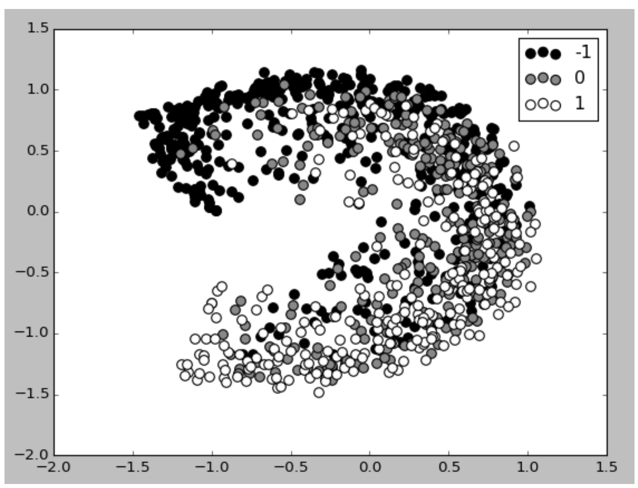
MDS

python では、sklean パッケージに manifold というライブラリがあり、この中に主な次元削減法が全て実装されている。例えば MDS なら、manifold.MDS()関数を使えば良い。

python コード

```
...
# MDS オブジェクトを生成（2 次元に落とし込む）
mds = manifold.MDS(n_components=2, max_iter=100, n_init=1)

# manifold 空間での 2 次元座標を計算
pos = mds.fit_transform(data)

# 表示
plt.scatter(pos[:,0], pos[:,1])
plt.show()
```



数学的な話

数学的な話は、以下のページが分かりやすい。

http://d.hatena.ne.jp/koh_ta/20110514/1305348816

要するに、N個のデータを指定された次元の空間上に配置する時に、データi、j間の距離が、定義した距離行列の値d_{ij}に最も近くなるように（最小二乗誤差）、座標x_iを決定するのだ。

つまり、

$$d_{ij}^2 = \|x_i - x_j\|^2 = \sum_k (x_{ik} - x_{jk})^2 = \sum_k (x_{ik}^2 + x_{jk}^2 - 2x_{ik}x_{jk}) \tag{1}$$

さて、データ行列Xの各行が、各データx_iに対応しているとする。そして、行列$B = XX^T$を定義した時に、もし行列Bが分かれば、行列Xを求めることが出来る。

というわけで、まずは行列Bを解こう。行列Bのi行j列成分b_{ij}は、

$$b_{ij} = \sum_k x_{ik}x_{jk} \tag{2}$$

ってことは、

$$\begin{cases} b_{ii} = \sum_k x_{ik}^2 \\ b_{jj} = \sum_k x_{jk}^2 \end{cases}$$

こっらを式(1)に代入すると、

$$d_{ij}^2 = b_{ii} + b_{jj} - 2b_{ij} \tag{3}$$

この式がとっても重要だ！この後の流れで、行列Bの対角成分が分かる。つまり、b_{ii}、b_{jj}が分かる。そしたら、この式で、任意の成分b_{ij}が求められるのだ。

では、つづきを行くよ。データ行列Xは、例によって、各列ごとについて、平均値を 0 とするように normalize しているとすると、

$$\sum_{i=1}^N x_{ik} = 0 \tag{4}$$

式(2)をiで sum してやって、この式(4)を代入すると、

$$\sum_{i=1}^N b_{ij} = \sum_{i=1}^N \sum_k x_{ik}x_{jk} = \sum_k x_{jk} \sum_{i=1}^N x_{ik} = 0$$

同様に、

$$\sum_{j=1}^N b_{ij} = 0$$

続いて、式(3)をiで sum してやると、

$$\sum_{i=1}^N d_{ij}^2 = \sum_{i=1}^N (b_{ii} + b_{jj} - 2b_{ij}) = \mathrm{Tr}(B) + Nb_{jj} \tag{5}$$

同様に、式(3)をjで sum してやると、

$$\sum_{j=1}^{N} d_{ij}^2 = \sum_{j=1}^{N} \left(b_{ii} + b_{jj} - 2b_{ij}\right) = Nb_{ii} + \mathrm{Tr}(B) \tag{6}$$

さらに、式(3)をi、jで sum してやると、

$$\sum_{i=1}^{N}\sum_{j=1}^{N} d_{ij}^2 = \sum_{i=1}^{N}\sum_{j=1}^{N} \left(b_{ii} + b_{jj} - 2b_{ij}\right) = 2N\mathrm{Tr}(B) \tag{7}$$

式(7)より、

$$\mathrm{Tr}(B) = \frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{N} d_{ij}^2$$

なので、行列Bのトレース、つまり、対角成分の和が求まる。これを、式(5)、または、式(6)に代入してやれば（どっちの式でも良い）、

$$b_{ii} = \frac{1}{N}\left(\sum_{j=1}^{N} d_{ij}^2 - \mathrm{Tr}(B)\right)$$

より、行列Bの対角成分が求まった。

そしたら、式(3)より、行列Bの任意の成分が以下のように計算できる。

$$b_{ij} = \frac{1}{2}\left(b_{ii} + b_{jj} - d_{ij}^2\right) \tag{8}$$

以上で行列Bが求まったので、後は、行列Xを計算するだけだ。

行列Bは、$B = XX^T$なので、当然、対称行列だよね。式(2)からも、対称行列だと分かるね。対称行列は、次のように特異値分解できることが分かっている。

$$B = V\Sigma V^T \tag{9}$$

※一般的な行列の特異値分解が$A = U\Sigma V^T$と違って、$U = V$となることに注目して欲しい！

ここで、ご存知の通り、Σは固有値が並んだ対角行列だ。

$$\Sigma = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_N \end{bmatrix}$$

なので、

$$\Sigma = \Sigma^{1/2}\Sigma^{1/2} = \Sigma^{1/2}\left(\Sigma^{1/2}\right)^T$$

よって、式(9)は次のように変形できる。

$$B = V\Sigma V^T = V\Sigma^{1/2}\left(V\Sigma^{1/2}\right)^T$$

つまり、

$$X = V\Sigma^{1/2} \tag{10}$$

ってことだね！

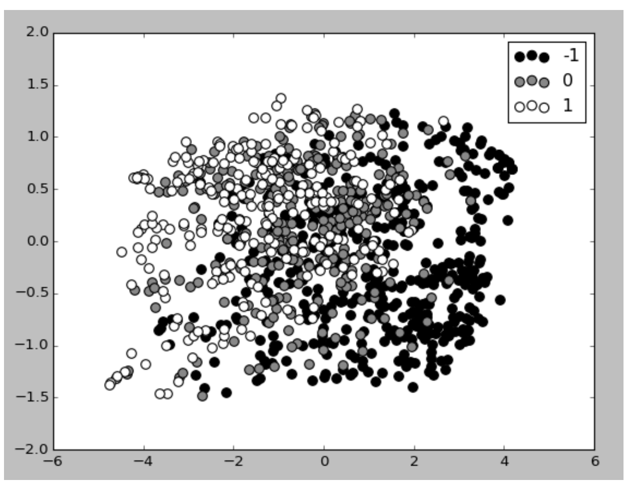
以上より、行列Xを求めることが出来る。

Isomap

Isomap も同様に、manifold.Isomap()関数で簡単に実現できる。

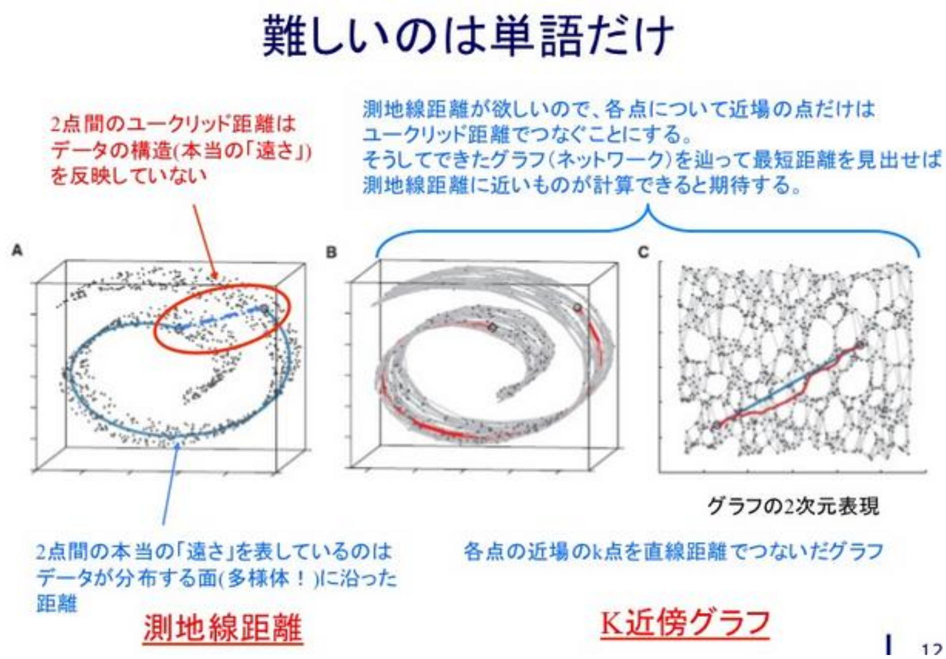
python コード

```
...
# Isomap オブジェクトを生成（2 次元に落とし込む）
isomap = manifold.MDS(n_neighbors=10, n_components=2)

# manifold 空間での 2 次元座標を計算
pos = isomap.fit_transform(data)

# 表示
plt.scatter(pos[:,0], pos[:,1])
plt.show()
```

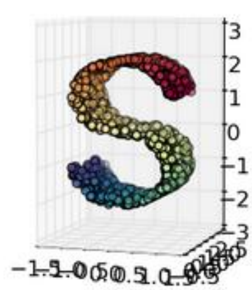


数学的な話

k近傍点とのユークリッド距離に基づいてグラフを作成し、グラフ上での各 2 点間の最短距離を **geodesics** 距離として計算する。以下のスライドが分かりやすいかな。



これを使って、あとは普通に MDS で 2 次元に表示するだけ。純粋な MDS と違って、*k*近傍点のみを使うので、例えば sklearn のウェブにある下図のような構造を持つデータの時に、うまいこと S 字に抽出できるわけだ。



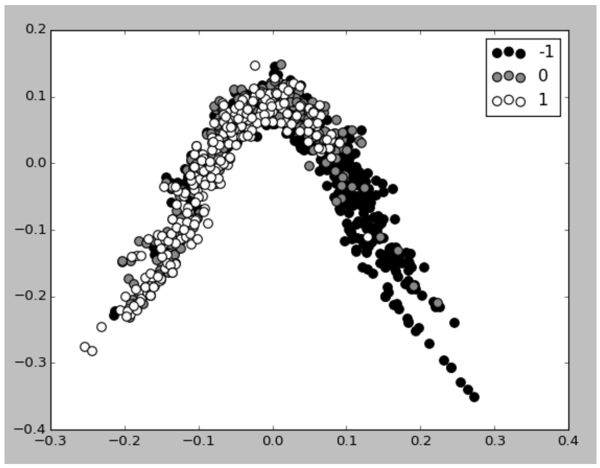
Spectral Embedding

これも同様に、manifold.SpectralEmbedding()関数で簡単に実現できる。簡単すぎて、ちょっと恐い。

python コード

```
...
# Spectral Embedding オブジェクトを生成（2 次元に落とし込む）
se = manifold.SpectralEmbedding(n_neighbors=10, n_components=2)
```

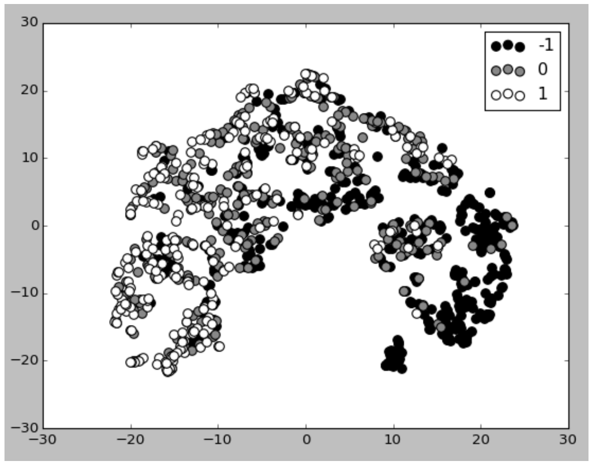
```
# manifold 空間での 2 次元座標を計算
pos = se.fit_transform(data)

# 表示
plt.scatter(pos[:,0], pos[:,1])
plt.show()
```



t-distributed stochastic neighbor embedding (t-SNE)

python コード

```
...
# t-SNE オブジェクトを生成（2 次元に落とし込む）
tsne = manifold.TSNE (n_components=2 , init='pca', random_state=0)

# manifold 空間での 2 次元座標を計算
pos = tsne.fit_transform(data)

# 表示
plt.scatter(pos[:,0], pos[:,1])
plt.show()
```



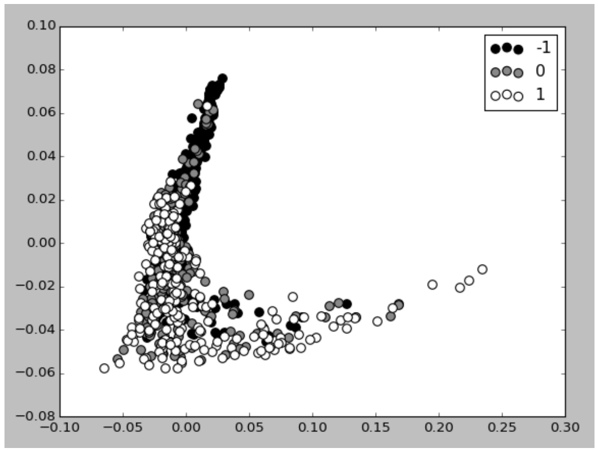
Locally Linear Embedding

Locally linear embedding は、manifold.LocallyLinearEmbedding()関数で簡単に実現できる。ただし、引数に、method='standard'を指定する。この引数を変えることで、LTSA、Hessian LLE、Modified LLE などを使用できる。

※俺のデータセットでは、なぜか LTSA と Hessian はエラーが出て失敗した。

python コード

```
...
# Locally linear embedding オブジェクトを生成（2 次元に落とし込む）
lle = manifold.LocallyLinearEmbedding(n_components=2 , n_neighbors=10,
eigen_solver='auto', method='standard')

# manifold 空間での 2 次元座標を計算
pos = lle.fit_transform(data)

# 表示
plt.scatter(pos[:,0], pos[:,1])
plt.show()
```

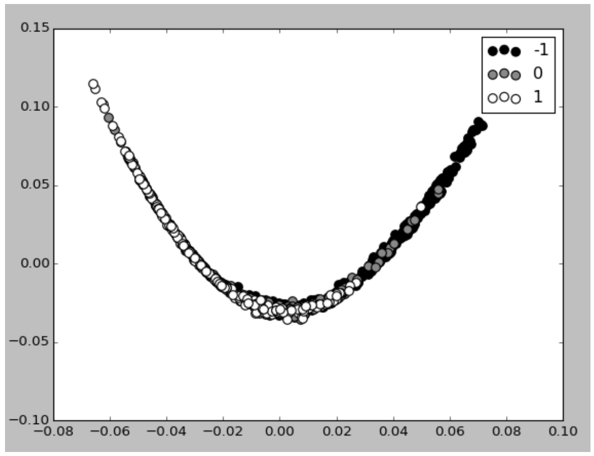


Modified LLE

manifold.LocallyLinearEmbedding()関数で簡単に実現できる。ただし、引数に、method='modified' を指定する。

python コード

```
...
# Locally linear embedding オブジェクトを生成（2 次元に落とし込む）
lle = manifold. LocallyLinearEmbedding(n_components=2 , n_neighbors=10,
eigen_solver='auto', method='modified')


# manifold 空間での 2 次元座標を計算
pos = lle.fit_transform(data)


# 表示
plt.scatter(pos[:,0], pos[:,1])
plt.show()
```



所感

2 次元でみると、ある程度はうまくグループ分けが出来ているケースもある。これは、Inverse 方向で linear regression がそこそこ良い結果が出したことを示唆しているのかも知れない。

しかし一方で、ものすごい近い位置にあるにもかかわらず、異なるラベルのものもある。これは、linear regression では無理があることも示唆していると思う。

そもそも、ものすごい近くなのに、異なるラベルがあるわけだから、quadratic や何やらと、次元を一旦増やしてやる必要があるのだろう。

あるいは、何か別の nonlinear dimensionality reduction では、うまく分類できるのか？

まずは、quadratic をやってみたいねぇ。。。