sheet01-programming

April 22, 2021

0.1 Implementing Locally Linear Embedding (25 P)

In this programming homework we will implement locally linear embedding (LLE) and experiment with it on the swiss roll dataset. In particular, the effects of neighbourhood size and noise on the quality of the embedding will be analyzed.

```
[1]: import numpy as np
     import matplotlib
     %matplotlib inline
     from matplotlib import pyplot as plt
     from mpl_toolkits.mplot3d.axes3d import Axes3D
     import sklearn,sklearn.datasets
```

The following code plots the swiss roll dataset (a commonly used dataset to test LLE) with N=1000 data points and a noise parameter of 0.25.

```
[2]: X,T = sklearn.datasets.make_swiss_roll(n_samples=1000, noise=0.25)
     plt.figure(figsize=(5,5))
     ax = plt.gca(projection='3d')
     ax.view_init(elev=10., azim=105)
     ax.scatter(X[:,0],X[:,1],X[:,2],c=T)
     plt.show()
```

```
<ipython-input-2-2ff129158a82>:3: MatplotlibDeprecationWarning: Calling gca()
with keyword arguments was deprecated in Matplotlib 3.4. Starting two minor
releases later, gca() will take no keyword arguments. The gca() function should
only be used to get the current axes, or if no axes exist, create new axes with
default keyword arguments. To create a new axes with non-default arguments, use
plt.axes() or plt.subplot().
  ax = plt.gca(projection='3d')
```
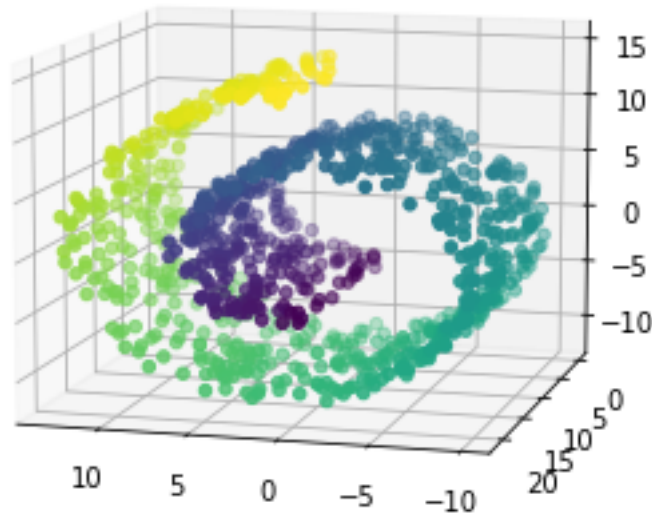
Although the dataset is in three dimensions, the points follow a two-dimensional low-dimensional structure. The goal of embedding algorithms is to extract this underlying structure, in this case, unrolling the swiss roll into a two-dimensional Euclidean space.

In the following, we consider a simple implementation of LLE. You are required to complete the code by writing the portion where the optimal reconstruction weights are extracted. *(Hint: During computation, you need to solve an equation of the type 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 input space, 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
def LLE(X,k):
    N = len(X)
    W = np.zeros([N,N])

    for i in range(N):
        # -------------------------------------
        # My code
        # calculate euclidean distance
        DisMat = X[i]-X
        DisMat = np.linalg.norm(DisMat, axis=1)
        # find the fist k smallest distances' index
        ind = np.argsort(DisMat)[1:k+1]
```

2

```
            # create matrix Z consisting of all neighbours of Xi [d] and subtract␣
    ↪Xi from Z
        Z = X[i] - X[ind,:]
        # compute the local covariance C=Z @ Z.T
        C = Z @ Z.T
        # in case k>d
        if k > len(X[0]):
            C[np.diag_indices_from(C)] += 0.05
        # solve linear system C*w = 1 for w
        w = np.linalg.solve(C,np.ones((k,1)) )
        # normalize w
        w = w / sum(w)
        w = w.reshape(k,)
        # ----------------------------------------
        W[i,ind] = w

    M = np.identity(N) - W - W.T + np.dot(W.T,W)
    E = np.linalg.svd(M)[0][:,-3:-1]

    return E
```

You can now test your implementation on the swiss roll dataset and vary the noise in the data and the parameter `k` of the LLE algorithm. Results are shown below:

```
[4]: f = plt.figure(figsize=(12,3))
for t,(k,noise) in enumerate([(2,0.1),(10,0.1),(25,0.1),(10,1)]):
    X,T = sklearn.datasets.make_swiss_roll(n_samples=1000, noise=noise)
    embedding = LLE(X,k=k)
    ax = f.add_subplot(1,4,t+1)
    ax.set_title('k=%d, noise=%.1f'%(k,noise))
    ax.set_xticks([])
    ax.set_yticks([])
    ax.scatter(embedding[:,0],embedding[:,1],c=T)
```
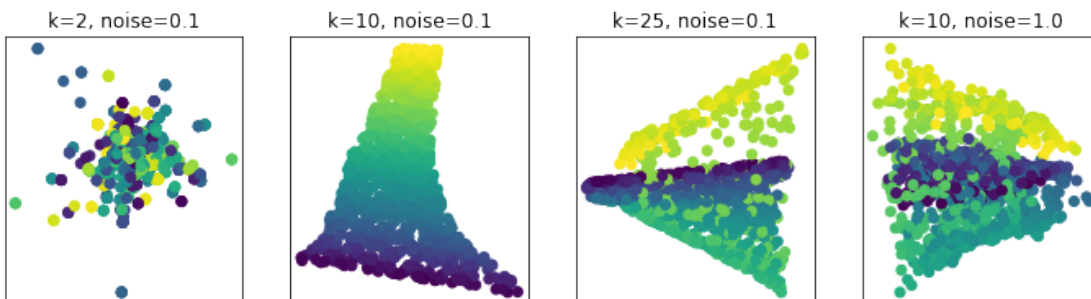

It can be observed that the parameter `k` must be carefully tuned to have sufficiently many neighbors for stability but also not too many. We can further observe that LLE works well as long as the

noise in the data remains low enough.