Machine Learning Homework3

Wenjian Hu, Jian Xu

We choose to do a) Kernelizing methods

1. Using two dimension reduction techniques (PCA, LLE, GDDR or something else of your choosing) project the images into a three dimensional space. Summarize the projection by specifying: a) the projection vectors and b) a scatter matrix of the position of images in the three dimensional space and c) a concise description of the results of the projection.

PCA method

Recall the input data matrix of N points is $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_N]^T$ where each $\mathbf{x}$ is a D-dimensional vector. PCA finds a projection matrix $\mathbf{P} = [\mathbf{p}_1, ..., \mathbf{p}_d]$ that maps each point to a low-dimensional space ($d \leq D$). As described, each $\mathbf{p}$ is a basis vector that maximizes the variance of $\mathbf{X}$ in orthogonal directions with respect to each other, and that the amount of variance preserved decreases from $\mathbf{p}_1$ to $\mathbf{p}_d$.

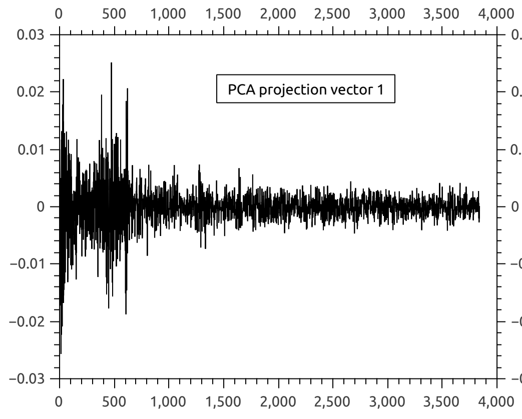
By definition, the covariance matrix of $\mathbf{X}$ is:

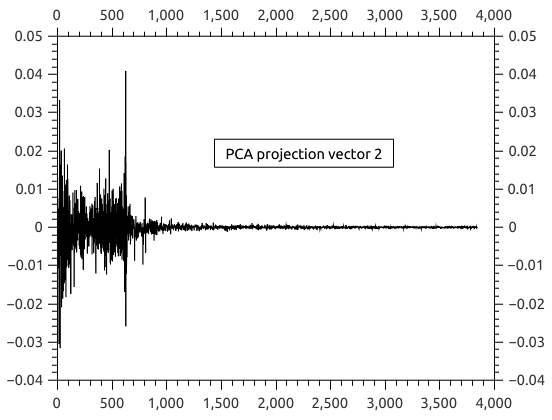
$$\mathbf{C} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \mu)(\mathbf{x}_n - \mu)^T$$

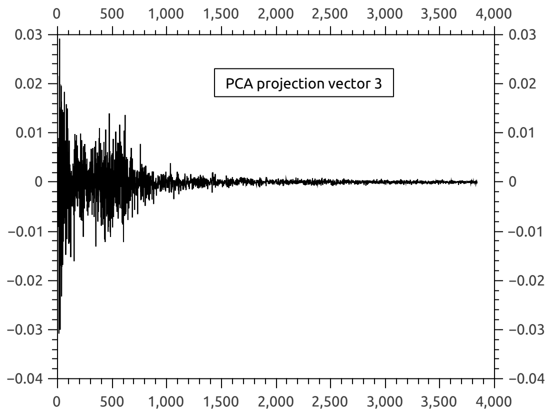
Where $\mu = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n$ is the mean vector.

Finding projection vectors is equivalent in solving an eigendecomposition problem for the covariance matrix $\mathbf{C}$, thus we could solve the equation $det(\mathbf{C} - \lambda\mathbf{I}) = 0$ to get corresponding eigenvectors(projection vectors) and eigenvalues. After we have projection vectors, we could form a projection matrix $\mathbf{P} = [\mathbf{p}_1, ..., \mathbf{p}_d]^T$ and get new projected data $\mathbf{Y} = [\mathbf{y}_1, ..., \mathbf{y}_N]^T$ by the relationship $\mathbf{Y} = \mathbf{P}^T\mathbf{X}$. Here are results.

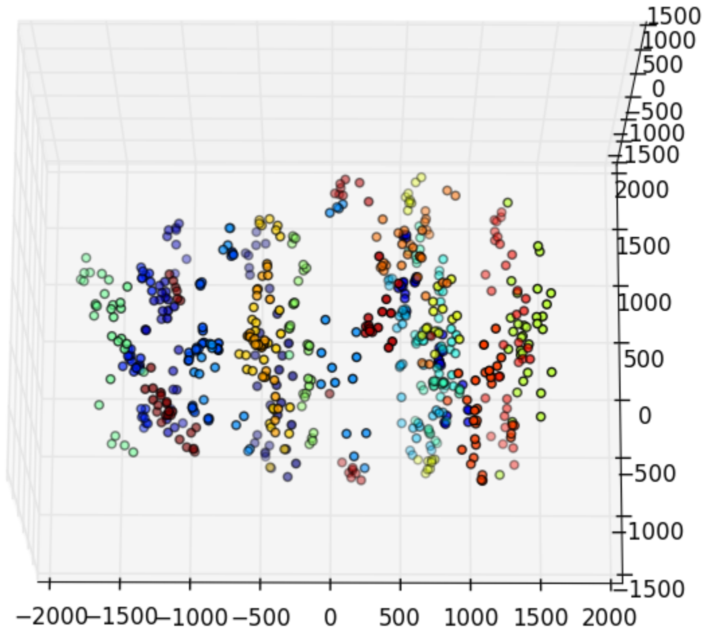
(a) The projection vectors:
Three projection vectors are attached, with the file name "PCA projection vectors 1","PCA projection vectors 2" and "PCA projection vectors 3". To visualize these three vectors, I plot them as follow (y axis is each dimension's value, x axis corresponds to its dimensions, there are 60×64 dimensions):



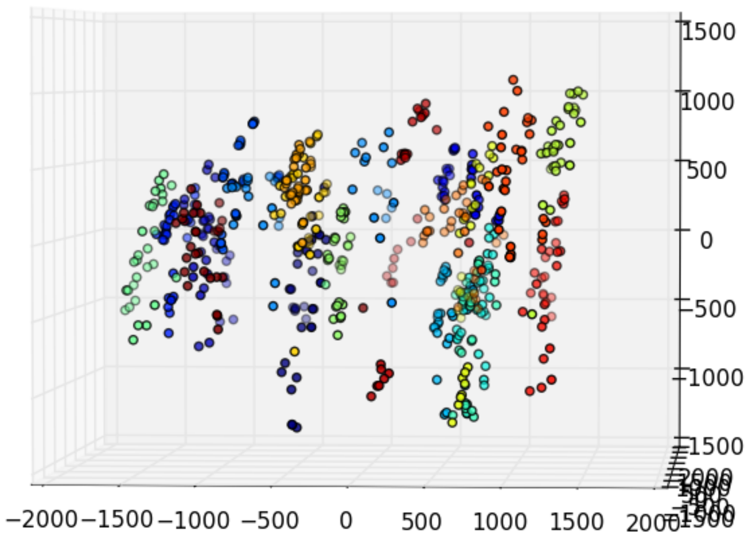




(b) a scatter matrix of the position of images in the three dimensional space:
The scatter matrix of the position is attached, with the file name ”PCA scatter matrix in 3D”. To visualize them, I plot the 3D graph as follow (**points with the same color corresponds to the same name**):



Another point of view to the 3D graph:



(c) a concise description of the results of the projection:
I think the result of PCA is good. We can distinguish most of the points with the different names (colors), although there exists one defect that some points belonging to different names are mixed up. Another defect is that points with the same name spread out a little too much.

LLE method

LLE consists of three steps: nearest neighbor search (to identify the nonzero elements of the weight matrix), constrained least squares fits (to compute the values of these weights), and singular value decomposition (to perform the embedding).

Step 1: Neighborhood Search

The first step of LLE is to identify the neighborhood of each data point. In the simplest formulation of the algorithm, one identifies a fixed number of nearest neighbors, K, per data point, as measured by Euclidean distance.

Step 2: Constrained Least Squares Fits

Consider a particular data point $\mathbf{x}_i$ with K nearest neighbors $\mathbf{x}_j$ and reconstruction weights w_{ij} that sum to one. We can write the reconstruction error as:

$$\epsilon_i = |\mathbf{x}_i - \sum_j w_{ij}\mathbf{x}_j| = |\sum_j w_{ij}(\mathbf{x}_i - \mathbf{x}_j)| = \sum_{jk} w_{ij} w_{ik} G^{(i)}_{jk}$$

where in the first identity, we have exploited the fact that the weights sum to one, and in the second identity, we have introduced the local Gram matrix,

$$G^{(i)}_{jk} =< \mathbf{x}_i - \mathbf{x}_j, \mathbf{x}_i - \mathbf{x}_k >= (\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_k)$$

By construction, this Gram matrix is symmetric and semipositive definite. The reconstruction error can be minimized analytically using a Lagrange multiplier to enforce the constraint that $\sum_j w_{ij} = 1$. In terms of the inverse Gram matrix, the optimal weights are given by:

$$w_{ij} = \frac{\sum_k (G^{(i)})^{-1}_{jk}}{\sum_{lm} (G^{(i)})^{-1}_{lm}}$$

Define the n-by-n sparse Weight Matrix $\mathbf{W}$, where $W[i][j] = w_{ij}$, and the other elements of $\mathbf{W}$ are 0. In general, $\mathbf{W}$ is a sparse matrix consisting of neighborhood characteristic of the original space.

Step 3: Eigenvalue Problem

Fix the optimal weight matrix $\mathbf{W}$ found at step 2. Find the $(t \times n)$-matrix $\mathbf{Y} = \mathbf{P}^T\mathbf{X}$ of embedding coordinates that solves

$$argmin_{\mathbf{Y}} \sum_{i=1}^{n} |\mathbf{y}_i - \sum_{j=1}^{n} w_{ij}\mathbf{y}_j|$$

subject to the constraints $\sum_i \mathbf{y}_i = \mathbf{0}$ and $\frac{1}{N}\sum_i \mathbf{y}_i\mathbf{y}_i^T = \mathbf{I}$.
Under these restrictions, the optimal embedding can be done in many ways, but the most straightforward is to find the bottom $d+1$ eigenvectors of the cost matrix M:

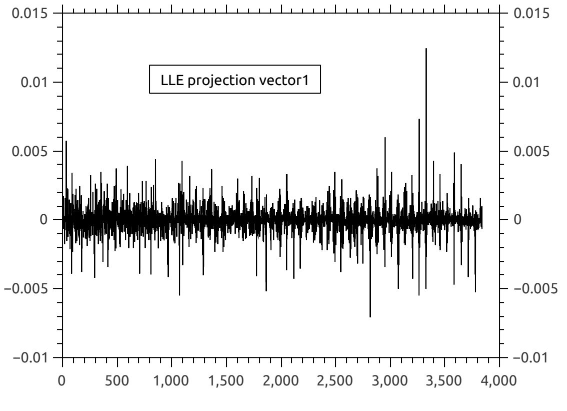
$$\mathbf{M} = (\mathbf{I} - \mathbf{W})^T(\mathbf{I} - \mathbf{W})$$

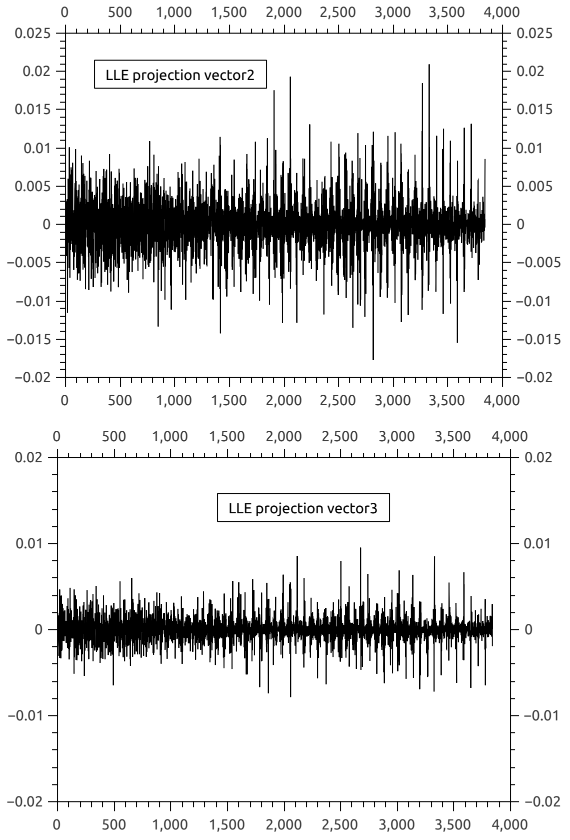
The bottom eigenvector of the matrix $\mathbf{M}$, which we discard, is the unit vector with all equal components; it represents the free translation mode of eigenvalue zero. The remaining d eigenvectors constitute the d embedding coordinates found by LLE. After we get the embedding coordinates $\mathbf{Y}$, we can simply solve the equation $\mathbf{Y} = \mathbf{P}^T\mathbf{X}$ to get the projection vectors

$$\mathbf{P}^T = \mathbf{Y}\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}$$

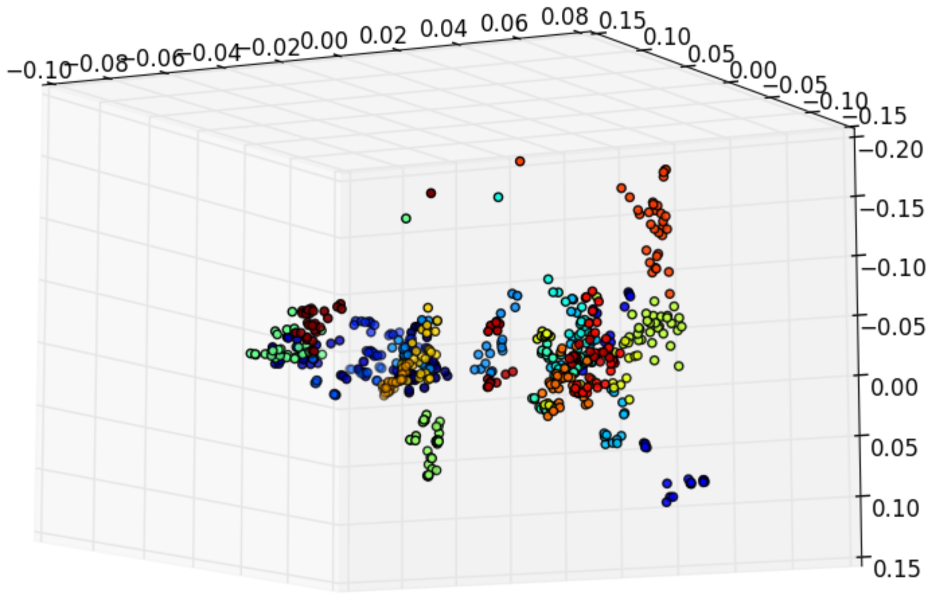
In the following mapping, I choose the nearest neighbors $K = 40$

(a) The projection vectors:
Three projection vectors are attached, with the file name "LLE projection vectors 1","LLE projection vectors 2" and "LLE projection vectors 3". To visualize these three vectors, I plot them as follow (y axis is each dimension's value, x axis corresponds to its dimensions, there are 60×64 dimensions):

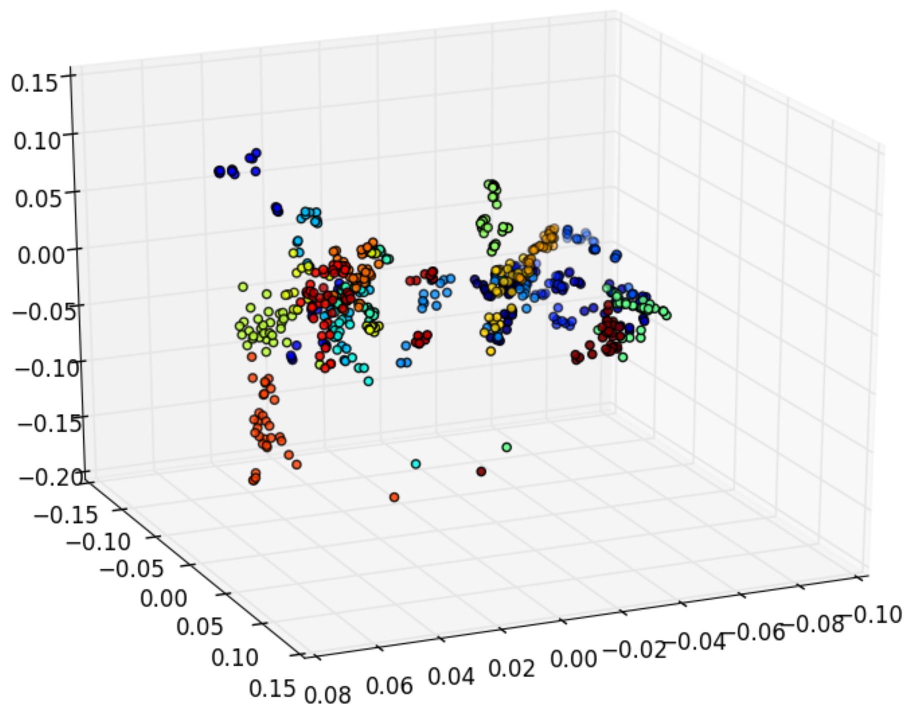




(b) a scatter matrix of the position of images in the three dimensional space:
The scatter matrix of the position is attached, with the file name "LLE scatter matrix in 3D". To visualize them, I plot the 3D graph as follow (**points with the same color corresponds to the same name**):



Another point of view to the 3D graph:



(c) a concise description of the results of the projection:
I think the result of PCA is also good. We can also distinguish most of the points with the different names (colors), although some points belonging to different names are mixed up. If points with different names (colors) can be a little more further, it will be better. To achieve this, we might need to tune the nearest neighbors K carefully.

2. Take one of the two dimension reduction techniques from 1) and kernelize it. Describe: a) how your kernelize the approach and the Kernel used and b) a scatter matrix of the position of images in the three dimensional space and c) a concise description of the results of the projection.

kernel PCA

(a) how your kernelize the approach and the Kernel used:
Assume for the moment that our data mapped into feature space, $\phi(\mathbf{x}_1), ..., \phi(\mathbf{x}_N)$, is centered, i.e. $\sum_{i=1}^{N} \phi(\mathbf{x}_i) = \mathbf{0}$. To do PCA for the covariance matrix

$$\mathbf{C} = \frac{1}{N} \sum_{j=1}^{N} \phi(\mathbf{x}_j)\phi(\mathbf{x}_j)^T$$

we have to find Eigenvalues $\lambda \geq 0$ and Eigenvectors $\mathbf{V}$ satisfying $\lambda\mathbf{V} = \mathbf{CV}$. We note that all solutions $\mathbf{V}$ lie in the span of $\phi(\mathbf{x}_1), ..., \phi(\mathbf{x}_N)$. This implies that we may consider the equivalent system

$$\lambda(\phi(\mathbf{x}_k)\mathbf{V}) = (\phi(\mathbf{x}_k)\mathbf{CV}), k = 1, ..., N$$

and that there exist coefficients $\alpha_1, ..., \alpha_N$ such that

$$\mathbf{V} = \sum_{i=1}^{N} \alpha_i \phi(\mathbf{x}_i)$$

defning an $N \times N$ matrix $\mathbf{K}$ by

$$K_{ij} :=< \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) >= \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$$

we arrive at

$$N\lambda \mathbf{K}\alpha = \mathbf{K}^2\alpha$$

where α denotes the column vector with entries $\alpha_1, ..., \alpha_N$. To find the solution, we solve the Eigenvalue problem

$$N\lambda\alpha = \mathbf{K}\alpha$$

for nonzero Eigenvalues. Then We normalize the k-th solutions α^k belonging to nonzero Eigenvalues by requiring that the corresponding vectors to be normalized i.e $< \mathbf{V}^k, \mathbf{V}^k >= 1$. Thus, we have

$$1 = \sum_{i,j=1}^{N} \alpha_i^k \alpha_j^k < \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) >=< \alpha^k, \mathbf{K}\alpha^k >= \lambda_k < \alpha^k, \alpha^k >$$

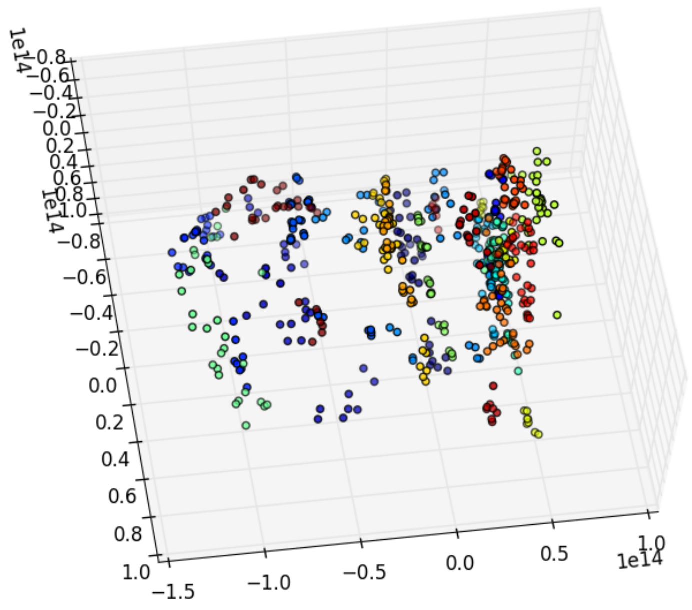
For principal component extraction, we compute projections of the image of a test point $\phi(\mathbf{x})$ onto the Eigenvectors $\mathbf{V}^k$ according to

$$< \mathbf{V}^k, \phi(\mathbf{x}) >= \sum_{i=1}^{N} \alpha_i^k < \phi(\mathbf{x}_i), \phi(\mathbf{x}) >$$

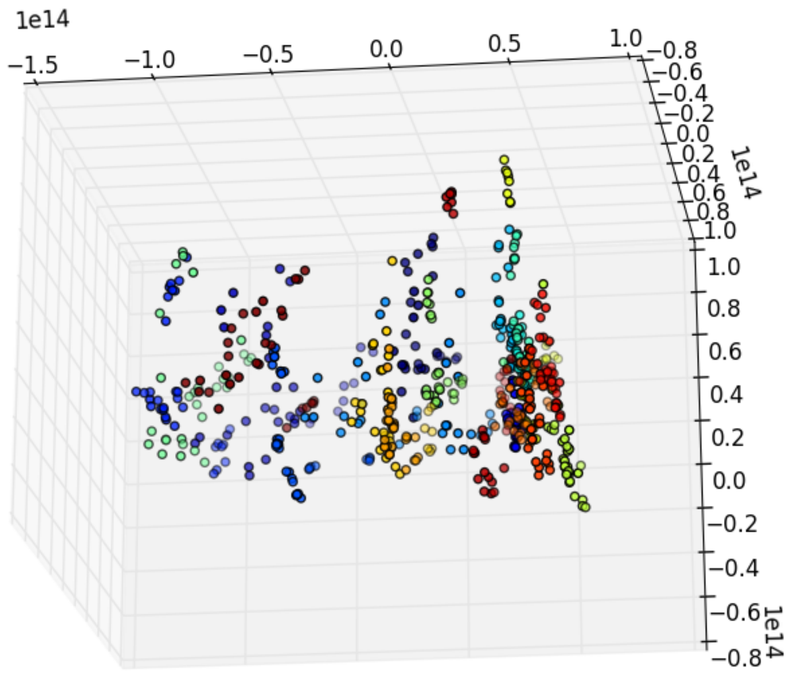
None of the above form requires the $\phi(\mathbf{x}_i)$ in explicit form - they are only needed in dot products. Therefore, we are able to use kernel functions for computing these dot products without actually performing the map ϕ.

In our mapping, we choose the polynomial kernels of degree 4.

(b) a scatter matrix of the position of images in the three dimensional space:
The scatter matrix of the position is attached, with the file name "kernelPCA scatter matrix in 3D". To visualize them, I plot the 3D graph as follow (**points with the same color corresponds to the same name**):



Another point of view to the 3D graph:



(c) a concise description of the results of the projection:
I think the result of kernelPCA is good too! We can also distinguish most of the points with the different names (colors), although some points belonging to different names are mixed up. Most points with the same name are close to each other and far away from other name points. Compared with the conventional PCA, which corresponds to the 'linear' kernel in kernel PCA, kernel PCA with the polynomial kernel produces similar results but with much larger separation between two points. Also the overall distribution is much more spread out for the polynomial kernel PCA.

I also attach my python code, with the name "Dimension Reduction.py"