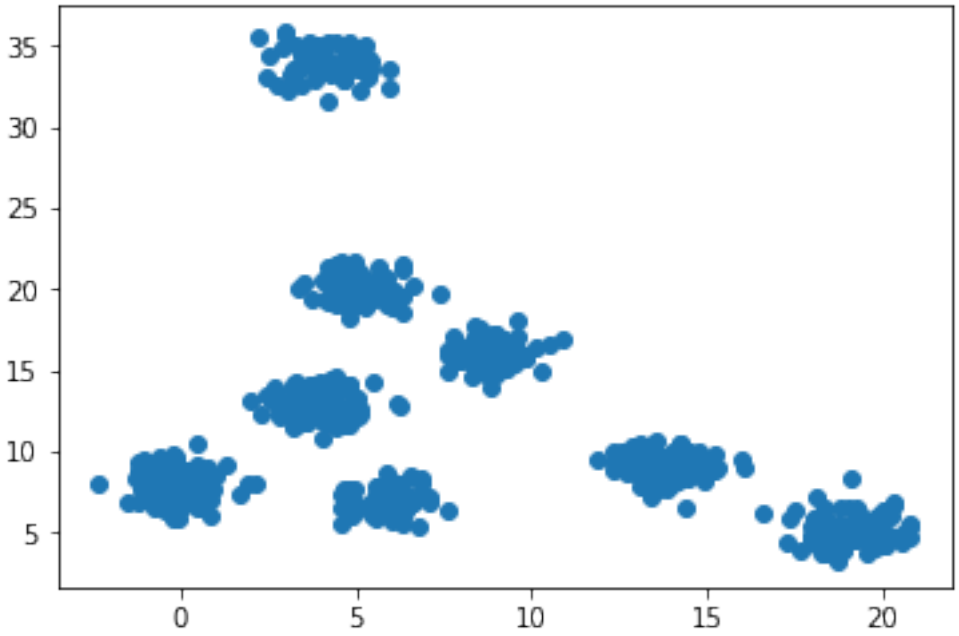


```
In [62]: eigenvals = spectral(X, k, 1)[2]
         plt.plot(eigenvals[:20], 'o')

Out[62]: [<matplotlib.lines.Line2D at 0x113cbdef9b0>]
```



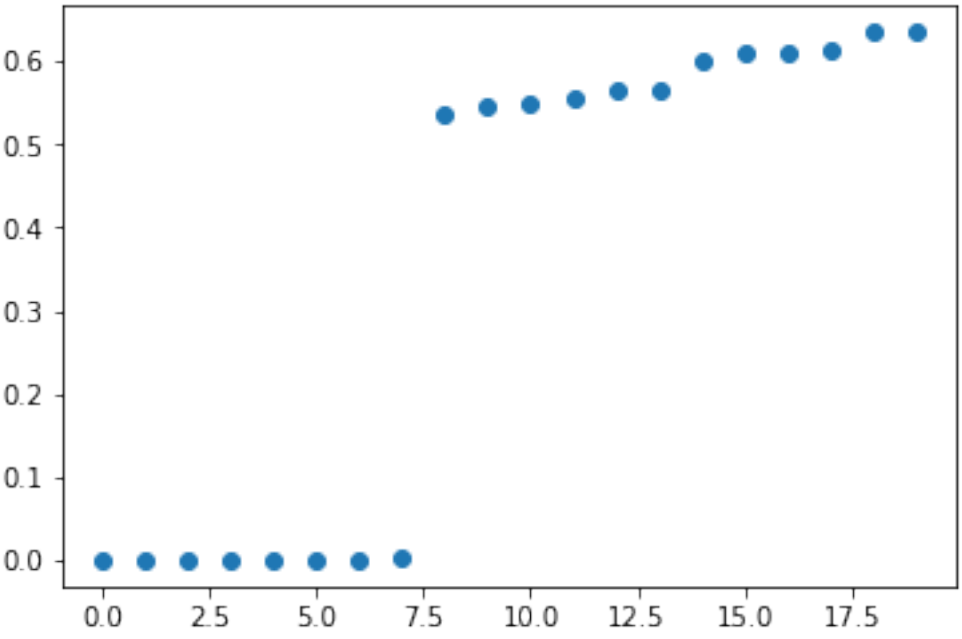
We can see that there is a gap after the 8th eigenvalue, but it is relatively small. Now we will produce well separated data:

```
In [54]: Y = produce_data(8)
         plt.scatter(Y[:,0], Y[:,1])

Out[54]: <matplotlib.collections.PathCollection at 0x113cbb10a20>
```




```
In [56]: eigenvals = spectral(Y, k, 1)[2]
         plt.plot(eigenvals[:20], 'o')
```

```
Out[56]: [<matplotlib.lines.Line2D at 0x113cbbd1588>]
```

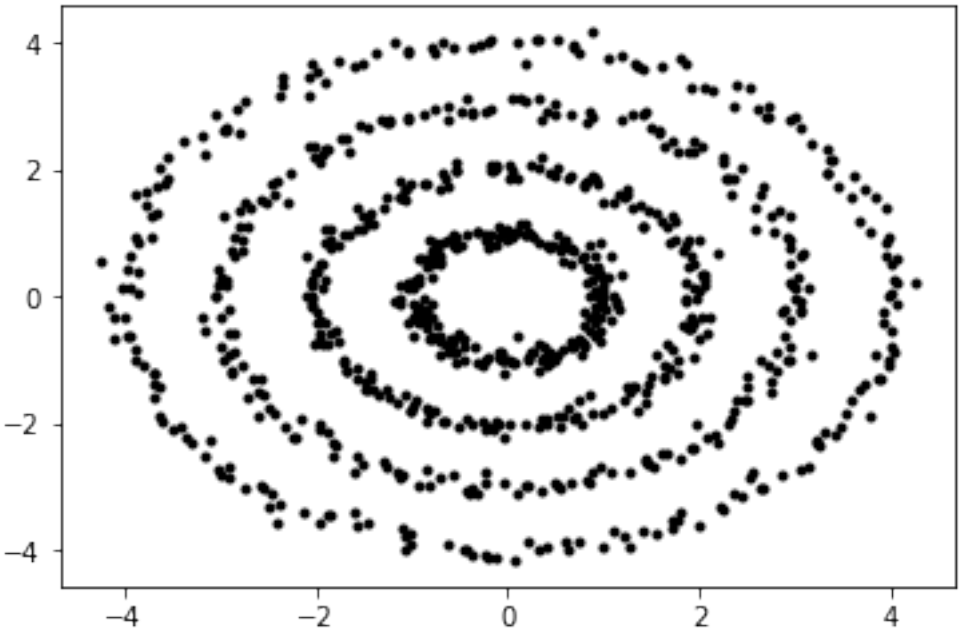


We can see that the gap is much clearer with well separated clusters in our data.

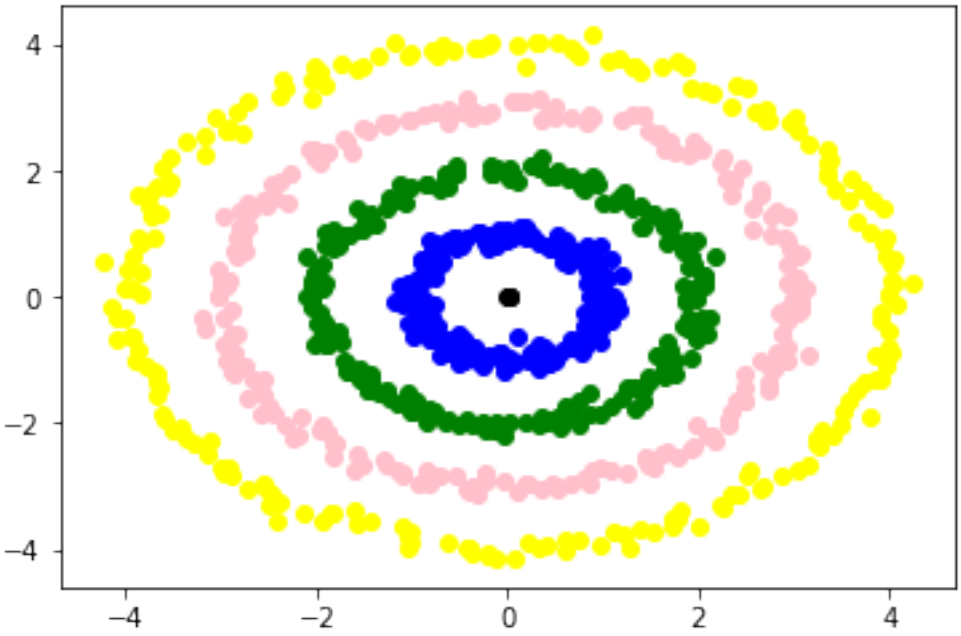
2 Spectral Clustering:

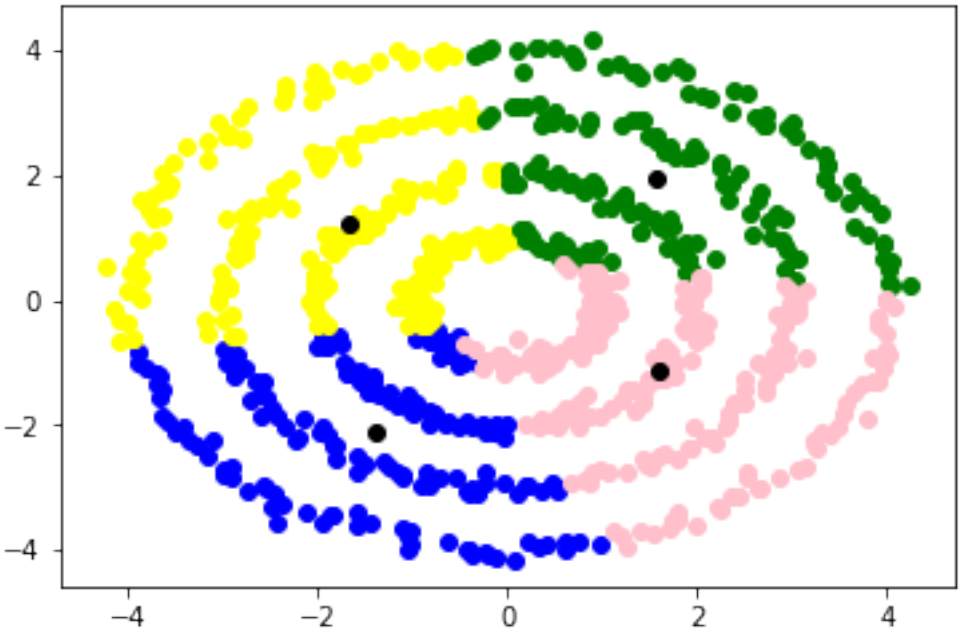
Circles

```
In [66]: circles = np.array(circles_example().transpose())
```




```
In [68]: a,b,c = spectral(circles, 4, 6, similarity = mnn)
         show_clustering(circles, a, b)

         a,b,c = kmeans(circles, 4)
         show_clustering(circles, a, b)
```

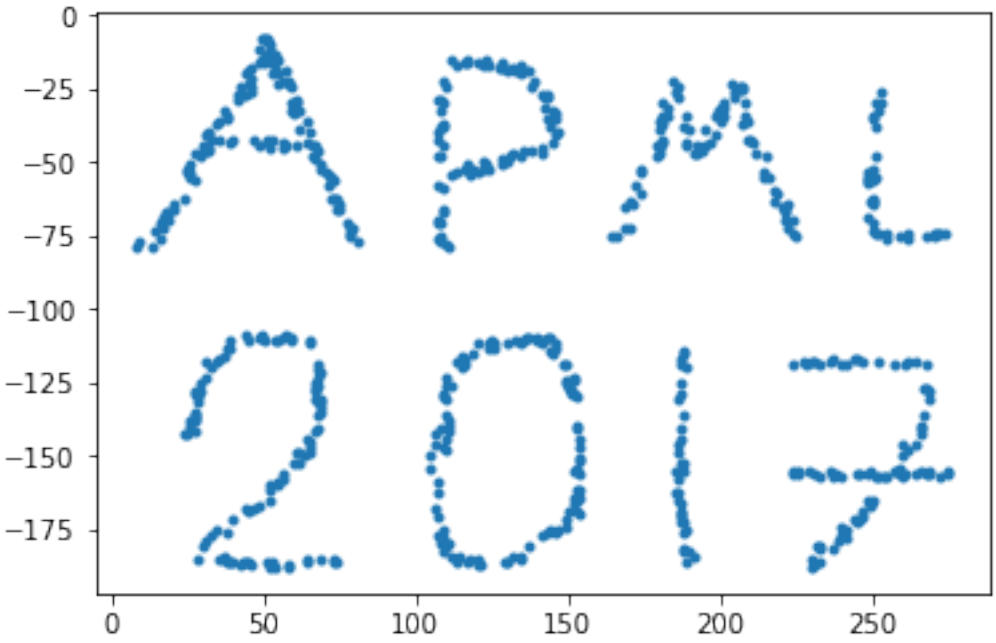




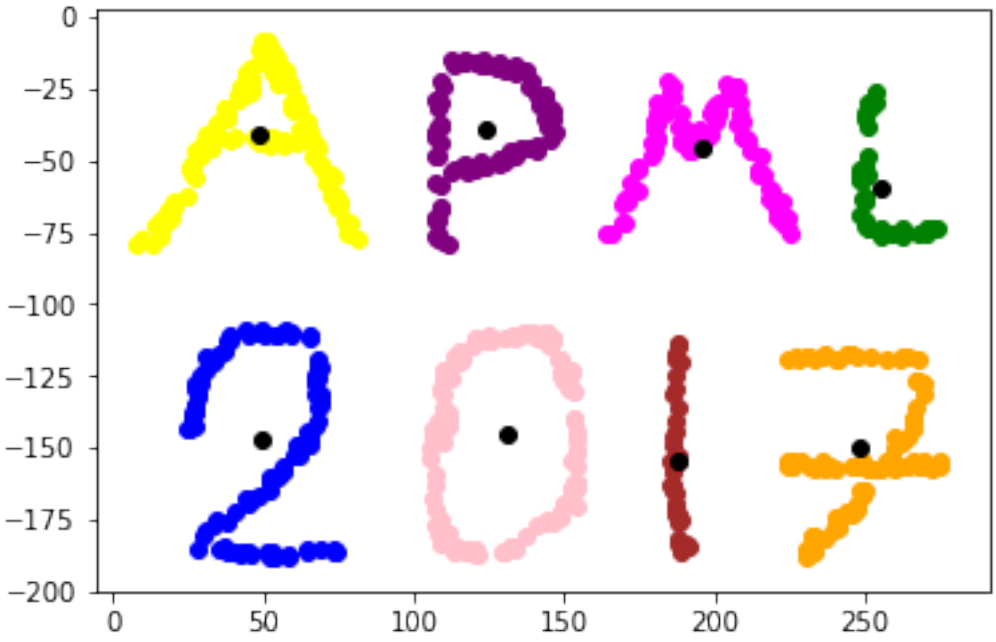
The spectral clustering successfully clusters the data into rings, when the regular k-means divides the picture into 4 relatively equal parts. That is because k-means considers the distances from the centers (the black dots in the figure) and assigns everything that is closest to the specific centroid to one cluster, whether the spectral k-means considers the relationship between the datapoints themselves, and clusters them according to their distances between themselves.

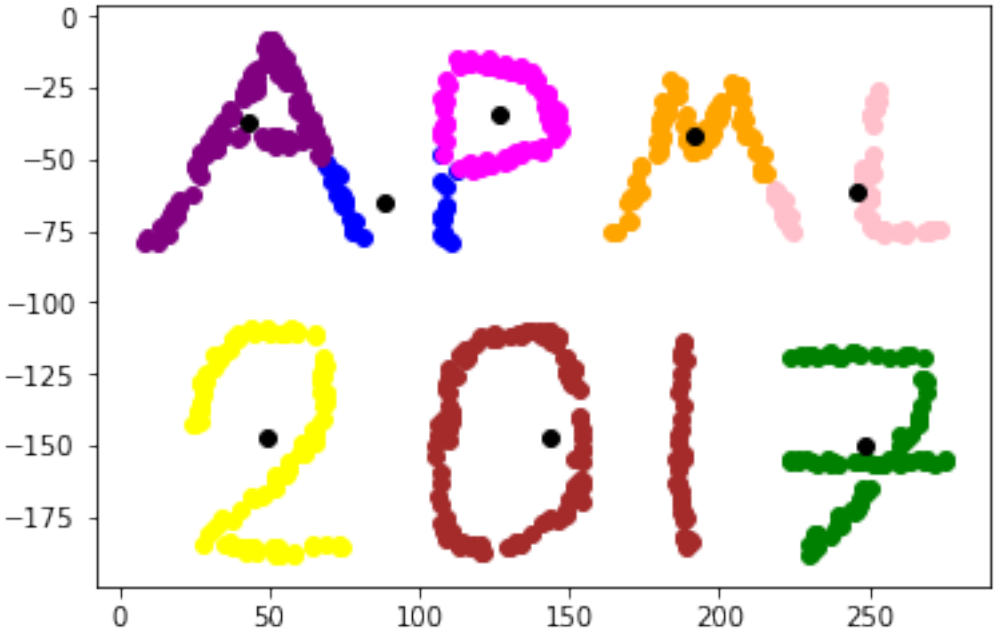
APML

In [69]:
```
apml = apml_pic_example()
```



In [72]:
```
a,b,c = spectral(apml, 8,  2, similarity = heat)
show_clustering(apml, a, b)

a,b,c = kmeans(apml, 8)
show_clustering(apml, a, b)
```

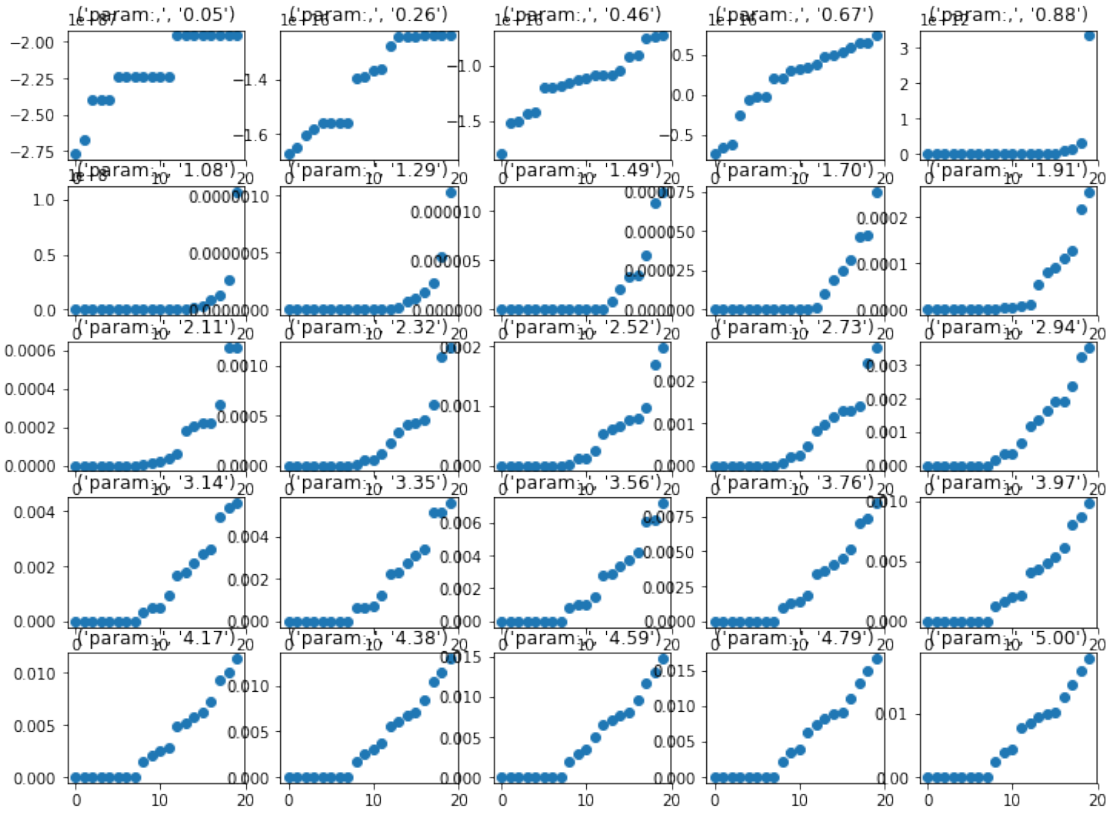




Again, the spectral clustering algorithm did a better job clustering the data to relevant clusters, and not just according to distance to centeroids.

Parameter Selection: Now I will demonstrate the parameter selection proccess for the APML dataset with heat kernel. The code iterates over a list of possible sigma parameters for the heat kernel and plots the eigengap graphs. The most obvious eigengap is after the 8th point, for sigmas over 3.

```
In [117]: sigmas = np.linspace(0.05, 5, 25)
          paramter_selection(params = sigmas, data = apml, func = heat)
```



3 Microarray:

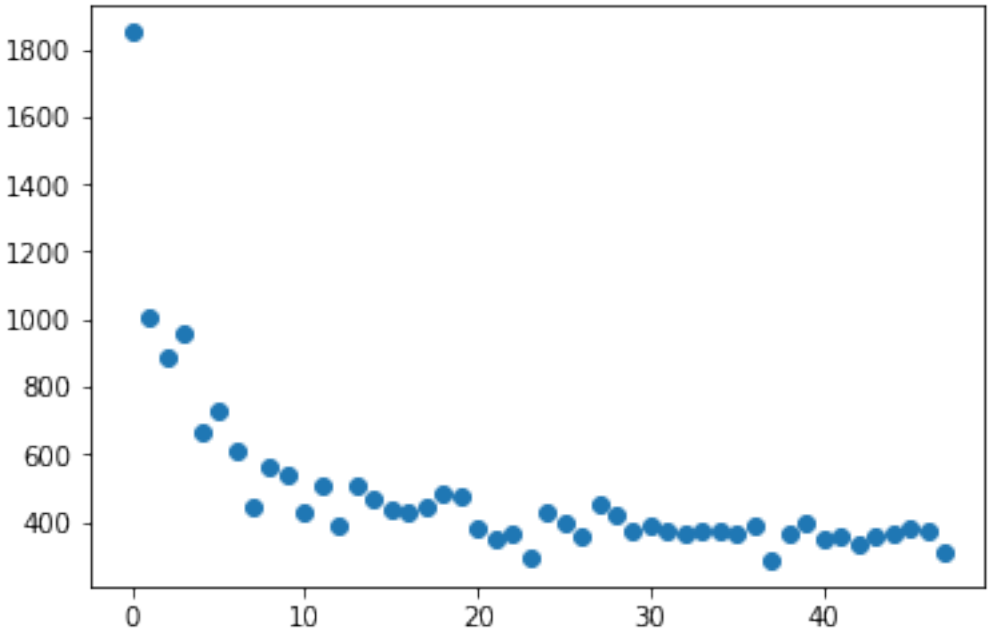
```
In [104]: data_path='microarray_data.pickle'
          genes_path='microarray_genes.pickle'
          conds_path='microarray_conds.pickle'

          with open(data_path, 'rb') as f:
              data = pickle.load(f)
          with open(genes_path, 'rb') as f:
              genes = pickle.load(f)
          with open(conds_path, 'rb') as f:
              conds = pickle.load(f)
```

3.0.1 K-means

Parameter Selection I will use the silhouette method:

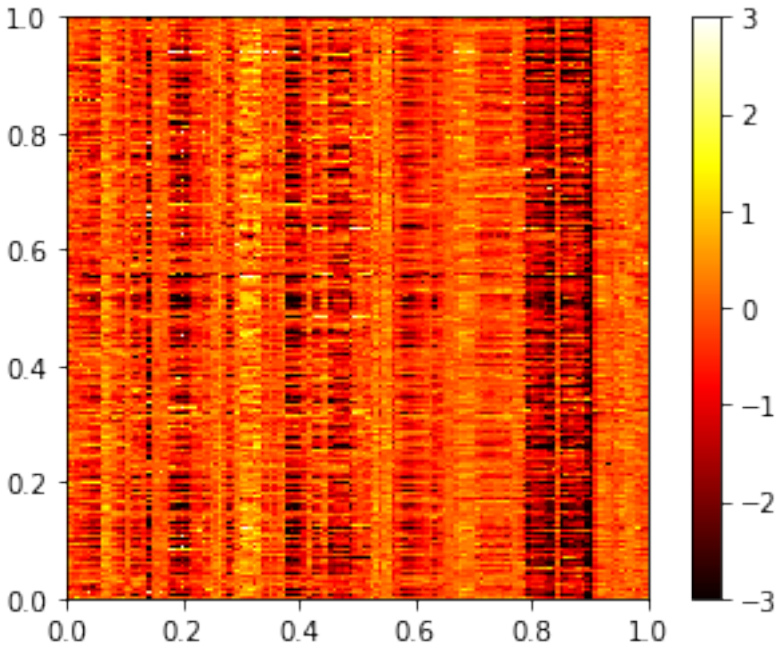
```
In [108]: sil = []
          for k in range(2,50):
              sil.append(kmeans(data, k)[2])
          plt.plot(sil, 'o')
```

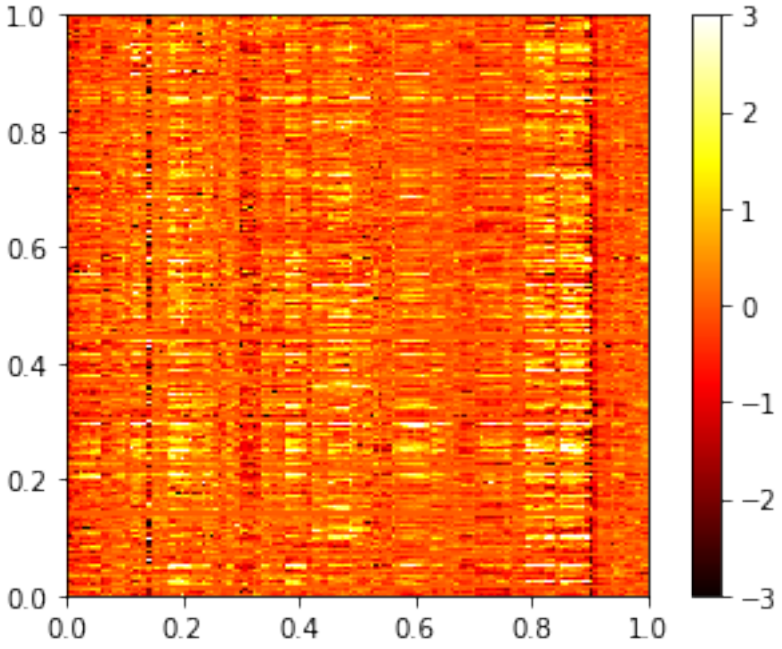
```
Out[108]: [<matplotlib.lines.Line2D at 0x113cc742cf8>]
```



The peak is at the first dot, which stands for k = 2, so we will run the kmeans algorithm for k=2.

```
In [110]: k_clusters, centers, stats = kmeans(data, 2)
          for i in range(2):
              plt.figure()
              plt.imshow(data[k_clusters == i],extent=[0, 1, 0, 1],
                         cmap="hot", vmin=-3, vmax=3)
              plt.colorbar()
              plt.show()
```



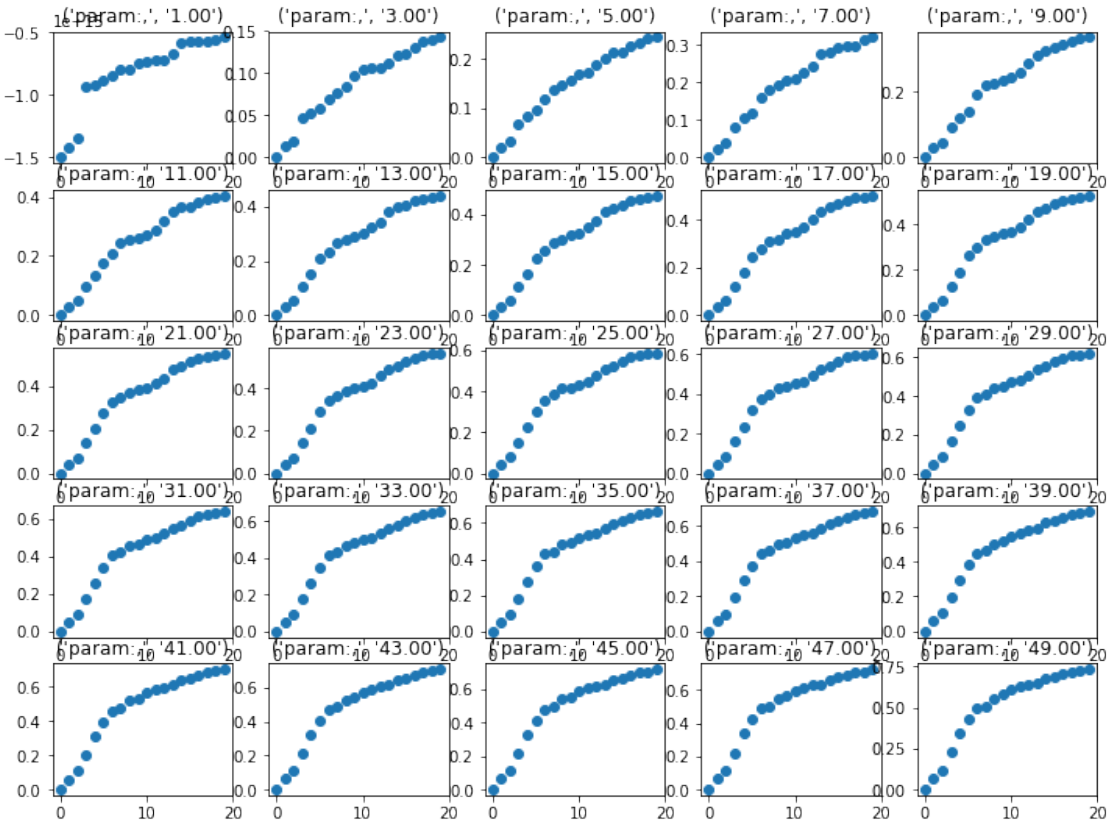


It seems that most of the values in the first cluster are below zero, and most of the values in the second cluster are above zero.

3.0.2 Spectral K-means

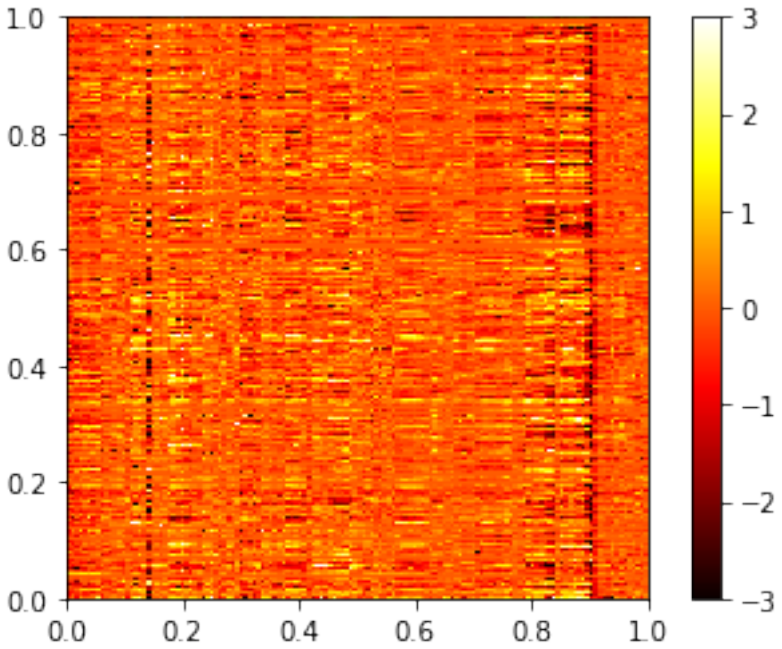
MNN - Parameter Selection

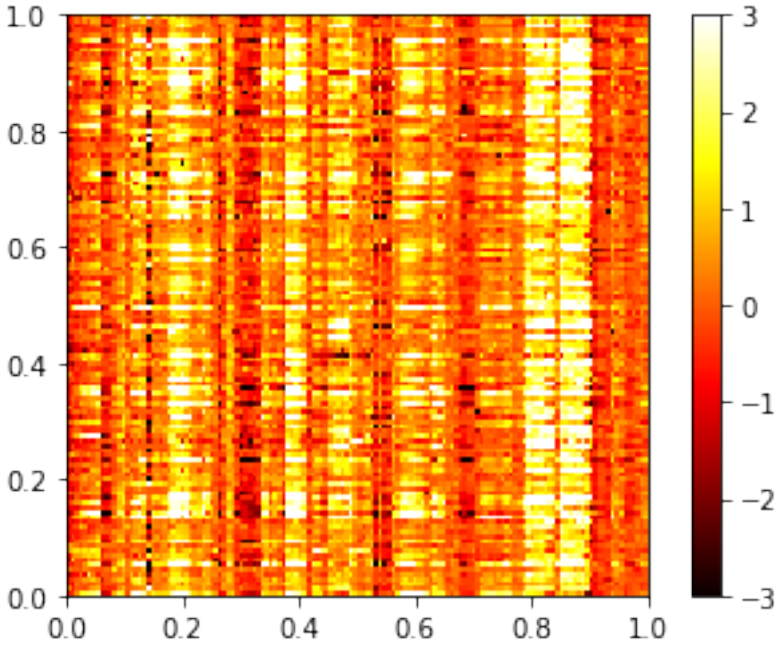
```
In [128]: n_nums = np.arange(1,51, step = 2)
          random_sample = np.array(data[np.random.choice(data.shape[0],
                                                         size = 1000)])
          paramter_selection(params = n_nums, func = mnn,
                             data = random_sample)
```

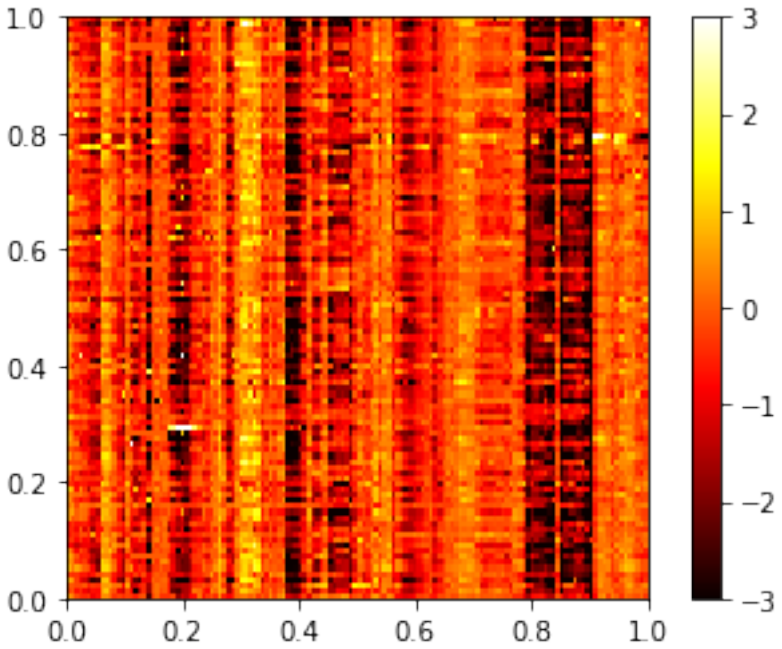


All the parameters for mnn indicate that there are 3 main clusters, I chose 3 nearest neighbors because the gap was the greatest, except of 1 nearest neighbor which yielded bad results. However, all the parameters seem to yield very similar results.

```
In [135]: clusters, centers, stats = spectral(random_sample, 3,3,mnn)
          for i in range(3):
              plt.figure()
              plt.imshow(random_sample[clusters == i],extent=[0, 1, 0, 1],
                         cmap="hot", vmin=-3, vmax=3)
              plt.colorbar()
              plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
3
2
1
0
−1
−2
−3

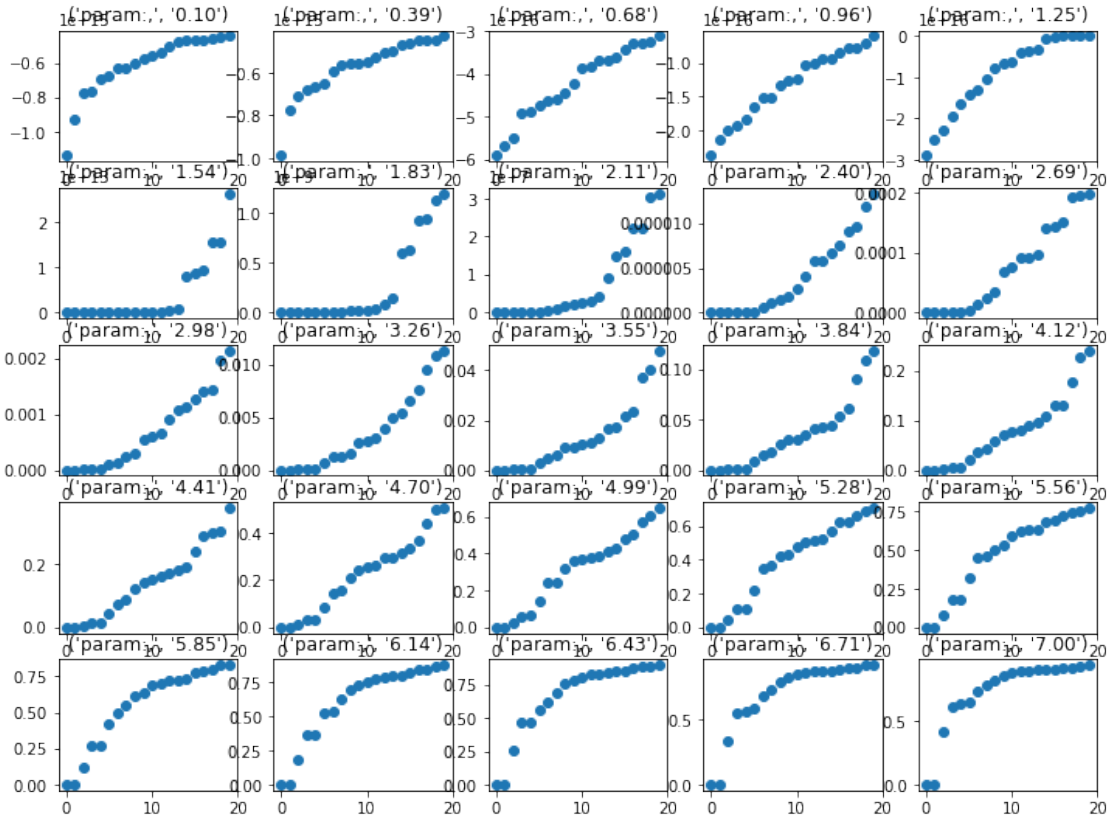

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
3
2
1
0
−1
−2
−3



The first cluster consists of values that are closer to zero, the second of values greater than zero, and the third of values below zero.

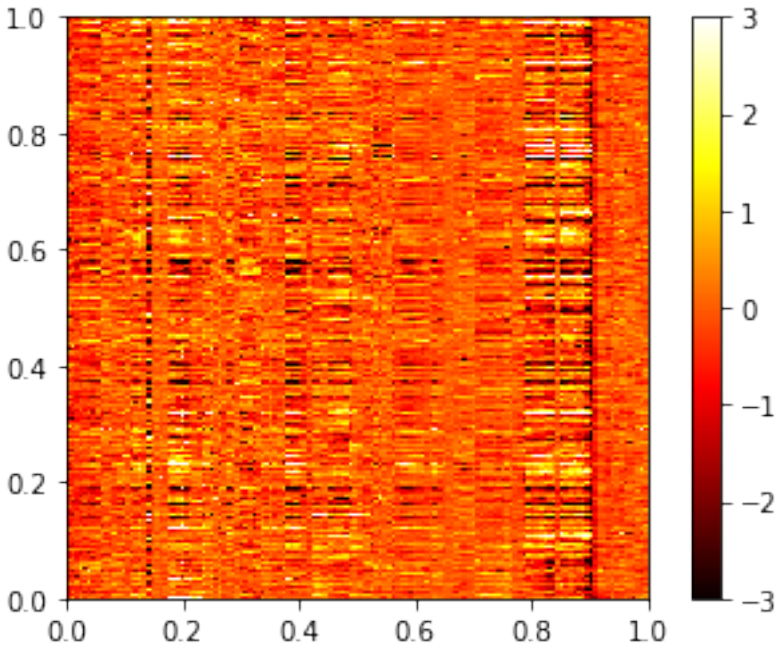
Heat Kernel - Parameter Selection

```
In [138]: sigmas = np.linspace(0.1, 7, 25)
          random_sample = np.array(data[np.random.choice(data.shape[0],
                                                         size = 1000)])
          paramter_selection(params = sigmas, func = heat, data = random_sample)
```

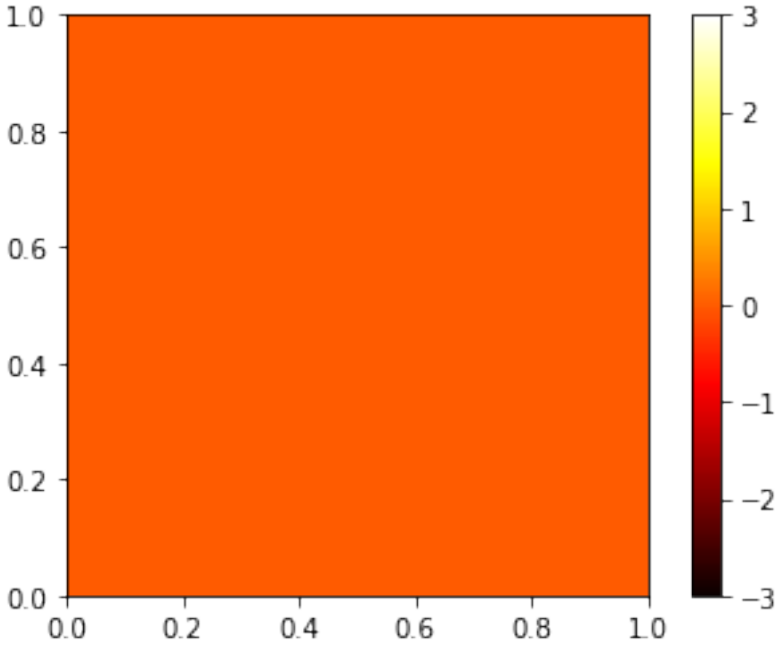


The first obvious gap is for sigma 1.54, with k = 14. I chose to use those parameters. The Second big gap is for k = 2, sigma > 6, but those parameters yield bad results.

```
In [147]: clusters, centers, stats = spectral(random_sample, 14, 1.54)
          for i in range(14):
              print('size of ',i,'th', 'cluster is:',
                    len(clusters[clusters == i]))

              if (len(clusters[clusters == i]) > 1):
                  plt.figure()
                  plt.imshow(random_sample[clusters == i],extent=[0, 1, 0, 1],
                             cmap="hot", vmin=-3, vmax=3)
                  plt.colorbar()
                  plt.show()

size of  0 th cluster is: 894
```




```
size of  1 th cluster is: 1
size of  2 th cluster is: 1
size of  3 th cluster is: 1
size of  4 th cluster is: 1
size of  5 th cluster is: 1
size of  6 th cluster is: 94
```



```
size of  7 th cluster is: 1
size of  8 th cluster is: 1
size of  9 th cluster is: 1
size of  10 th cluster is: 1
size of  11 th cluster is: 1
size of  12 th cluster is: 1
size of  13 th cluster is: 1
```

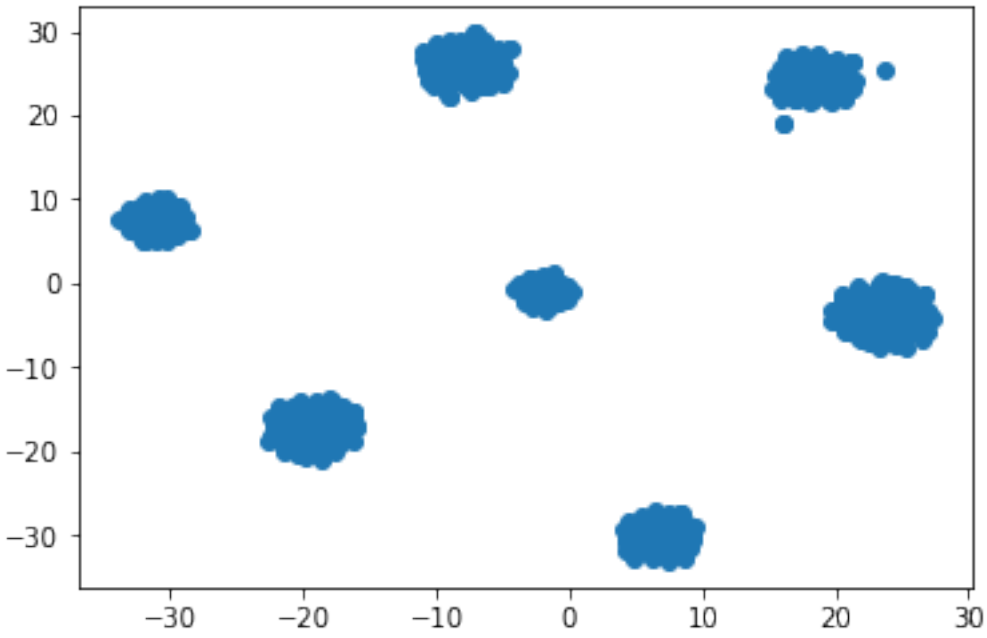
As we see, most of the clusters are outliers of size 1, but the separation for the two big clusters if of vectors that are very close (or equal) to zero and all the other vectors.

4 t-SNE:

Now we will produce new data with 7 clusters.

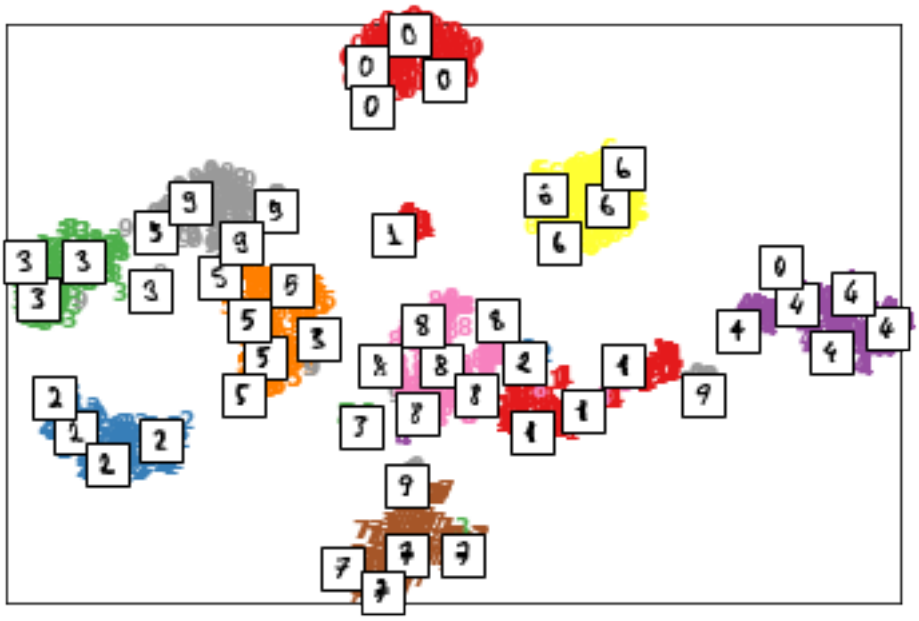
```
In [64]: X = produce_data(7, dim  = 100)
         X_embedded = TSNE(n_components=2).fit_transform(X)
         fig, ax = plt.subplots()
         ax.scatter(X_embedded[:,0], X_embedded[:,1])
```

```
Out[64]: <matplotlib.collections.PathCollection at 0x113c99ba160>
```



t-SNE successfully reduces the dimension to 2, and it has 7 clusters!
On MNIST data-set:

```
In [65]: digits = datasets.load_digits()
         digits_X = digits.data
         y = digits.target
         X_tsne = TSNE(n_components=2, init='pca', random_state=0).fit_transform(digits_X)
         plot_embedding(X_tsne)
```



Amazing! The clustering is much better than the algorithms from ex1, except some outlier groups - the 1s in the middle and the 9s on the right.

```
In [ ]:
```