Unsupervised Statistical Learning: Back to Dimension Reduction

Siva Balakrishnan
Data Mining: 36-462/36-662

April 9th, 2019

Outline for Today

- Recap: Spectral Clustering
- More Algorithms for Dimension Reduction

Recap: The Graph Laplacian

The matrix:

$$L = D - W,$$

is called the *Graph Laplacian*. It is a symmetric, real-valued matrix, so it has an eigendecomposition. We have already seen that for any vector v:

$$v^T L v = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} W_{ij} (v(i) - v(j))^2 \geq 0,$$

so all its eigenvalues are positive.

Intuition: Bottom few eigenvectors are very smooth (as close to constant as they can be) and are very useful for partitioning the graph.

Recap: Cuts and Eigenvectors

If we want to find the minimum balanced cut we can instead solve the following problem:

$$\underline{\arg\min_v v^T(D-W)v}$$

min balanced.

- Entries of v are all $\underline{\{+1,-1\}}$ (so it is a cut vector). ×
- Entries of $\underline{v \text{ sum to } 0}$ (so it is balanced).

Spectral clustering into two clusters is a *relaxation* of this. We solve:

$$\arg\min_v v^T(D-W)v$$

find 2nd smallest EV of L.

- Entries of v sum to 0 (so it is balanced),

and then threshold the entries of v to find our clusters.

Recap: Basic Spectral Clustering Algorithm

If we want to cluster our data into two clusters we will follow these steps:

- Build a (weighted) graph on the data points (in one of three ways).
- Construct the graph Laplacian matrix, i.e. compute the matrix $D - W$.
- Find its second-smallest eigenvector v_2.
- Threshold its entries to find the clusters, i.e. take $A = \{i : v_2(i) > 0\}$, and $B = \{i : v_2(i) \leq 0\}$.

The second smallest eigenvector of the Laplacian has its own name (Fiedler vector).

Recap: Algorithm for clustering into k-clusters

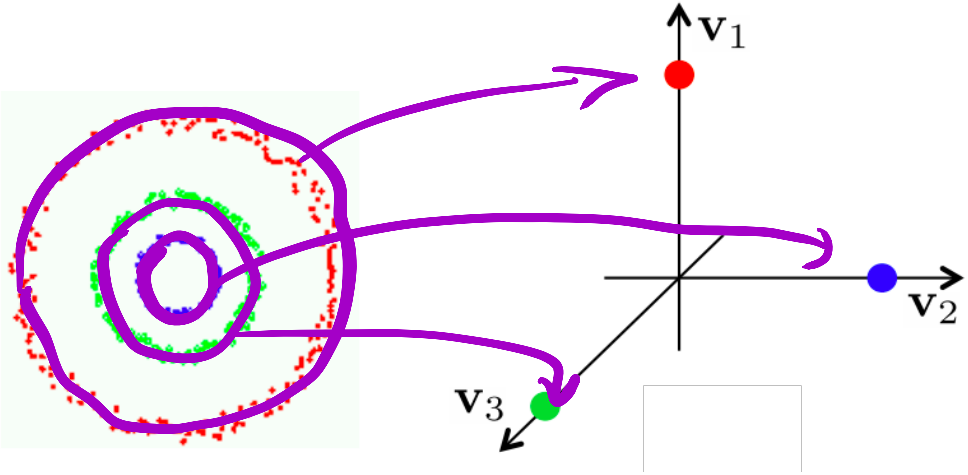
If we want to cluster our data into k-clusters we will follow these steps:

- ✓ Build a (weighted) graph on the data points (in one of three ways).
- ✓ Construct the graph Laplacian matrix, i.e. compute the matrix $D - W$.
- Find its smallest k eigenvectors $\{v_1, v_2, \ldots, v_k\}$, put them in a matrix $\underline{\underline{V}} \in \mathbb{R}^{n \times k}$. → spectral embedding.
- Interpret the rows of V as our data points. Run k-means on this data to find k-clusters.

Key Point: Spectral Embedding (i.e. mapping into coordinates defined by the eigenvectors of the Laplacian) tends to separate graph clusters.

Recap: The Spectral Embedding

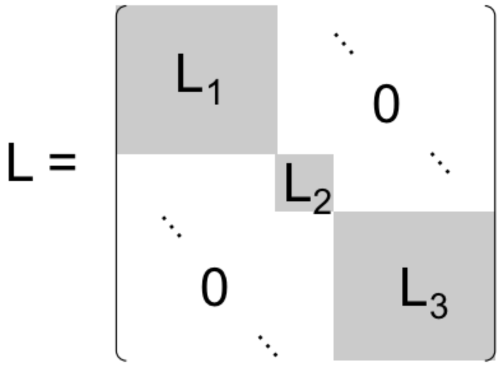
Just some intuition through pictures:



Key point: k-means on the embedding performs much better than k-means on the original data.

Recap: Clustering disconnected graphs

The Laplacian of a disconnected graph looks like this:



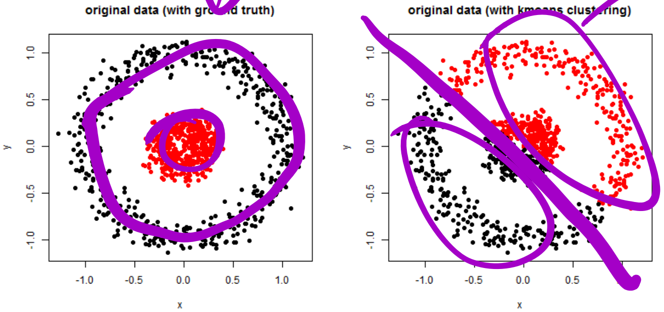
What are the bottom k eigenvectors of this matrix?

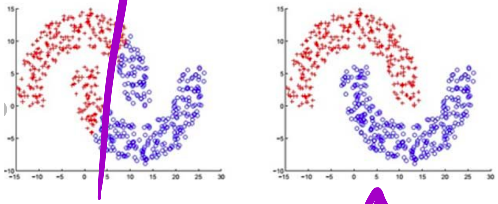
$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Perfect for running k-means.

Recap: Spectral clustering v/s k-means

k-means.





- If clusters are (well-separated) blobs: k-means will do well but so will spectral clustering.
- If clusters are dense but have strange shapes then spectral clustering will typically do much better.

Recap: Spectral Clustering in Practice

- People often use the *normalized Laplacian* instead of the graph Laplacian.
- The normalized Laplacian divides each entry of the Laplacian by the square root of the degrees of the two corresponding nodes, i.e.:

$$L_{\text{normalized}} = D^{-1/2} L D^{-1/2}.$$

  So,

$$L_{\text{normalized}}(i,j) = \frac{L(i,j)}{\sqrt{d_i d_j}}.$$

  The normalized Laplacian has many of the same properties as the graph Laplacian but tends to give better clusters in practice.
- The rest of the algorithm is identical (compute the bottom k eigenvectors and run k-means on them).

Recap: Spectral Clustering in Practice

- The algorithm now has more tuning parameters than k-means.

  1. **The number of clusters k:** In the ideal case, we know we should have k eigenvalues close to 0, and the $k+1$ eigenvalue should be large. So in practice we often look for the first large gap in the eigenvalues. Roughly,

     $$k^* = \arg\max_k |\lambda_k - \lambda_{k+1}|.$$

  2. **The choice of the similarity graph:** Need to choose between k-NN similarity, ϵ-neighborhood and weighted. In practice, the k-NN graph often works well (and is a good starting point). It also gives us a sparse graph (which is useful computationally).
  3. **The choice of k in the k-NN graph:** Again, a very hard question to answer. Often the heuristic is to make sure k is large enough so that the resulting graph has very few disconnected components. (If the graph has many disconnected components then spectral clustering just returns some subset of those components.)

Back to data visualization and PCA

$X \in \mathbb{R}^{n\times d}.$

$$X = \overset{n\times d.}{U} \times \overset{d\times d}{D} \times V^T. \leftarrow \text{SVD}(X).$$

just use columns of $U\times D$.

A curious fact:

- Recall, that in order to visualize data using PCA, all we needed we the principal component scores (i.e. the things in the matrix $U \times D$). Why?
- Suppose instead of giving you the matrix X (i.e. the data) I only gave you the matrix X^TX. Could you still "visualize" the data?
- How about if I gave you the matrix XX^T?

what is it dimension? $d\times d$

Using $\underline{X^T X}$ / n we can compute V.

$$\frac{X^T X}{n} = \frac{(U \times D \times V^T)^T}{n} \frac{(U \times D \times V^T)}{n}$$

$$= V\left(\frac{D^2}{n^2}\right)V^T.$$

$UD.$ $\quad , (V, D)$

$$XX^T = \underline{UD^2U^T.} \longrightarrow \text{can compute } U \times D.$$

The matrix XX^T $(X^TX)^T = X^TX.$

- The matrix $\underline{\underline{XX^T}}$ $\in \mathbb{R}^{n\times n}.$ is called the *inner-product matrix*. Why?

$$(XX^T)_{ij} = x_i^T x_j.$$

- Our curious fact in words: We can visualize the data even without having the original data. Using PCA (or an eigendecomposition) we can go from similarities to a meaningful point cloud.

Kernel PCA

The fact that we can do PCA with just the inner-products between the data points should suggest something. Why not kernelize PCA?

Kernel PCA algorithm: $x_i^T x_j$ replace $K(x_i, x_j)$.

- First compute the gram-matrix:

$$K = \begin{bmatrix} K_{11} & K_{12} & \dots & K_{1n} \\ \vdots & & & \\ K_{n1} & \dots & K_{n(n-1)} & K_{nn} \end{bmatrix}, \in \mathbb{R}^{n \times n}.$$

  where

$$\text{RBF kernel} \Leftarrow K_{ij} = \exp\left(-\frac{\|X_i - X_j\|_2^2}{\gamma^2}\right),$$

$$(XX^T) = UD^2U^T.$$

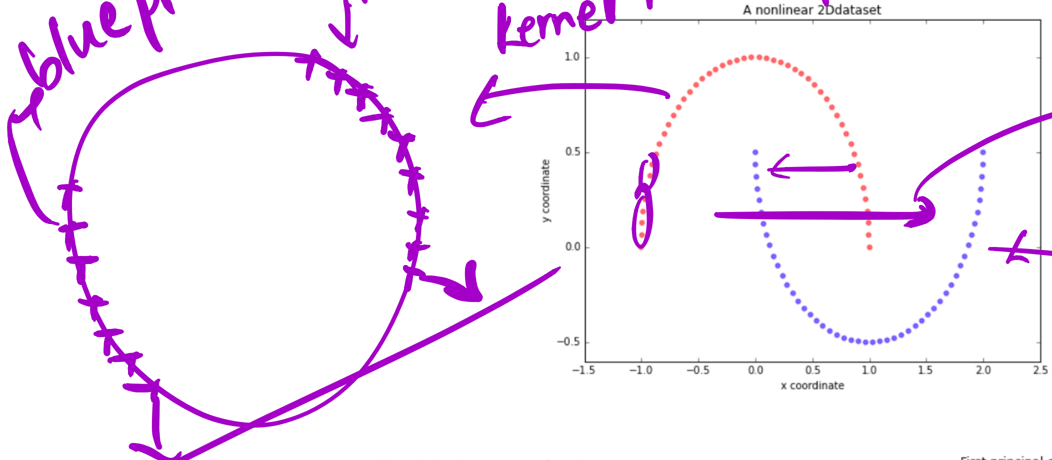
  (usually also need to center K).

- Compute the eigendecomposition of $K = U \times D \times U^T$, and use $\underline{U \times \sqrt{D}}$ as our PC scores. → pick 2 columns and plot.

Might seem counter-intuitive – we are first increasing the dimension (by our feature map) and then reducing it (by PCA).

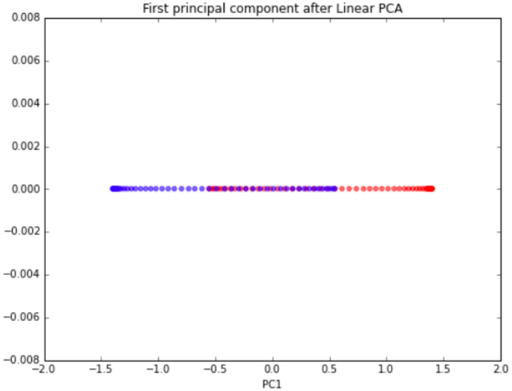
The power of non-linear dimensionality reduction

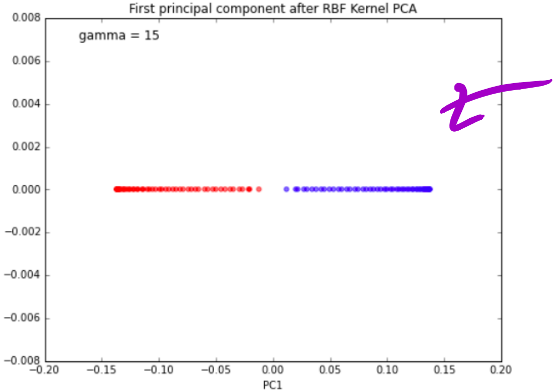
Some examples:



direction of max variance.

no good linear projections.

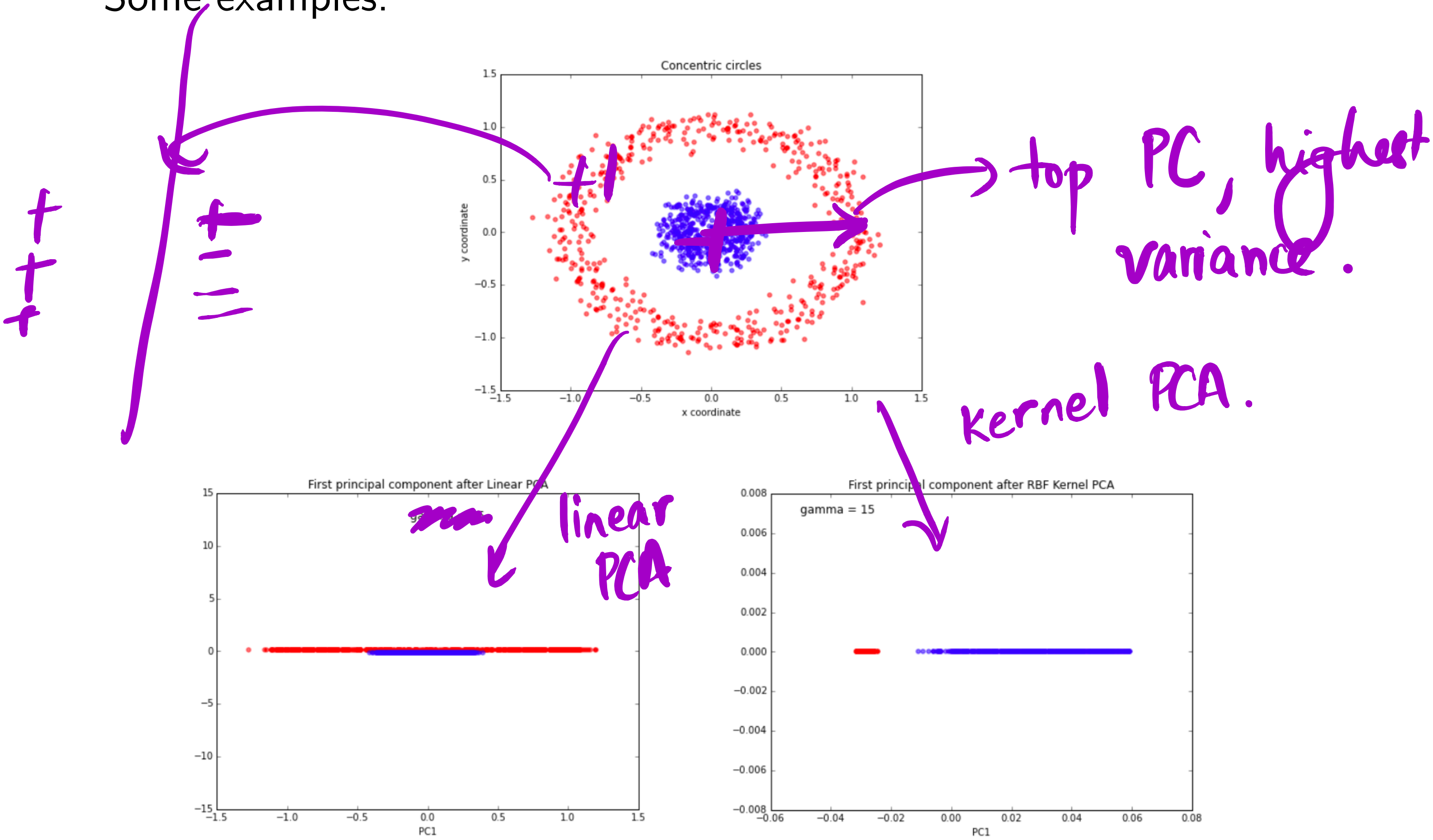




kernel PCA.

The power of non-linear dimensionality reduction

Some examples:



More curious facts: Multi-dimensional Scaling

We are going to go from similarities to distances.
We have seen that we can visualize just given XX^T. Suppose instead I just gave you the matrix $\Delta \in \mathbb{R}^{n \times n}$ of pairwise distances:

$$\Delta_{ij} = \|X_i - X_j\|_2.$$

k=2

We want a lower-dimensional representation $z_1, \ldots z_n \in \mathbb{R}^k$, for some small k (e.g. $k = 2$ or 3), such that $\|z_i - z_j\|_2 \approx \Delta_{ij}$, for every $i, j = 1, \ldots n$

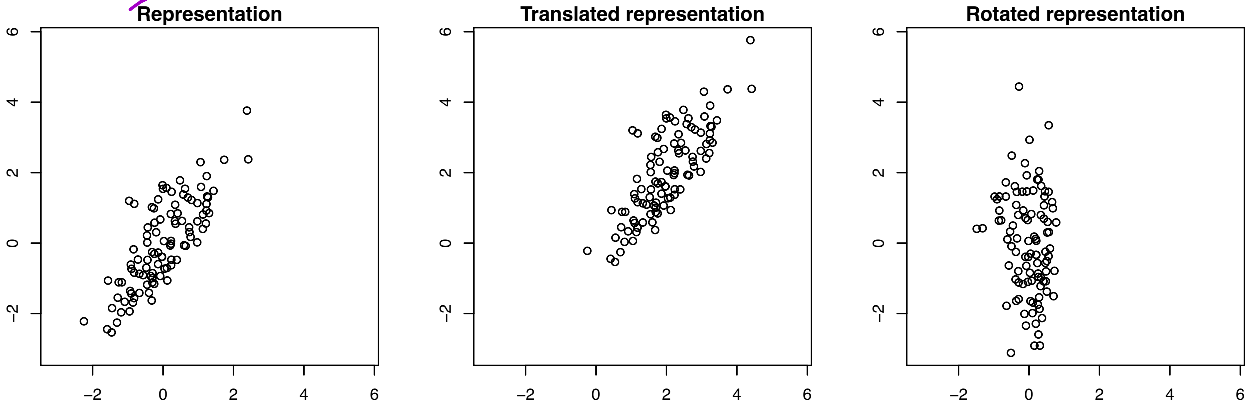
We saw that the principal component scores do exactly this, but these rely on X or XX^T, which we don't have here. We can only use the distances Δ_{ij}

Is this possible?

Non-identifiability

X has mean 0.

↳ dist matrix has less information.



Distances Δ_{ij} are invariant under any orthonormal transformation $O \in \mathbb{R}^{p \times p}$ of $x_1, \ldots x_n$ (i.e., $O^T O = I$)

Multidimensional scaling

Assume that $X \in \mathbb{R}^{n \times p}$ has been column centered (this is enough to deal with translation unidentifiability)

Multidimensional scaling (MDS): given distance matrix $\Delta \in \mathbb{R}^{n \times n}$, we

1. Recover the inner-product matrix $\boxed{B = XX^T} \in \mathbb{R}^{n \times n}$ → how?
2. Factorize B to get the first k principal component scores

(Called classical multidimensional scaling, there are other flavors, e.g., least squares multidimensional scaling)

We've already seen how to do step 2: remember that we compute the eigendecomposition $\underline{B = UDU^T}$ and then the first k principal component scores are the first k columns of $\underline{U\sqrt{D}}$

So, how do we do step 1, i.e., how do we recover B?

$$\|x_i - x_j\|_2^2 = (x_i - x_j)^T (x_i - x_j)$$

$$= \underline{x_i^T x_i + x_j^T x_j} - 2 x_i^T x_j$$

given

want this.

$$B = XX^T$$

$$\boxed{\Delta_{ij}^2 = B_{ii} + B_{jj} - 2 B_{ij}}$$

Recovering inner-products from distances

The following procedure can be used to recover the inner-products $B = XX^T$ from Δ:

1. Compute $A_{ij} = \underline{-\frac{1}{2}\Delta_{ij}^2}$ to form the matrix $A \in \mathbb{R}^{n \times n}$

2. Easy to understand way: Solve the system of equations to obtain B:

$$A_{ij} = B_{ij} - (B_{ii} + B_{jj})/2.$$

3. Alternative fancy way: Double center A—i.e., center both the columns and rows of A—to recover the matrix $B \in \mathbb{R}^{n \times n}$. Note that this is the same as:

$$B = (I - M)A(I - M)$$

where $M = \frac{1}{n}\mathbf{1}\mathbf{1}^T \in \mathbb{R}^{n \times n}$

Beyond Euclidean distance

If the Δ_{ij} were actual Euclidean distances between the rows of a centered matrix $X \in \mathbb{R}^{n \times p}$, we get back the first k principal component scores exactly.

MDS would be just like doing PCA, except the result would not be identified up to an orthogonal transformation. Why do it?

Importantly, multidimensional scaling can be applied to any Δ_{ij}, not just Euclidean distances. can compute non-linear dist.

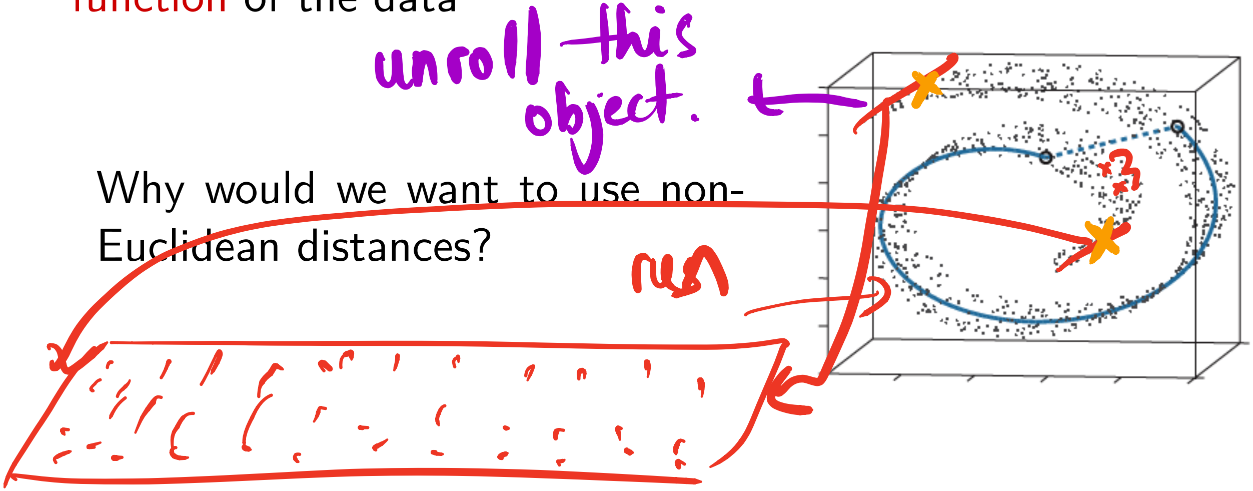
In these settings, MDS finds a set of points $Z \in \mathbb{R}^{n \times k}$ so that the distances between rows $x_i, x_{i'}$ are close to $\Delta_{ii'}$. & do then MDS.

Beyond Euclidean distance

There is a class of methods which construct a fancier metric Δ_{ij} between high-dimensional points $x_1, \ldots x_n \in \mathbb{R}^p$, and then they feed these Δ_{ij} through multidimensional scaling to get a low-dimensional representation $z_1, \ldots z_n \in \mathbb{R}^k$.

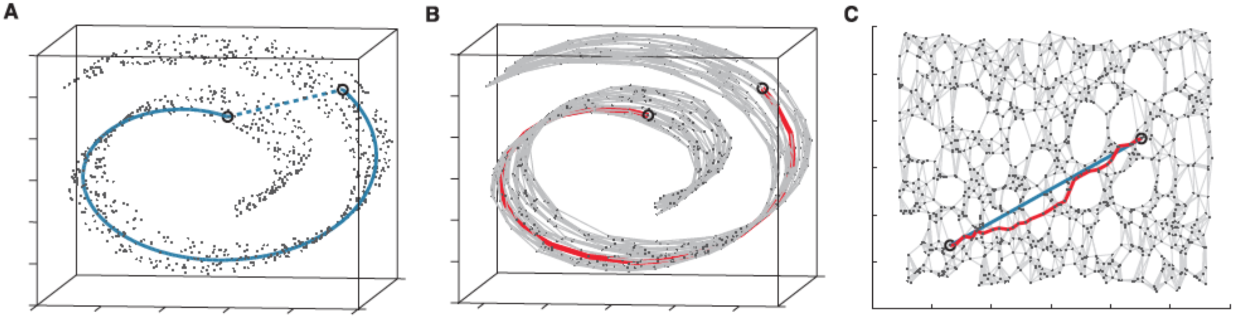
In this case, we don't just get principal component scores, and our low-dimensional representation can end up being a nonlinear function of the data

Why would we want to use non-Euclidean distances?



Isometric feature mapping

Isometric feature mapping[1] (Isomap) learns structure in a more general setting to define distances. The basic idea is to construct a graph $G = (V, E)$, i.e., construct edges E between vertices $V = \{1, \dots n\}$, based on the structure between $x_1, \dots x_n \in \mathbb{R}^p$. Then we define a graph distance $\{\Delta_{ij}^{\text{Isomap}}\}.$ between i and j, and use multidimensional scaling for our low-dimensional representation



(From Tenenbaum et al. (2000))

[1] Tenenbaum et al. (2000), "A global geometric framework for nonlinear dimensionality reduction"

The Isomap graph distances

Constructing the graph: for each pair i, j, we connect i, j with an edge if either:

- x_i is one of x_j's k nearest neighbors, or
- x_j is one of x_i's k nearest neighbors

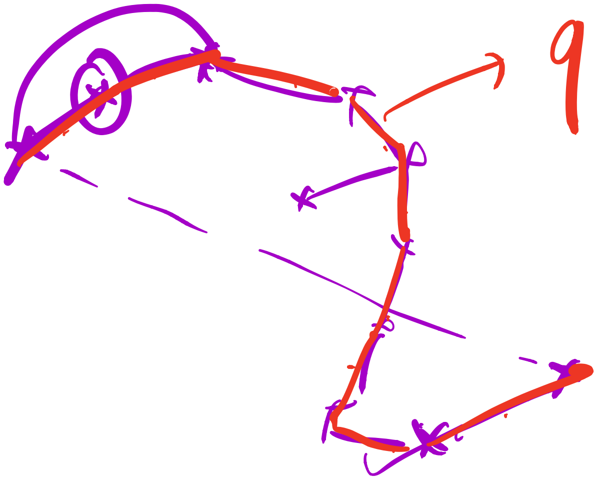
The weight of this edge $e = \{i, j\}$ is then $w_e = \|x_i - x_j\|_2$ (can also ignore the weights)

Defining graph distances: now that we have built a graph, i.e., we have built an edge set E, we define the graph distance $\Delta_{ij}^{\text{Isomap}}$ between x_i and x_j to be the shortest path in our graph from i to j:

assume connected graph.

$$\Delta_{ij}^{\text{Isomap}} = \min_{\substack{\text{paths } P \subseteq E \\ \text{from } i \text{ to } j}} \sum_{e \in P} w_e$$

$\hookrightarrow \Delta$ find $\rightarrow B = XX^T \rightarrow$ do usual PCA.



9

Isomap Algorithm

- Build graph.
- Compute shortest path distances.
- Perform MDS on shortest path distances to obtain visualization.

Laplacian Eigenmaps

Another slightly more optimization-theoretic viewpoint.

- Suppose that we built our graph, and all that we want is to to find a 1D embedding (embedding on to a line) so that points that are close in the graph are also close in the embedding.
- For each data point, we associate a real number $f_1, \ldots, f_n$, and try to minimize:

$$\min_{f_1,\ldots,f_n} \sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}(f_i - f_j)^2.$$

if i & j are conn. then $W_{ij} = 1$. & I want $(f_i - f_j)^2$ to be small.

Seems sensible but has a trivial solution.

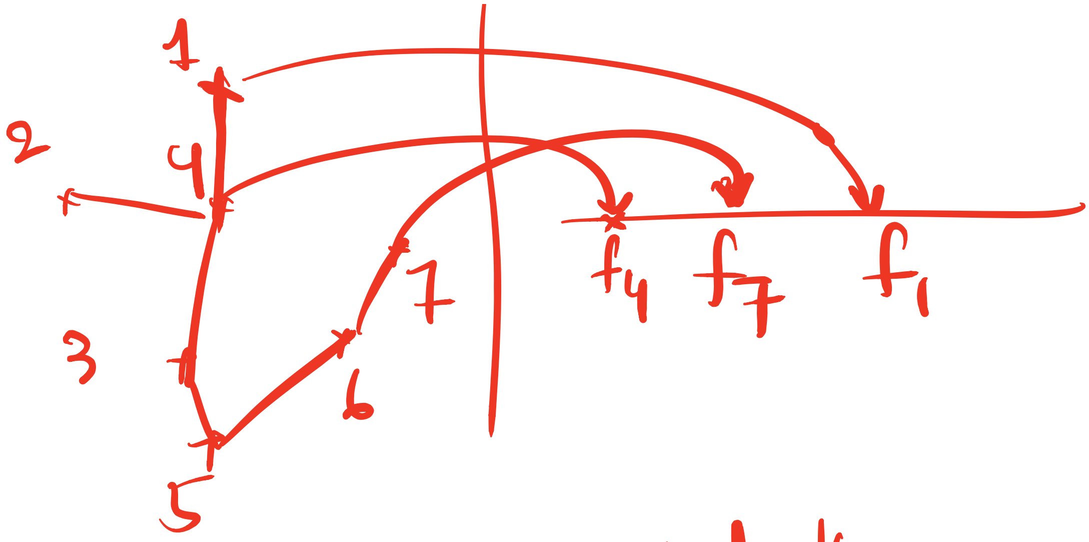
- Add some constraints:

$$\min_{f_1,\ldots,f_n} \sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}(f_i - f_j)^2,$$

$$\text{subject to } \sum_{i=1}^{n} f_i^2 = 1, \sum_{i=1}^{n} f_i = 0.$$

(ie) have mean 0 & variance 1.

& pts should be spread out



If 1 & 4 are connected then I want f_1 & f_4 to be close

$$\min \underline{f^T L f} \quad \Big] $$
$$\|f\|_2 = 1, \quad \underline{f^T 1 = 0.}$$

↳ solution is just compute second smallest EV of L.

Laplacian Eigenmaps

We know the solution to this problem:

$$\min_{f_1,\ldots,f_n} \sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij}(f_i - f_j)^2,$$

$$\text{subject to} \quad \sum_{i=1}^{n} f_i^2 = 1, \sum_{i=1}^{n} f_i = 0.$$

Simply embed according to the second smallest eigenvector of the Laplacian.

↳ compute spectral embedding.

More generally, we can use the spectral embedding from spectral clustering for dimension reduction. This is called Laplacian Eigenmaps.

Laplacian Eigenmaps: Algorithm

- Build graph.
- Find Laplacian, compute bottom k eigenvectors $V \in \mathbb{R}^{n\times k}$.
- Visualize using V.

The same reason why V helps us in clustering applies here, it helps us in visualization by non-linearly preserving clusters.