Locally Linear Embedding (40 P)

In this assignment we will look at locally linear embedding and experiment with it on artificially generated datasets. The effects of neighbourhood size and noise on result quality will be analyzed.

Information about the algorithm, publications and demos can be found at
http://www.cs.nyu.edu/~roweis/lle/ (http://www.cs.nyu.edu/~roweis/lle/)

A guide for plotting can be found here: http://nbviewer.ipython.org/github/jrjohansson/scientific-python-lectures/blob/master/Lecture-4-Matplotlib.ipynb
(http://nbviewer.ipython.org/github/jrjohansson/scientific-python-lectures/blob/master/Lecture-4-Matplotlib.ipynb)

We first start by importing some basic python libraries for numerical computation and plotting.

```
In [3]:  import numpy as np
         import matplotlib
         from matplotlib import pyplot as plt
         from mpl_toolkits.mplot3d.axes3d import Axes3D
         import scipy.spatial, scipy.linalg
```

The file `utils.py` contains several methods to generate pseudo-random three-dimensional datasets. They all have a low-dimensional manifold structure. The following code plots each dataset with default generation parameters (`N=1000` examples, and Gaussian noise of scale `0.25`).
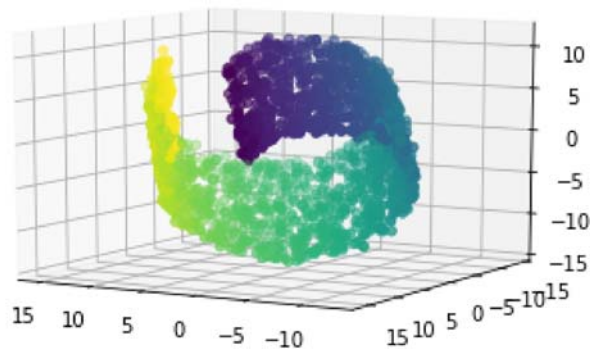
```
In [4]:  import utils
         %matplotlib inline

         for dataset in [utils.spiral,utils.roll,utils.wave,utils.fold]:
             print(dataset.__name__)
             data,color = dataset(N=2000,noise=0.25)
             plt.figure()
             ax = plt.gca(projection='3d')
             ax.view_init(elev=10., azim=120)
             ax.scatter(data[:,0],data[:,1],data[:,2],c=np.ravel(color))
             plt.show()
```
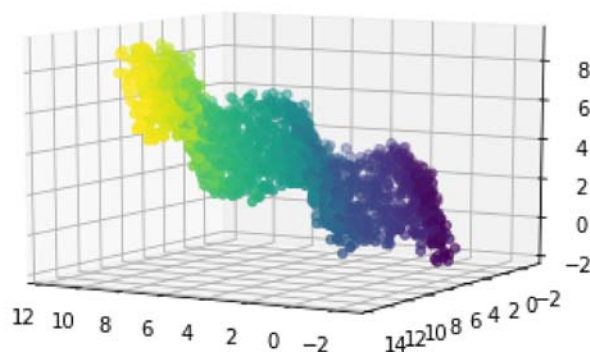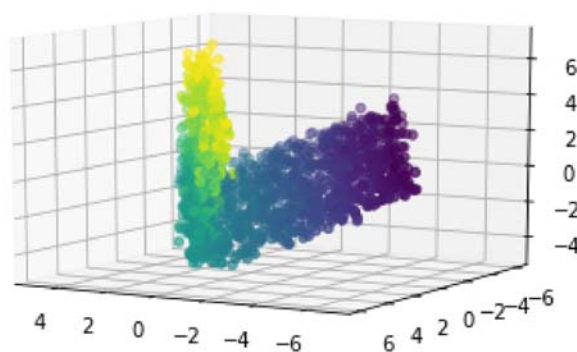
spiral
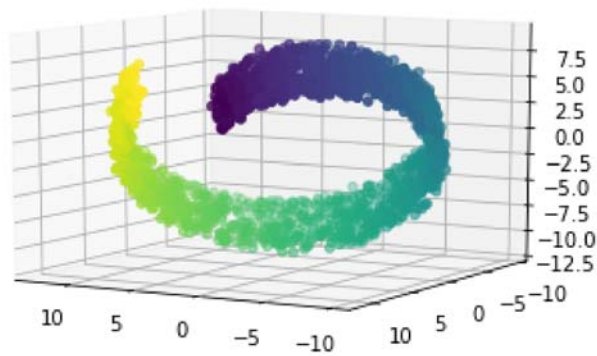
roll

wave

```
fold
```


Implementing LLE (20 P)

Implement a function `LLE(data,k)` . The function takes as input the high-dimensinonal data the number of neighbors `k` used for reconstruction in the LLE algorithm. It returns the resulting 2D embedding (a two-dimensional array of size `Nx2`). A backbone of the implementation is given below. The implementation of LLE is described in the paper "An Introduction to LLE" linked from ISIS.

*Reminder: During computation, you need to solve the problem Cw=**1**, where **1** is a column vector (1,1,...,1). In case k>d i.e. the size of the neighbourhood is larger than the number of dimensions of the space we're mapping to, it is necessary to regularize the matrix C. You can do this by adding positive terms on the diagonal. A good starting point is 0.05.*

```python
In [126]:  def LLE(data,k):

               N = len(data)
               W = np.zeros([N,N]) # matrix for storing reconstruction weights
               M = np.zeros([N,N]) # matrix M of which eigenvectors are computed
               E = np.zeros([N,2]) # eigenvectors of M forming the embedding

               D = np.shape(data)[1]
               d = 2

               k = np.minimum(N - 1, k) # there cannot be more neighbors than data points -

               # indices of k nearest neighbors sorted by distance
               KNNindex = np.argsort(scipy.spatial.distance_matrix(data, data), axis = 1)[:,

               # Iterate over all data points to find their associated reconstruction weight.
               for i in range(N):

                   x = np.reshape(data[i],(1,D))

                   # 1. find nearest neighbors of x
                   KNN = data[KNNindex[i]]

                   # 2. compute local covariance matrix
                   C = np.dot(x - KNN, x.T - KNN.T)

                   # regularization if k>d
                   if k > d: # (this is the case)
                       C = C + np.eye(k)*0.05

                   # inverse
                   Cinv = np.linalg.inv(C)

                   # 3. compute reconstruction weights and store them in the row of the matr
                   W[i, KNNindex[i]] = (1 / np.sum(Cinv)) * np.sum(Cinv, axis = 1)


               # 4. Compute the matrix M from W and compute the desired eigenvectors E of M
               I = np.eye(N)
               M = np.dot((I-W).T,I-W)

               Evals, Evecs = np.linalg.eig(M)

               # get indices d+1 smallest eigenvalues and return d eigenvectors (discarding
               E =  Evecs[:,np.argsort(Evals)[1:d + 1]]

               # return E
               return E
```

Test your implementation by running the code below. It plots the original data (projected on the first two dimensions), and next to it, the two-dimensional embedding. A correct implementation produces a two-dimensional dataset where the manifold is unfolded, and where nearby points in the embedding are also neighbors in the manifold, in particular, neighboring points in the plot should have similar color.

```
In [127]: import utils
          %matplotlib inline
          data,color = utils.roll()
          embedding = LLE(data,k=20)
          fig = plt.figure(figsize=(10,5))

          ax = fig.add_subplot(1,2,1,projection='3d')
          ax.view_init(elev=10., azim=120)
          ax.scatter(data[:,0],data[:,1],data[:,2],c=np.ravel(color))
          ax.set_title('data')

          ax = fig.add_subplot(1,2,2)
          ax.scatter(embedding[:,0],embedding[:,1],c=np.ravel(color))
          ax.set_title('LLE')
```

Out[127]: Text(0.5,1,'LLE')
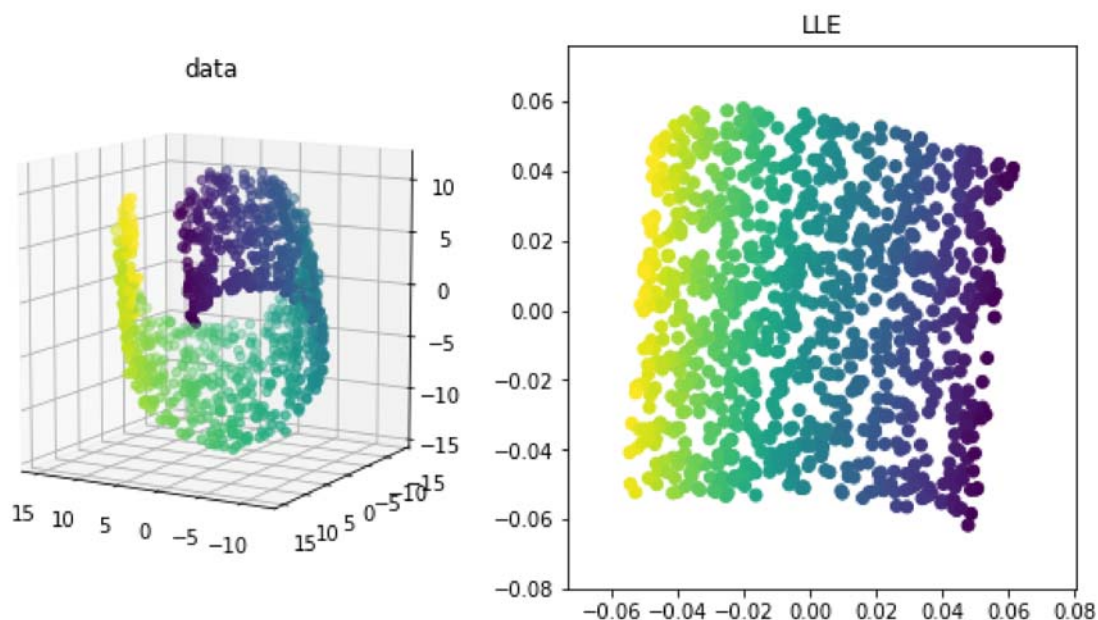
Experiments (20 P)

The function `compare(embed,dataset)` takes as input an embedding function and a dataset and plots the resulting embeddings for various choices of the parameter `k`, in particular, `k=5,20,80`.

```python
In [130]: def compare(embed, dataset):
              cols = 4

              fig = plt.figure(figsize=(3*cols, 3))

              # Plot the data
              data, color = dataset()
              ax = fig.add_subplot(1, cols, 1,projection='3d')
              ax.view_init(elev=10., azim=120)
              ax.scatter(data[:,0],data[:,1],data[:,2],c=np.ravel(color))

              ax.set_title('data')
              ax.set_xticks([], [])
              ax.set_yticks([], [])
              ax.set_zticks([], [])

              # Plot embeddings with various parameters K
              for i, k in enumerate([5,20,80]):
                  ax = fig.add_subplot(1, cols, 2+i)

                  z = embed(data,k=k)

                  ax.scatter(z[:,0], z[:,1],c=np.ravel(color))
                  ax.set_title('LLE, k=%d'%k)
                  ax.set_xticks([], [])
                  ax.set_yticks([], [])
              plt.tight_layout()
              plt.show()
```

Datasets and optimal parameter k

The code below tests the LLE embedding algorithm on each dataset: spiral, roll, wave, and fold.
Explain what is a good parameter k of the LLE algorithm, and how this parameter relates to the
various properties of the dataset.

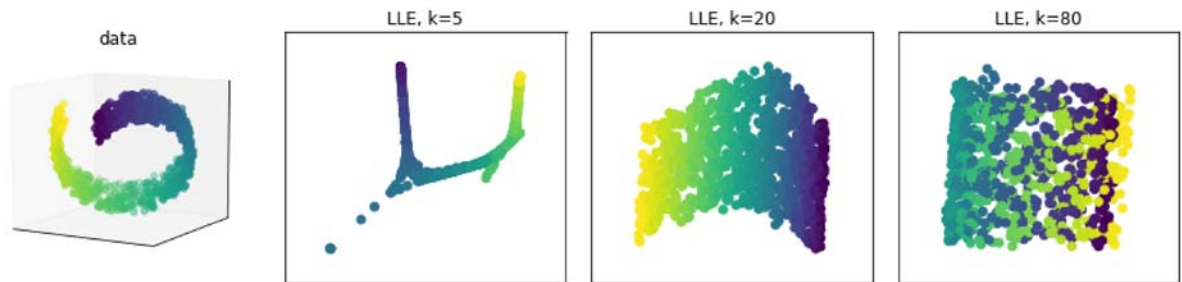**[TODO: WRITE YOUR ANSWER HERE]**

Choosing a very small k-value can lead to discardong of global structure and coninciding of
embedded points, since there is only a local estimation.

Choosing k too large however discards too much local structure and behaves like PCA, where we
observe the embedding to be just a d dimensional shadow of the high diemensional input

A 'good' parameter for k is not easy to find in general. We have to introduce measures on the
resulting embeddings to quantify a 'good' k-value range (e.g. Spearman's rho)

```
In [131]:   for dataset in [utils.spiral,utils.roll,utils.wave,utils.fold]:
                print(dataset.__name__)
                compare(LLE,dataset)
```
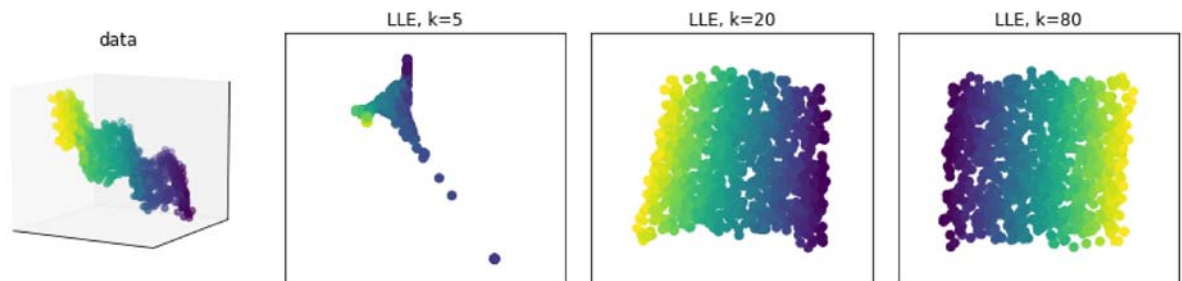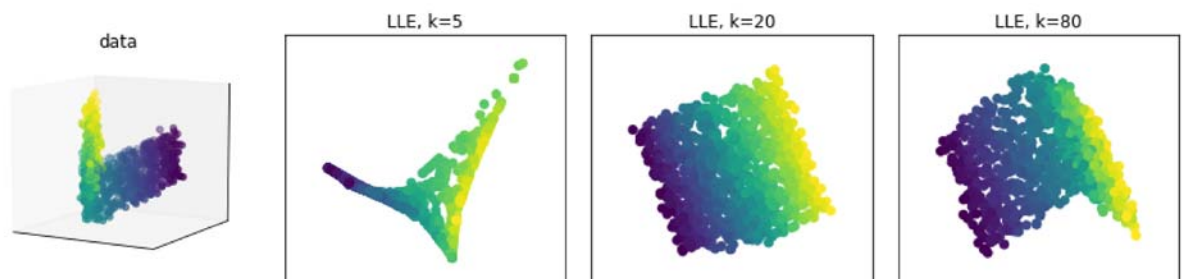
spiral


roll
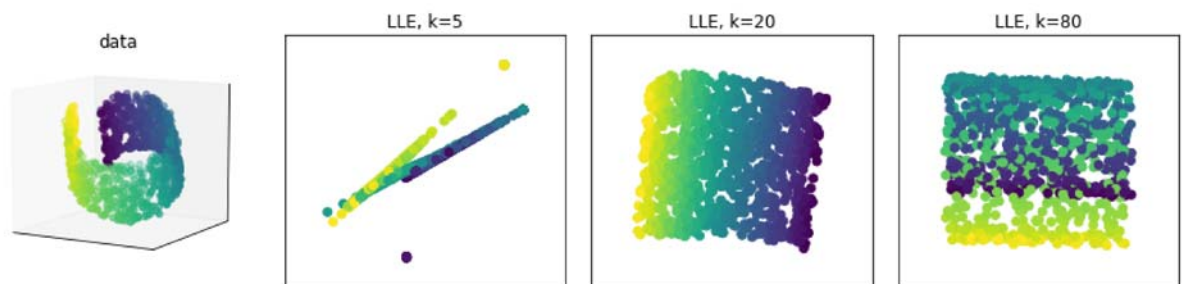
wave

fold

LLE and the effect of noise

The code below tests LLE on the spiral dataset with three different levels of noise. **Describe** how the noise affects the embedding.
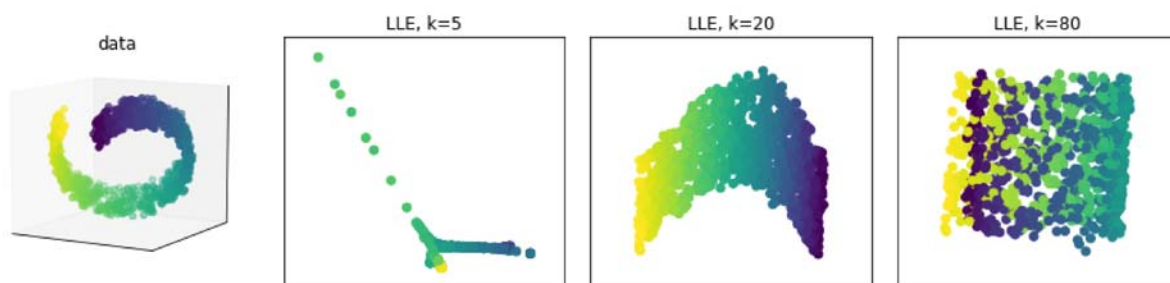
[TODO: WRITE YOUR ANSWER HERE]

Introducing heavy noise on the data can 'trick' the algorithm into misinterpreting the topology of the data. This is extremnely obvious in the last row, where the resulting embedding forms a circle rather rather than a correctly unraveled rectangle. The noise basically pulls the two sides of the rectangle together.

```
In [132]: for noise in [0.1,0.3,1.0]:
              print('noise=%.3f'%noise)
              dataset = lambda: utils.spiral(noise=noise)
              compare(LLE,dataset)
```
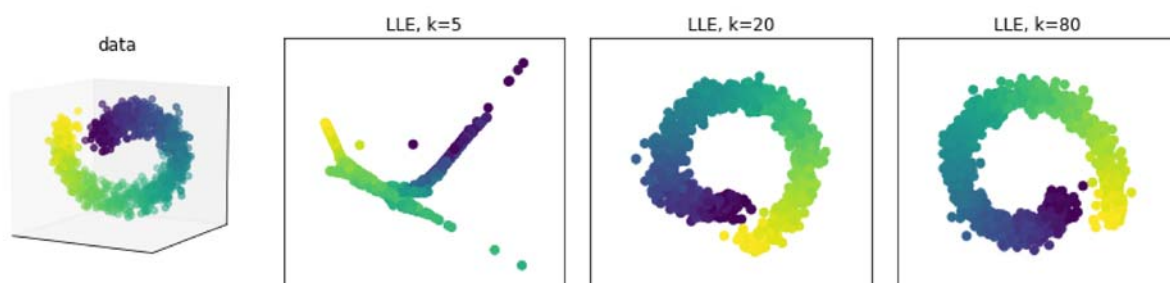
noise=0.100
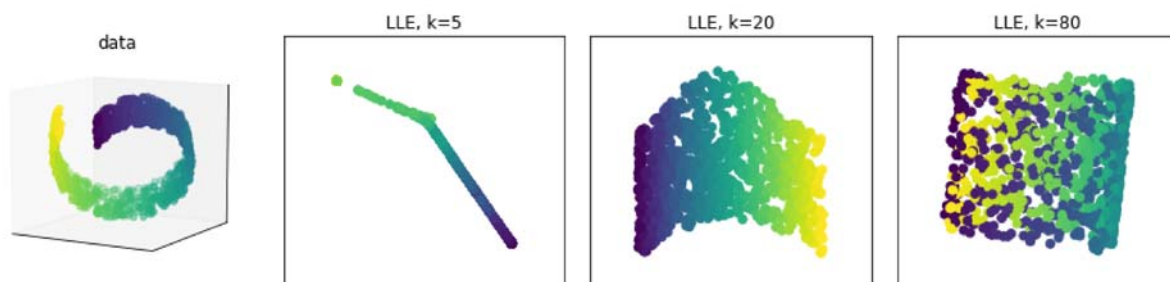
noise=0.300

noise=1.000

Discussion

- When applying LLE to a 200-dimensional dataset which can't be visualized, how would you assess whether the found embedding is good? Discuss how you would do it or argue why it can't be done.

  [TODO: WRITE YOUR ANSWER HERE]

For very high dimensional input one has to introduce additional measures on the embeddings, to get a better idea of their quality. This can be done in various ways (see above in the answer for selecting k). The basic idea is that if the relative distances between data points stays roughly the same, we have found a good embedding for the data.

- Could utilizing this technique in conjuction with a classifier improve its performance? Which classifiers, if any, would benefit the most?

  [TODO: WRITE YOUR ANSWER HERE]

  If the underlying data allows for a good linear mapping (that is it can be unraveled to a linear structure while keeping most of its features) linear classifiers like LDA probably benefit the most. In general LLE seems to perform pretty well across the board and would likely boost other classifiers such as SVM's as well.

In [ ]: