Answers

November 16, 2017

In [177]:
```
import pickle
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, metrics
import sklearn
from matplotlib import offsetbox

from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
```

```
def plot_embedding_list(data, y, titles):

    fig, ax = plt.subplots(nrows = len(data), ncols = 1)
    fig.set_size_inches(7, 21)
    for i in range(len(titles)):
        ax[i].set_title(titles[i])

    for j in range(len(data)):

        X = data[j]

        x_min, x_max = np.min(X, 0), np.max(X, 0)
        X = (X - x_min) / (x_max - x_min)


        for i in range(X.shape[0]):
            ax[j].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[j].add_artist(imagebox)
        plt.xticks([]), plt.yticks([])

def plot_embedding(X, y, 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(y[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)
    if title is not None:
        plt.title(title)



def plot_faces_with_images(data, images, titles, image_num=25):
    '''
    A plot function for viewing images in their embedded locations. The
    function receives the embedding (X) and the original images (images) and
    plots the images along with the embeddings.

    :param X: Nxd embedding matrix (after dimensionality reduction).
    :param images: NxD original data matrix of images.
```

```
    :param title: The title of the plot.
    :param num_to_plot: Number of images to plot along with the scatter plot.
    :return: the figure object.
    '''


    n, pixels = np.shape(images)
    img_size = int(pixels**0.5)

    fig, ax = plt.subplots(nrows = 1, ncols=len(data))
    fig.set_size_inches(21,6)
    for i in range(len(titles)):
        ax[i].set_title(titles[i])



    # draw random images and plot them in their relevant place:
    for j in range(len(data)):
        X = data[j]
        # get the size of the embedded images for plotting:
        x_size = (max(X[:, 0]) - min(X[:, 0])) * 0.1
        y_size = (max(X[:, 1]) - min(X[:, 1])) * 0.1

        for i in range(image_num):

            img_num = np.random.choice(n)
            x0, y0 = X[img_num, 0] - x_size / 2., X[img_num, 1] - y_size / 2.
            x1, y1 = X[img_num, 0] + x_size / 2., X[img_num, 1] + y_size / 2.
            img = images[img_num, :].reshape(img_size, img_size)
            ax[j].imshow(img, aspect='auto', cmap=plt.cm.gray, zorder=100000,
                  extent=(x0, x1, y0, y1))

            # draw the scatter plot of the embedded data points:
            ax[j].scatter(X[:, 0], X[:, 1], marker='.', alpha=0.7)

    return fig



def MDS(X, d):
    '''
    Given a NxN pairwise distance matrix and the number of desired dimensions,
    return the dimensionally reduced data points matrix after using MDS.

    :param X: NxN distance matrix.
    :param d: the dimension.
    :return: Nxd reduced data point matrix.
    '''
```

```
    X = X**2
    n = np.shape(X)[0]
    H = np.matrix((np.eye(n))-(1/n)*np.ones(np.shape(X)))
    S = (-0.5)*np.dot(H,np.dot(X,H)) #matrix multiplication
    (eigvals, eigvecs) = np.linalg.eigh(S)

    #take the square root of last d eigenvalues
    eigvals_root = np.sqrt(eigvals[(n-d):]).tolist()
    #take the last d eigenvectors
    biggest_eigvecs = eigvecs[:,(len(eigvals)-d):]
    #duplicate the eigvals vector n times
    eigvals_root_matrix = [[eigvals_root],]*n
    eigvals_root_matrix = np.reshape(eigvals_root_matrix, (n,d))

    # element-wise multiplication
    return eigvals, np.array(biggest_eigvecs)*np.array(eigvals_root_matrix)


def knn(X,k):
    N = np.shape(X)[0]

    distances = sklearn.metrics.pairwise.euclidean_distances(X)
    nearest = np.zeros((N,N))
    for i in range(N):
        sorted_indexes = np.argsort(distances[i])
        nearest[i,sorted_indexes[1:k+1]] = 1
    return nearest

def finding_W(X, nearest,k):
    N = np.shape(X)[0]
    W = np.zeros((N,N))
    for i in range(N):
        indexes = nearest[i].astype(bool)
        neighbors = X[indexes]-X[i]
        gram = np.dot(neighbors, neighbors.transpose())
        w = np.dot(np.linalg.pinv(gram), np.ones(k))
        w = w/np.sum(w)
        W[i][indexes] = w
    return W

def finding_Y(W,d):
    N = np.shape(W)[0]
    M = np.dot((np.eye(N)-W).transpose(), np.eye(N)-W)
    (eigvals, eigvecs) = np.linalg.eigh(M)
    Y = eigvecs[:,1:d+1]
    return Y
```

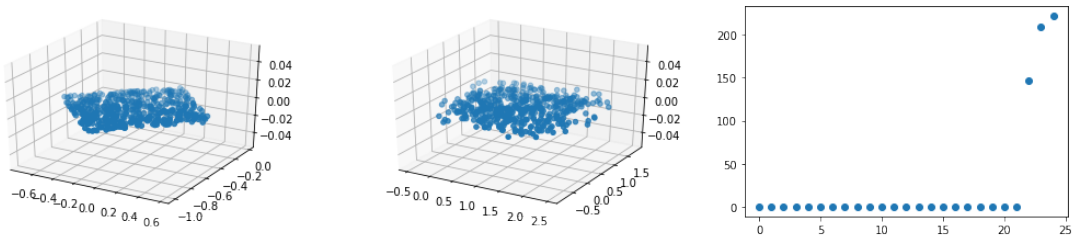
```
def LLE(X, d, k):
    '''
    Given a NxD data matrix, return the dimensionally reduced data matrix after
    using the LLE algorithm.

    :param X: NxD data matrix.
    :param d: the dimension.
    :param k: the number of neighbors for the weight extraction.
    :return: Nxd reduced data matrix.
    '''

    nearest = knn(X, k)
    W = finding_W(X, nearest, k)
    return finding_Y(W, d)


def DiffusionMap(X, d, sigma, t, k=None):
    '''
    Given a NxD data matrix, return the dimensionally reduced data matrix after
    using the Diffusion Map algorithm. The k parameter allows restricting the
    gram matrix to only the k nearest neighbor of each data point.

    :param X: NxD data matrix.
    :param d: the dimension.
    :param sigma: the sigma of the gaussian for the gram matrix transformation.
    :param t: the scale of the diffusion (amount of time steps).
    :param k: the amount of neighbors to take into account when
        calculating the gram matrix.
    :return: Nxd reduced data matrix.
    '''
    N = np.shape(X)[0]
    K = np.zeros((N,N))

    X_dists = (metrics.pairwise.euclidean_distances(X)**2)/(-sigma)
    K = np.exp(X_dists)

    if not (k==None):
        for i in range(N):
            K[i,np.argsort(K[i])[0:N-k]] = 0


    K = K/np.sum(K, axis=1,keepdims=True)

    (eigvals, eigvecs) = np.linalg.eig(K)
    sorted_indexes = np.flip(np.argsort(eigvals),0)
    eigvecs = eigvecs[:,sorted_indexes][:,1:d+1]
    eigvals = eigvals[sorted_indexes][1:d+1]
    return np.multiply(eigvals**t,eigvecs)
```

```
def plot_swiss_roll(data, titles, color):
    fig = plt.figure(figsize=(18,5))
    for i in range(len(data)):
        X = data[i]
        ax = fig.add_subplot(1,len(data),i+1, projection='3d')
        ax.set_title(titles[i])
        ax.scatter(X[:, 0], X[:, 1], c=color, cmap=plt.cm.Spectral)
    return fig


if __name__ == '__main__':

    # TODO: YOUR CODE HERE

    pass
```

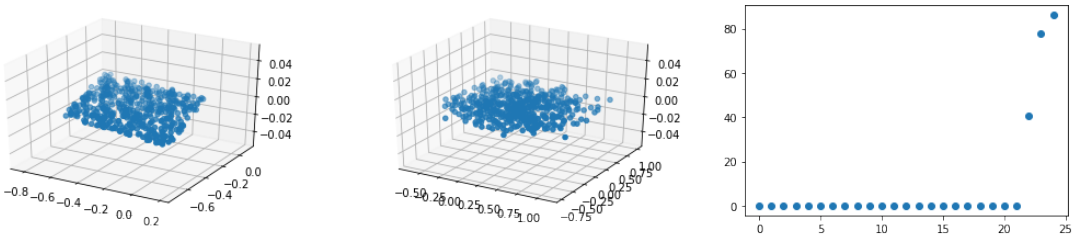
1 Scree Plot

In [3]:
```
def screeplot(size, noise_coeff):
    '''
    Plots 3 figures -
                the original 2d data that is embedded in 3d
                the data with added gaussian noise,
                multiplied by given coefficient
                the eigenvalues returned by MDS function
    '''
    Q,R = np.linalg.qr(np.random.normal(size=(3,2)))
    X = np.dot(Q,np.random.rand(2,size)).transpose()
    noise = np.random.rand(size,3)
    noisy_X = X + noise_coeff*noise

    fig = plt.figure(figsize = (18,12))
    ax = fig.add_subplot(331, projection='3d')
    ax.scatter(X[:,0], X[:,1])

    ax2 = fig.add_subplot(332, projection='3d')
    ax2.scatter(noisy_X[:,0], noisy_X[:,1])

    eigvals_MDS = MDS(sklearn.metrics.pairwise.euclidean_distances(noisy_X), 2)[0]

    ax3 = fig.add_subplot(333)
    ax3.plot(eigvals_MDS[int(size - 0.05*size):], 'o')
```

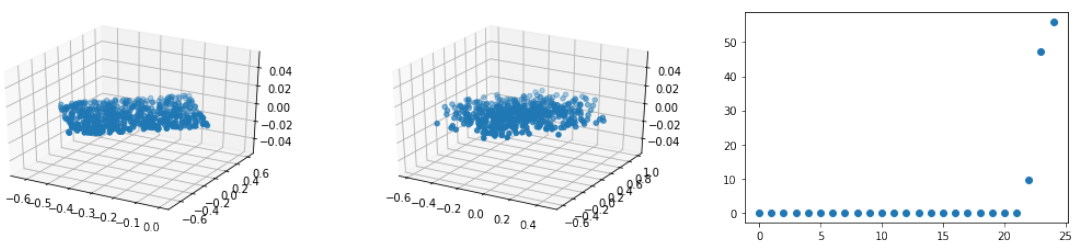
In [4]:
```
screeplot(size = 500, noise_coeff = 2) # The gaussian noise multiplied by 2
```



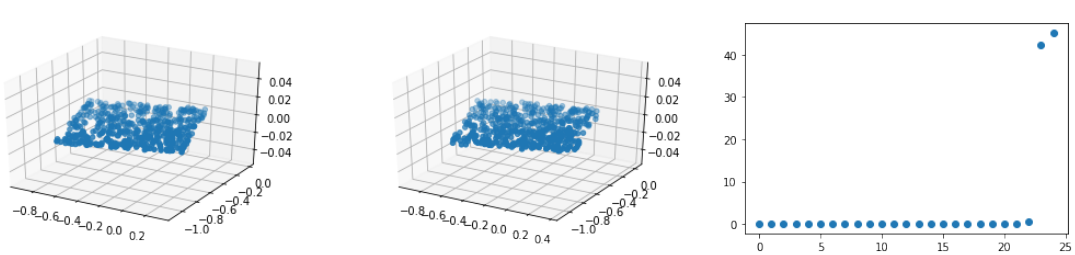
In [5]: `screeplot(size = 500, noise_coeff = 1)` *# The gaussian noise multiplied by 1*



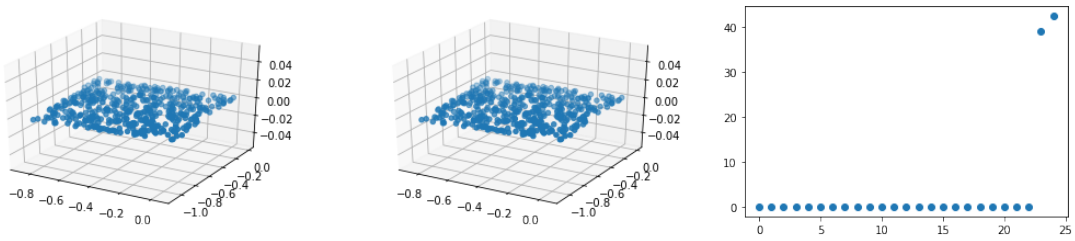
In [7]: `screeplot(size = 500, noise_coeff = 0.5)` *# The gaussian noise multiplied by 0.5*



In [8]: `screeplot(size = 500, noise_coeff = 0.1)` *# The gaussian noise multiplied by 0.1*



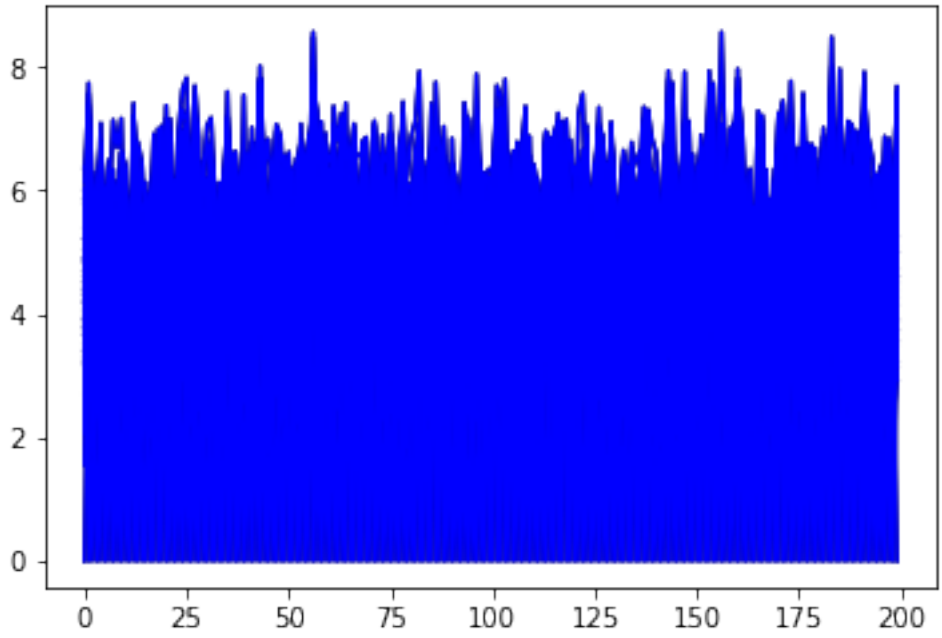
```
In [9]: screeplot(size = 500, noise_coeff = 0) # The gaussian noise multiplied by 0 (the origi
```



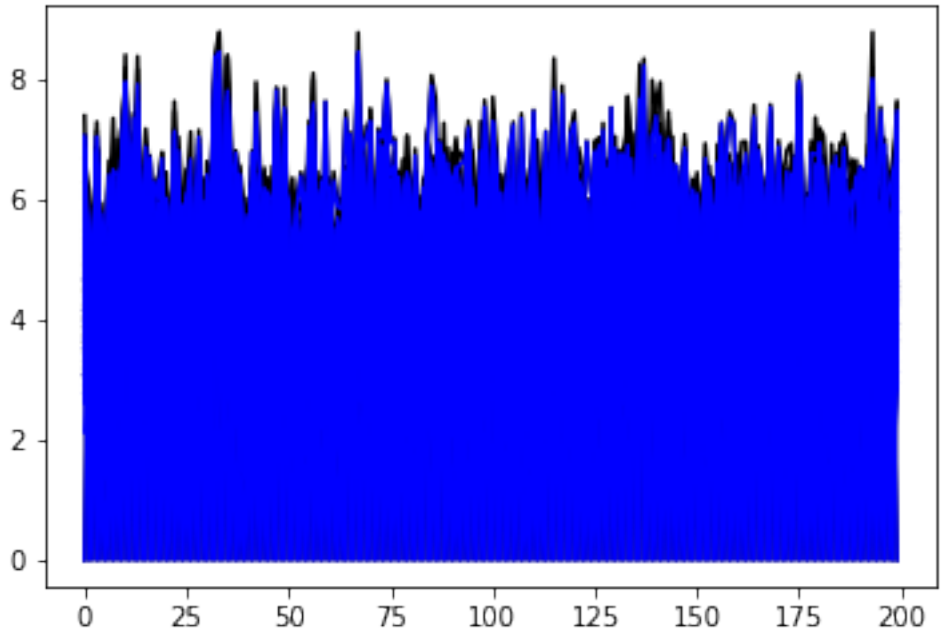
We can see that as we increase the noise, the third eigenvalue returned by the MDS algorithm gets closer to the biggest two. That happens because the variance of the vectors on the third axis (orthogonal to the span of the original data) increases.

2 Lossy Compression of Distances Information

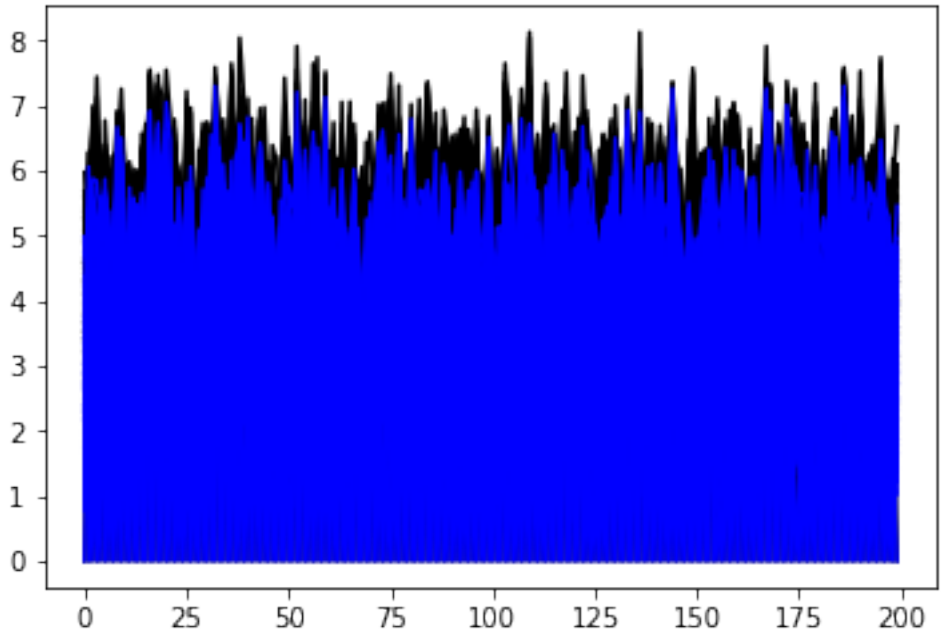
```
In [12]: def lossy_compression(D, d):
             """
             Plots distances of original datapoints and the distances of the
             datapoints after MDS on the same figure.
             """
             N = 200
             X = np.random.normal(size=(N,D))
             dists = sklearn.metrics.pairwise.euclidean_distances(X)
             mds_dists = sklearn.metrics.pairwise.euclidean_distances(MDS(dists, d)[1])
             fig, ax = plt.subplots()
             ax.plot(dists, color = 'black')
             ax.plot(mds_dists, color = 'blue')

In [13]: lossy_compression(10, 10) # MDS doen't reduce dimension
```



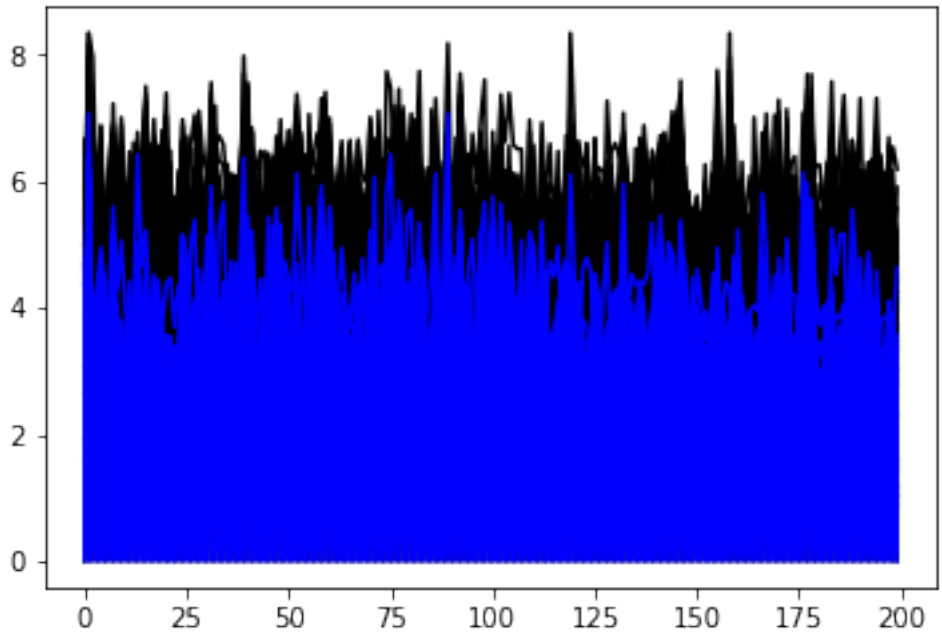
In [14]: lossy_compression(10, 8) # MDS reduces dimension to 8



In [15]: lossy_compression(10, 5) # MDS reduces dimension to 5



In [17]: lossy_compression(10, 2) # MDS reduces dimension to 2



3 MNIST

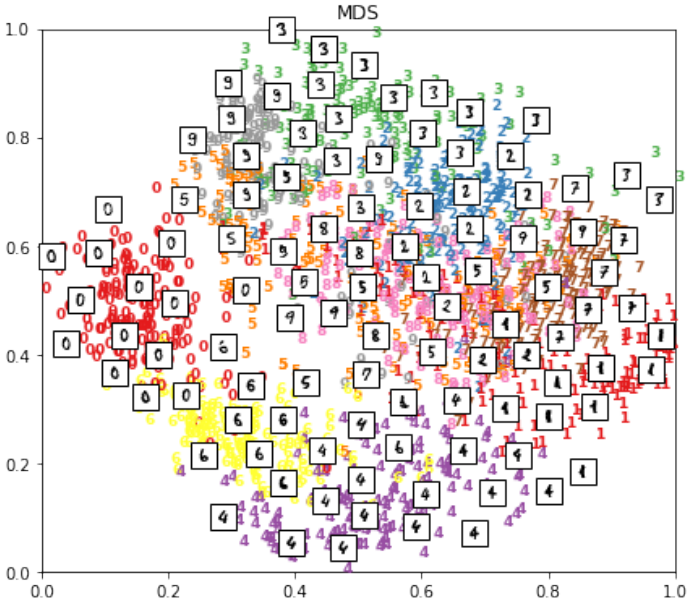
In [43]:
```
# load the MNIST data:
digits = datasets.load_digits()
data = (digits.data)/255.
#data = digits.data / 255.
labels = digits.target
images = digits.images[digits.target ==1 ]
```

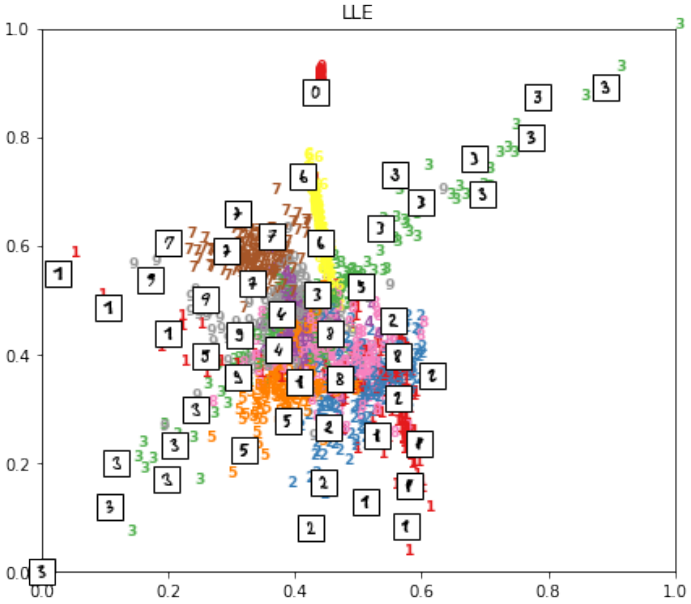
In [44]:
```
DM_digits = DiffusionMap(data, 2, 1000, 50)
```

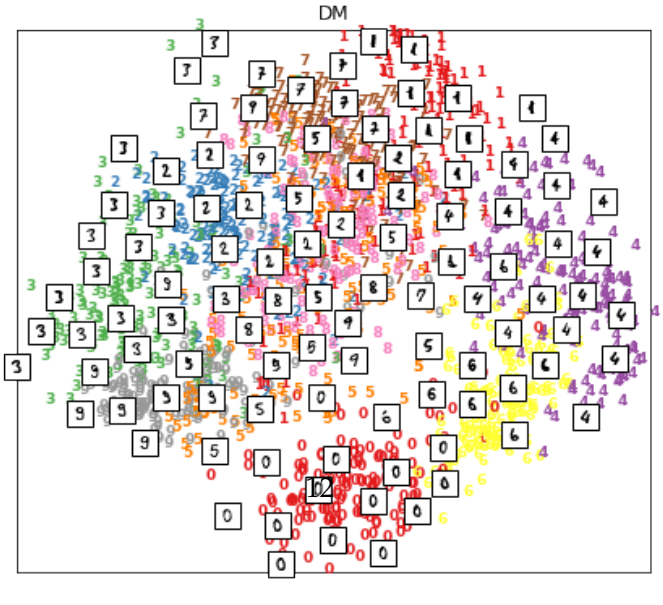
In [45]:
```
MDS_digits = MDS(sklearn.metrics.pairwise.euclidean_distances(data), 2)[1]
```

In [49]:
```
LLE_digits = LLE(data, k = 15, d = 2)
```

In [50]:
```
plot_embedding_list([MDS_digits, LLE_digits, np.real(DM_digits)],
                            labels, ['MDS', 'LLE', 'DM'])
```


MDS
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

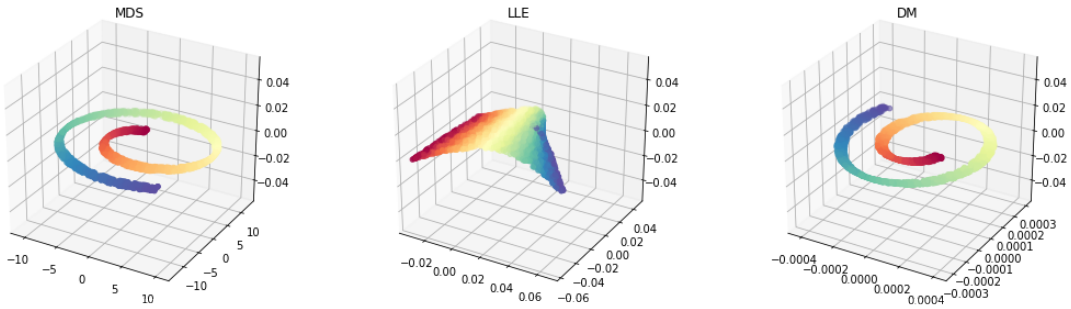

LLE
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


DM

As we can see, none of the algorithms couldn't separate the data well, there are still many overlaps, especcially with 8 and 5 which are all over the place. MDS and Diffusion Maps algorithms separated the data better than LLE, with DM being slightly superior than MDS. LLE succeeds with assigning specific areas to the digits, some of which are really clustered together, like 0 and 6 for example. However, there is no margin between the digits. As the k parameter goes higher, the digits are more mashed together and scattered over the figure, and there is less distinct separation.

4 SwissRoll

```
In [51]: X, color = datasets.samples_generator.make_swiss_roll(n_samples=2000)

In [52]: MDS_swiss = MDS(sklearn.metrics.pairwise.euclidean_distances(X),2)[1]

In [79]: LLE_swiss = LLE(X, k = 36, d = 2)

In [113]: DM_swiss = np.real(DiffusionMap(X, d = 2, sigma = 100, t = 1))

In [112]: f =plot_swiss_roll([MDS_swiss, LLE_swiss, DM_swiss],
                             ['MDS', 'LLE', 'DM'], color)
```

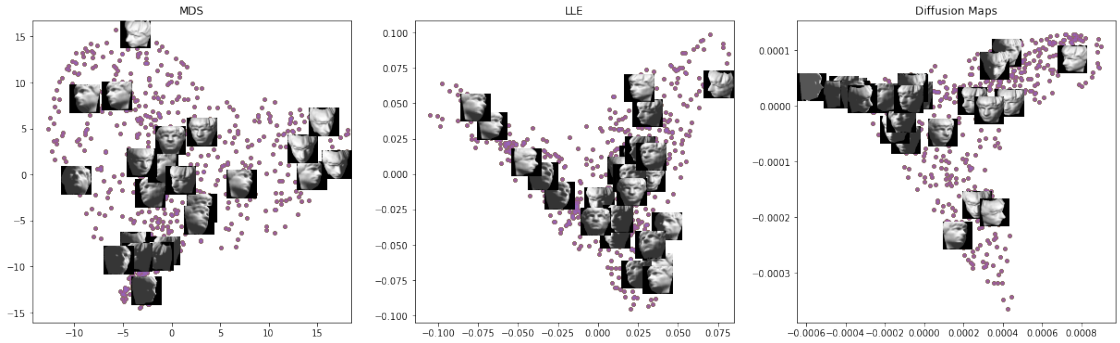


The LLE is the best algorithm for this dataset. The MDS and Diffusion maps algorithms show similar results, I couldn't bring DM to show better results. For larger sigma the results are similar, for smaller sigma the unraveling is worse - it resembles the Nike logo. This is very surprising to me because I would expect that for sigma that is not too large the unravelling would be similar to the LLE, because the Markov chain matrix would assign very small probability to transition to datapoints that are not nearest neighbors. MDS demonstrates such poor results because it flattens the dimension where the variance of the data is the smallest, so it can't unravel a manifold, only flatten it with as little information loss as possible.

5 Faces

```
In [114]: with open('faces.pickle', 'rb') as f:
              X = pickle.load(f)
```

```
In [156]: LLE_faces = LLE(X, d=2, k=18)

In [116]: MDS_faces = MDS(sklearn.metrics.pairwise.euclidean_distances(X), 2)[1]

In [165]: DM_faces = np.real(DiffusionMap(X, d=2, sigma=150, t=10))

In [166]: f=plot_faces_with_images([MDS_faces, LLE_faces, DM_faces], X,
                                   ['MDS', 'LLE', 'Diffusion Maps'], image_num=25)
```

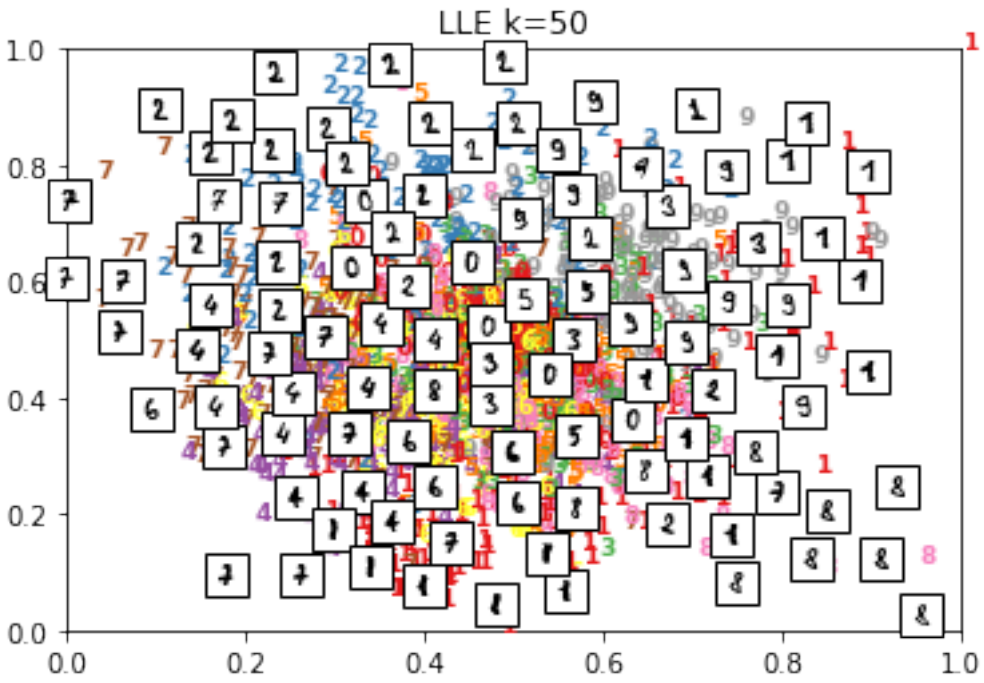


The reducted dimension is 2 - one appears to be the angle of the head and the other the brightness. If the reduction is to bigger dimension, we get similar results in each algorithm. The MDS succeds in differentienting the data by brightness - the darker figures are on the bottom, however the division by head angle is not so good - the heads on the left seem to look left, and the heads on the right seem to look right, but the division is not clear. In the results of LLE the orientation is very clear, all the heads seem to look at the middle of the figure. Moreover, the darker heads are closer to the x axis. In the results of the DM algorithm there is clear division between the bright and the dark heads, and the orientation within those clusters appears to be counter-clockwise.

6 Parameter Tweaking

When choosing k parameter that is too lorge, the data gets meshed. For example, when choosing large k for the MNIST dataset, the differentiation between the numbers is not as clear as with the k that is shown above.

```
In [167]: LLE_digits_large_k = LLE(data, k = 50, d = 2)

In [169]: plot_embedding(LLE_digits_large_k, labels, 'LLE k=50')
```



However, when we choose k that is too small the data tends to collapse into itself - all the similar datapoints merge into a line, in the fugure below according to orientation.

```
In [175]: LLE_digits_small_k = LLE(data, k = 10, d = 2)
```

```
In [174]: plot_embedding(LLE_digits_small_k, labels, 'LLE k=10')
```