DSC 140B
Representation Learning

Lecture 15 | Part 1

Solving the Optimization Problem

A New Formulation

▶ **Given**: an $n \times n$ similarity matrix W

▶ **Compute**: embedding vector $\vec{f}$ **minimizing**

$$\text{Cost}(\vec{f}) = \frac{1}{2}\vec{f}^{T}L\vec{f}$$

subject to $\|\vec{f}\| = 1$ and $\vec{f} \perp (1, 1, \dots, 1)^{T}$

▶ This might sound familiar...

Recall: PCA

▶ **Given**: a $d \times d$ covariance matrix C

▶ **Find**: vector $\vec{u}$ **maximizing** the variance in the direction of $\vec{u}$:
$$\vec{u}^T C \vec{u}$$
subject to $\|\vec{u}\| = 1$.

▶ **Solution**: take $\vec{u} =$ top eigenvector of C

A New Formulation

▶ Forget about orthogonality constraint for now.

▶ **Compute**: embedding vector $\vec{f}$ **minimizing**

$$\text{Cost}(\vec{f}) = \frac{1}{2}\vec{f}^T L \vec{f}$$

subject to $\|\vec{f}\| = 1$.

▶ **Solution**: the *bottom* eigenvector of L.
  ▶ That is, eigenvector with smallest eigenvalue.

Claim

▶ The bottom eigenvector is $\vec{f} = \frac{1}{\sqrt{n}}(1, 1, \ldots, 1)^T$

▶ It has associated eigenvalue of 0.

▶ That is, $L\vec{f} = 0\vec{f} = \vec{0}$

Spectral[1] Theorem

Theorem

If A is a symmetric matrix, eigenvectors of A with distinct eigenvalues are orthogonal to one another.

The Fix

▶ Remember: we wanted $\vec{f}$ to be orthogonal to $\frac{1}{\sqrt{n}}(1, 1, \ldots, 1)^T$.

  ▶ i.e., should be orthogonal to bottom eigenvector of L.

▶ Fix: take $\vec{f}$ to the be eigenvector of L with with smallest eigenvalue $\neq 0$.

▶ Will be $\perp \frac{1}{\sqrt{n}}(1, 1, \ldots, 1)^T$ by the **spectral theorem**.

Spectral Embeddings: Problem

▶ **Given**: **similarity graph** with n nodes

▶ **Compute**: an **embedding** of the n points into $\mathbb{R}^1$ so that similar objects are placed nearby

▶ **Formally**: find embedding vector $\vec{f}$ **minimizing**

$$\min_{\vec{f}} \; \text{Cost}(\vec{f}) = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij}(f_i - f_j)^2 = \frac{1}{2}\vec{f}^T L \vec{f}$$

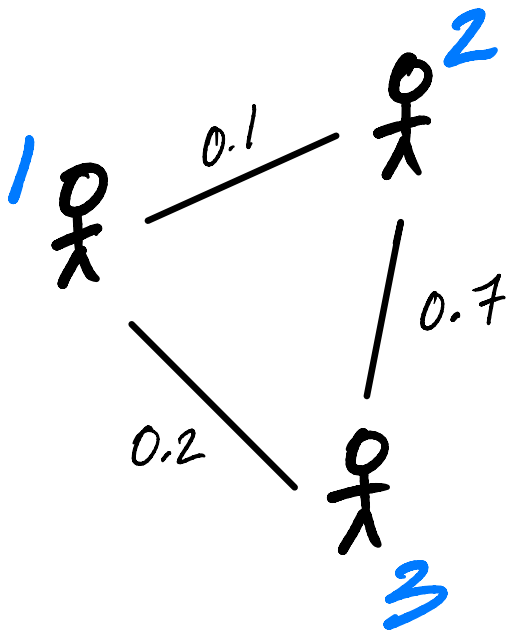subject to $\|\vec{f}\| = 1$ and $\vec{f} \perp (1, 1, \ldots, 1)^T$

Spectral Embeddings: Solution

▶ Form the **graph Laplacian** matrix, $L = D - W$

▶ Choose $\vec{f}$ be an eigenvector of L with smallest eigenvalue > 0

▶ This is the embedding!

Example


```python
W = np.array([
    [1, 0.1, 0.2],
    [0.1, 1, 0.7],
    [0.2, 0.7, 1]
])

D = np.diag(W.sum(axis=1))
L = D - W

vals, vecs = np.linalg.eigh(L)
f = vecs[:,1]
```
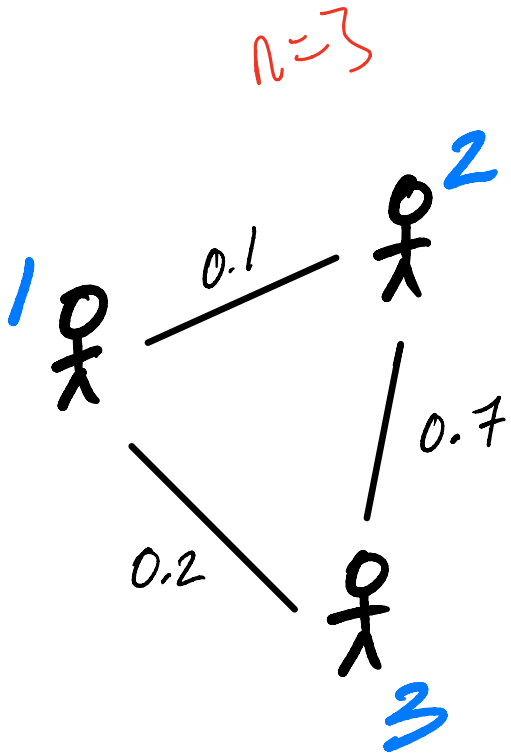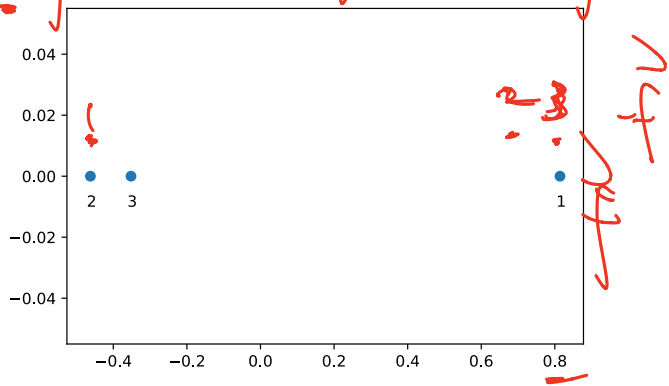
(handwritten annotation, crossed out) vecs[:,0]

Example

$n = 3$

$\vec{f} = (0.8, -0.45, -0.35) \in \mathbb{R}^3$

$L\vec{f} = \lambda \vec{f}$

$L(-\vec{f}) = \lambda(-\vec{f})$

$\lambda \vec{f}$

$L \cdot (3\vec{f}) = \lambda(3\vec{f})$

Embedding into $\mathbb{R}^k$

▶ This embeds nodes into $\mathbb{R}^1$.

▶ What about embedding into $\mathbb{R}^k$? $k \leq d.$

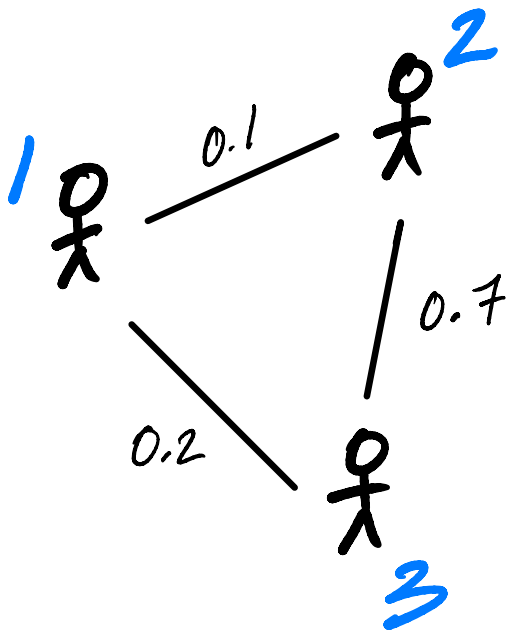▶ Natural extension: find bottom k eigenvectors with eigenvalues > 0

New Coordinates

▶ With k eigenvectors $\vec{f}^{(1)}, \vec{f}^{(2)}, \ldots, \vec{f}^{(k)}$, each node is mapped to a point in $\mathbb{R}^k$.

$$\text{Embedding}(i) = \begin{pmatrix} \vec{f}_i^{(1)}, & \vec{f}_i^{(2)}, & \cdots \\ & & \vec{f}_i^{(k)} \end{pmatrix}$$

▶ Consider node i.
  ▶ First new coordinate is $\vec{f}_i^{(1)}$.
  ▶ Second new coordinate is $\vec{f}_i^{(2)}$.
  ▶ Third new coordinate is $\vec{f}_i^{(3)}$.
  ▶ ⋮

Example


```python
W = np.array([
    [1, 0.1, 0.2],
    [0.1, 1, 0.7],
    [0.2, 0.7, 1]
])

D = np.diag(W.sum(axis=1))
L = D - W

vals, vecs = np.linalg.eigh(L)

# take two eigenvectors
# to map to R^2
f = vecs[:,1:3]
```
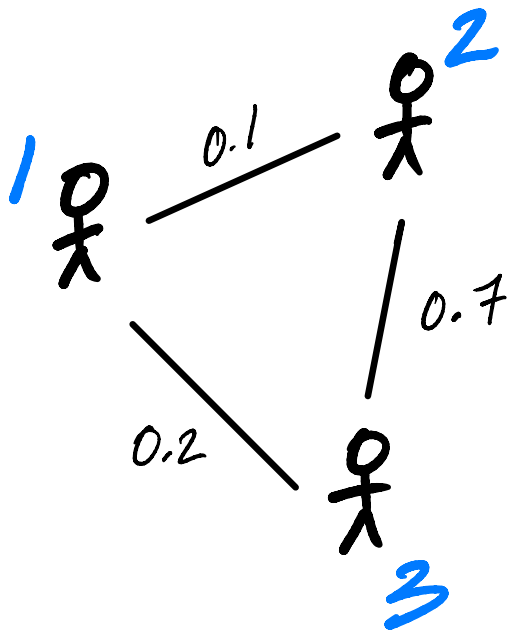
$n \times 2$

Example
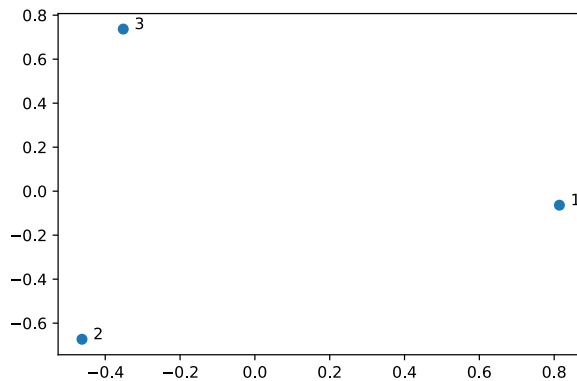
Laplacian Eigenmaps

▶ This approach is part of the method of "**Laplacian eigenmaps**"

▶ Introduced by Mikhail Belkin[2] and Partha Niyogi

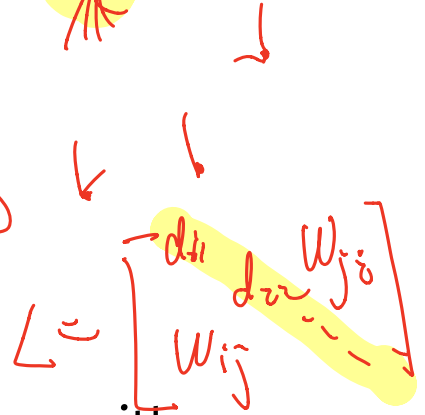▶ It is a type of **spectral embedding**

---

[2]Now at HDSI

A Practical Issue

▶ The Laplacian is often **normalized**:

$$L_{\mathrm{norm}} = D^{-1/2} L D^{-1/2}$$

where $D^{-1/2}$ is the diagonal matrix whose ith diagonal entry is $1/\sqrt{d_{ii}}$.

▶ Proceed by finding the eigenvectors of L_{norm}.

In Summary

▶ We can **embed** a similarity graph's nodes into $\mathbb{R}^k$ using the eigenvectors of the graph Laplacian

▶ Yet another instance where eigenvectors are solution to optimization problem

▶ Next time: using this for dimensionality reduction

DSC 140B
Representation Learning

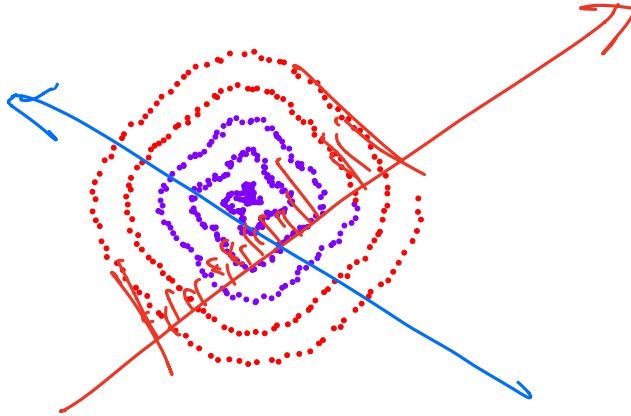Lecture 15 | Part 2

Nonlinear Dimensionality Reduction

Scenario

- ▶ You want to train a classifier on this data.

- ▶ It would be easier if we could "unroll" the spiral.

- ▶ Data seems to be one-dimensional, even though in two dimensions.

- ▶ Dimensionality reduction?
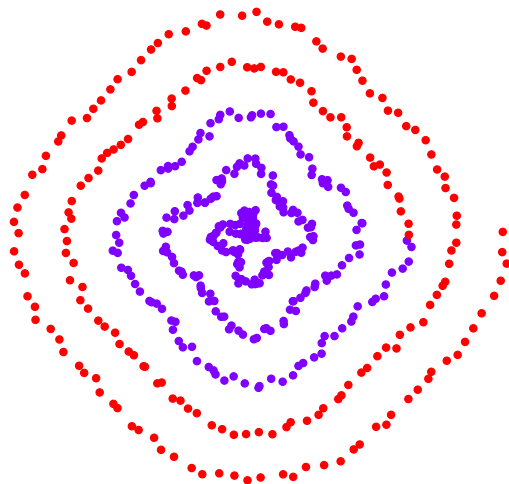
PCA?

▶ Does PCA work here?

▶ Try projecting onto one principal component.

No

PCA?

▶ PCA simply "rotates" the data.

▶ No amount of rotation will "unroll" the spiral.

▶ We need a fundamentally different approach that works for non-linear patterns.

Today

► Non-linear dimensionality reduction via **spectral embeddings**.

Last Time: Spectral Embeddings

▶ **Given**: a similarity graph with n nodes, number of dimensions k.

▶ **Embed**: each node as a point in $\mathbb{R}^k$ such that similar nodes are mapped to nearby points

▶ **Solution**: *bottom k* non-constant eigenvectors of graph Laplacian

Idea

- Build a similarity graph from points.

- Points *near the spiral* should be similar.

- Embed the similarity graph into $\mathbb{R}^1$
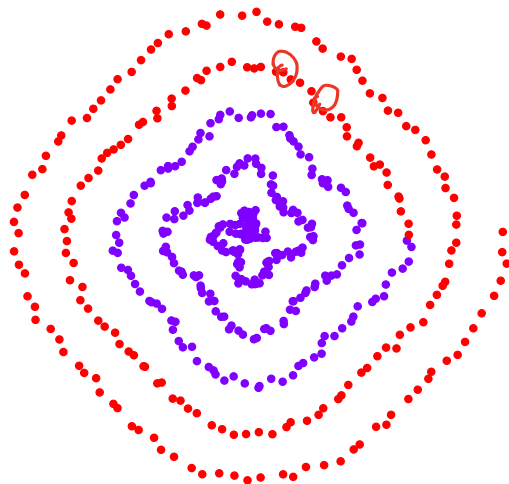
Today

▶ 1) How do we build a graph from a set of points?

▶ 2) Dimensionality reduction with Laplacian eigenmaps

DSC 140B
Representation Learning

Lecture 15 | Part 3

From Points to Graphs

Dimensionality Reduction

▶ **Given**: n points in $\mathbb{R}^d$, number of dimensions $k \leq d$

▶ **Map**: each point $\vec{x}$ to new representation $\vec{z} \in \mathbb{R}^k$

Idea

▶ Build a similarity graph from points in $\mathbb{R}^2$

▶ Use approach from last lecture to embed into $\mathbb{R}^k$

▶ But how do we represent a set of points as a similarity graph?

Why graphs?
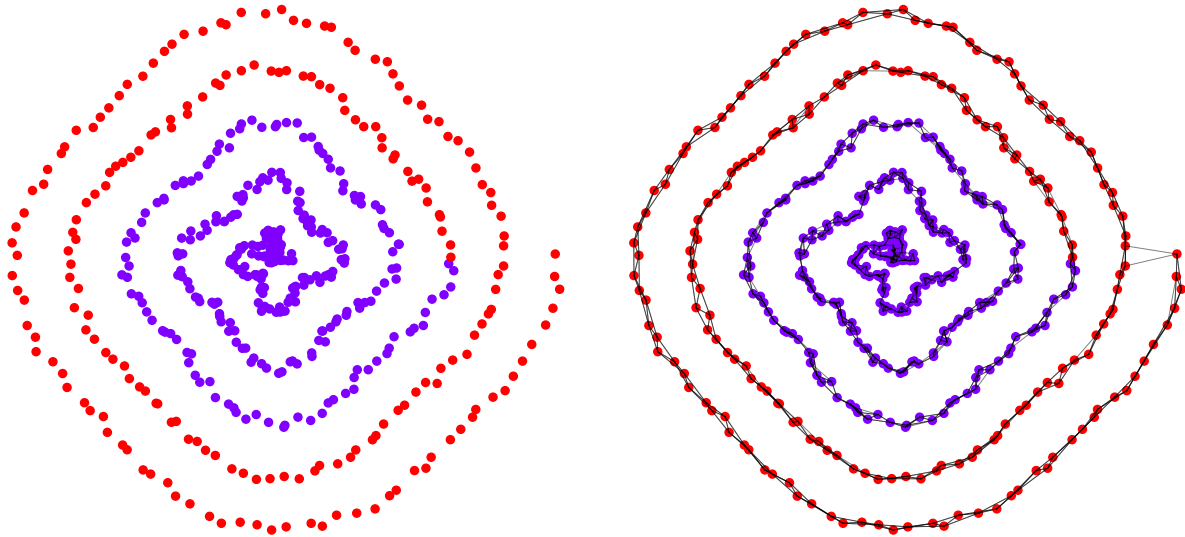
Three Approaches

▶ 1) Epsilon neighbors graph

▶ 2) k-Nearest neighbor graph

▶ 3) fully connected graph with similarity function

Epsilon Neighbors Graph

$d = 2$

▶ Input: vectors $\vec{x}^{(1)}, \ldots, \vec{x}^{(n)}$, a number ε

▶ Create a graph with one node i per point $\vec{x}^{(i)}$

▶ Add edge between nodes i and j if $\|\vec{x}^{(i)} - \vec{x}^{(j)}\| \le \varepsilon$

▶ Result: **unweighted** graph

$\vec{x}^{(i)}$
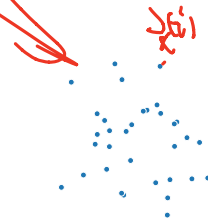
Exercise

What will the graph look like when ε is small? What about when it is large?
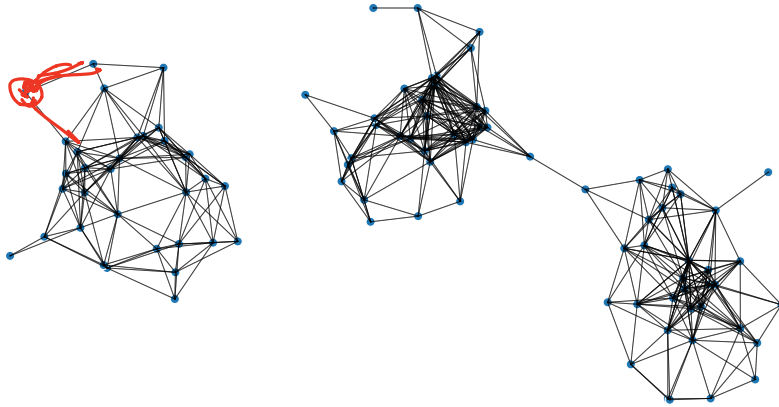
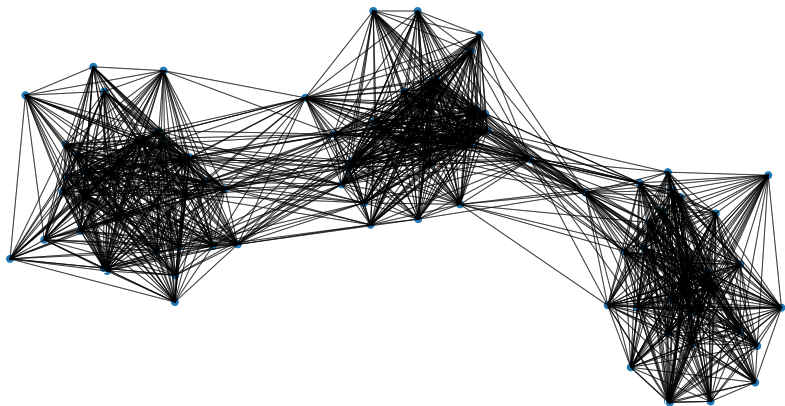# Epsilon Neighbors Graph
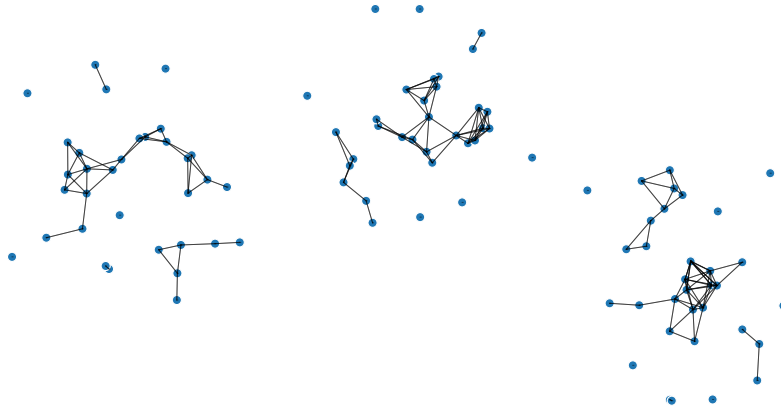
ε small

Epsilon Neighbors Graph
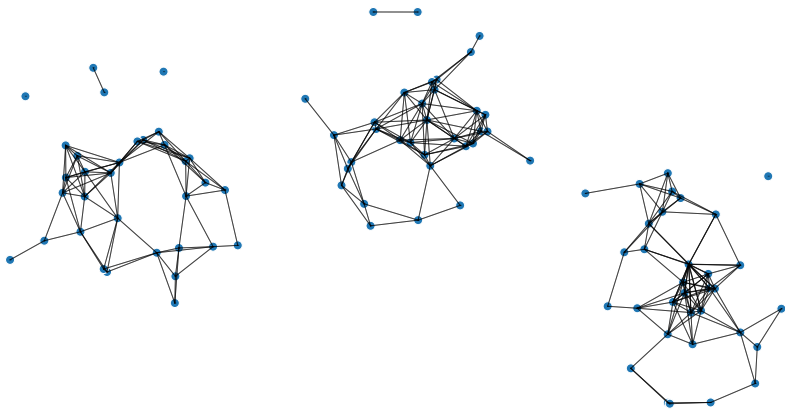
Epsilon Neighbors Graph

Epsilon Neighbors Graph

Note

▶ We've drawn these graphs by placing nodes at the same position as the point they represent

▶ But a graph's nodes can be drawn in any way

Epsilon Neighbors: Pseudocode

n x d

```python
# assume the data is in X
n = len(X)
adj = np.zeros_like(X)
for i in range(n):
    for j in range(n):
        if distance(X[i], X[j]) <= epsilon:
            adj[i, j] = 1
```

Picking ε

▶ If ε is too small, graph is underconnected

▶ If ε is too large, graph is overconnected

▶ If you cannot visualize, just try and see

With scikit-learn

```python
import sklearn.neighbors
adj = sklearn.neighbors.radius_neighbors_graph(
        X,
        radius=...
    )
```

k-Neighbors Graph

▶ Input: vectors $\vec{x}^{(1)}, \ldots, \vec{x}^{(n)}$, a number k

▶ Create a graph with one node i per point $\vec{x}^{(i)}$

▶ Add edge between each node i and its k closest neighbors

▶ Result: **unweighted** graph

k-Neighbors: Pseudocode

```python
# assume the data is in X
n = len(X)
adj = np.zeros_like(X)
for i in range(n):
    for j in k_closest_neighbors(X, i):
        adj[i, j] = 1
```

Exercise

Is it possible for a k-neighbors graph to be dis-conected?
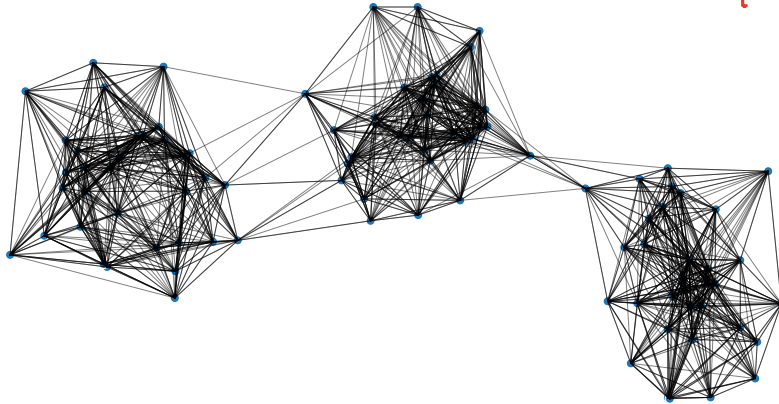
k-Neighbors Graph
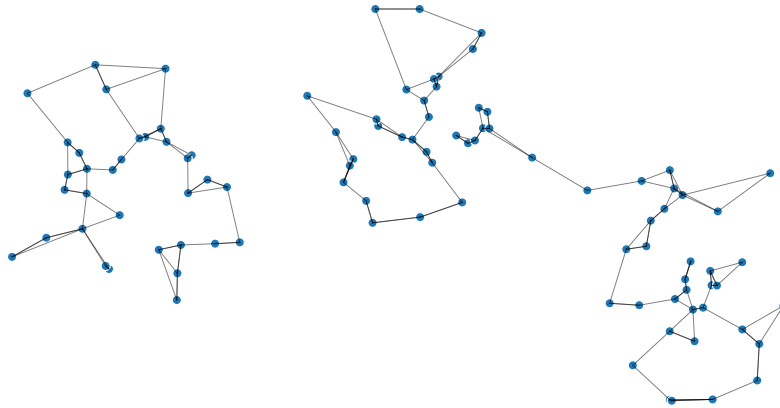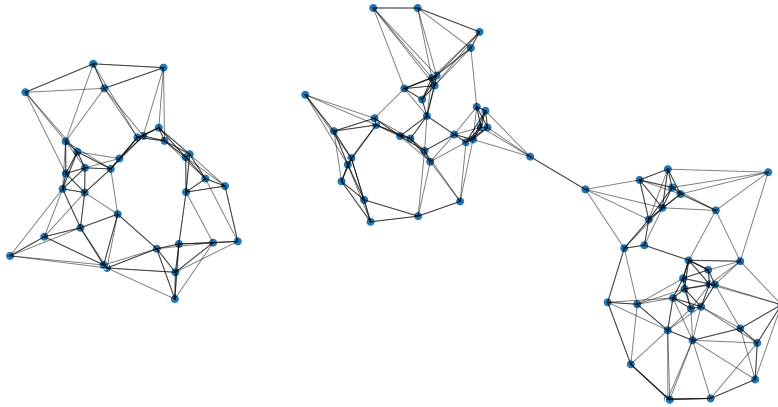
k-Neighbors Graph
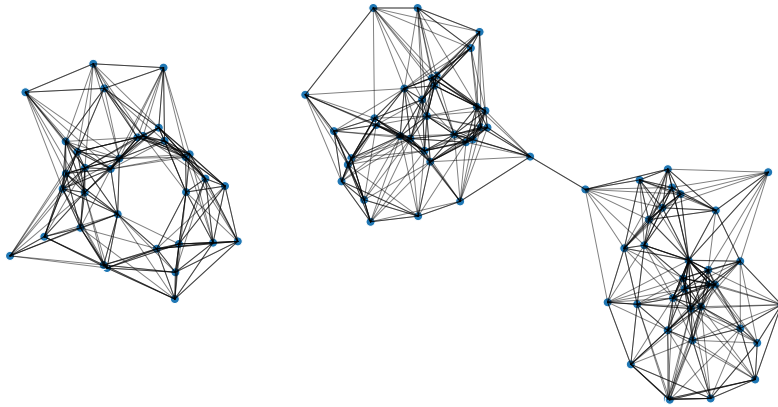
k-Neighbors Graph

k-Neighbors Graph

$k \geq n-1$

With scikit-learn

```python
import sklearn.neighbors
adj = sklearn.neighbors.kneighbors_graph(
        X,
        n_neighbors=...
    )
```

Fully Connected Graph

▶ Input: vectors $\vec{x}^{(1)}, \dots, \vec{x}^{(n)}$, a similarity function h

▶ Create a graph with one node i per point $\vec{x}^{(i)}$

▶ Add edge between every pair of nodes. Assign weight of $h(\vec{x}^{(i)}, \vec{x}^{(j)})$
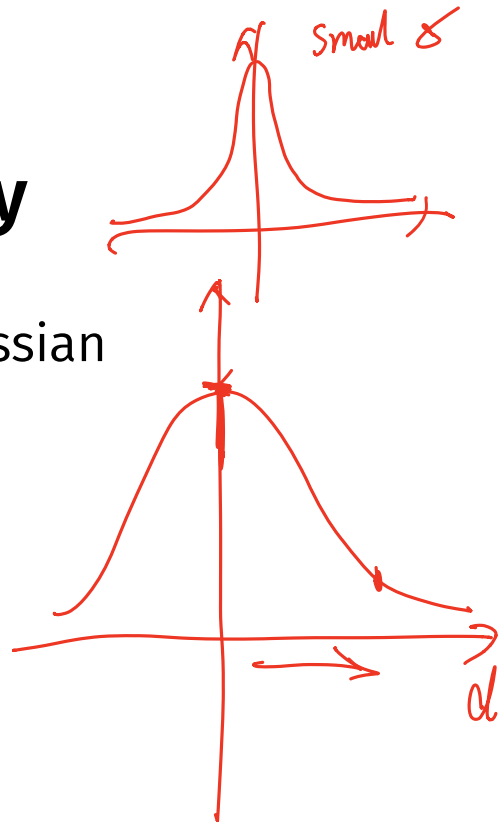
▶ Result: **weighted** graph

← difference

Gaussian Similarity

► A common similarity function: Gaussian

► Must choose σ appropriately
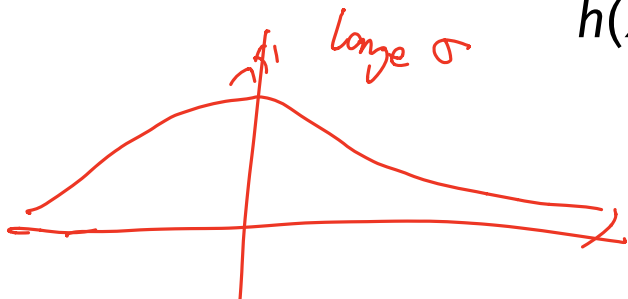
$$h(\vec{x}, \vec{y}) = e^{-\|\vec{x} - \vec{y}\|^2 / \sigma^2}$$

Small σ

Large σ

d_r

d

Fully Connected: Pseudocode

```python
def h(x, y):
    dist = np.linalg.norm(x, y)
    return np.exp(-dist**2 / sigma**2)

# assume the data is in X
n = len(X)
w = np.ones_like(X)   ones((n,n))
for i in range(n):
    for j in range(n):
        w[i, j] = h(X[i], X[j])
```
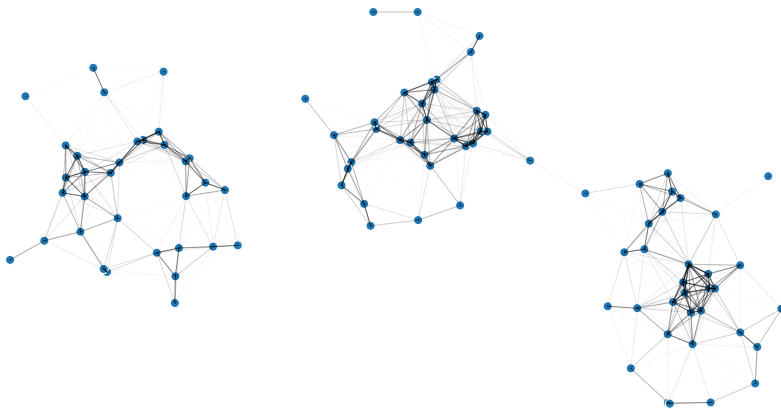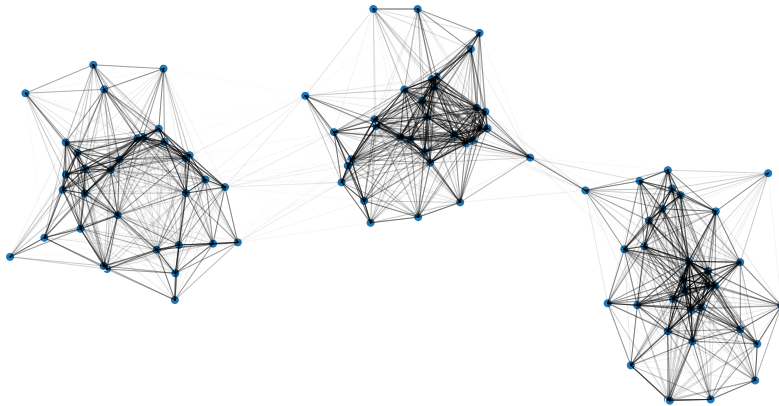
$x : n \times d$

$w : n \times n$

With SciPy

```
distances = scipy.spatial.distance_matrix(X, X)
w = np.exp(-distances**2 / sigma**2)
```
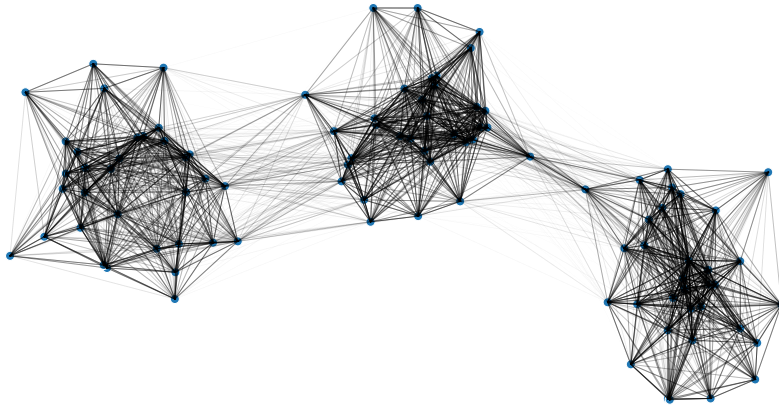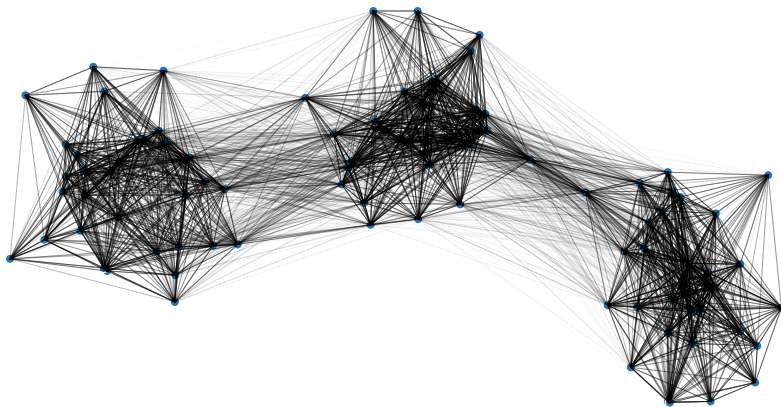
Gaussian Similarity

Gaussian Similarity

Gaussian Similarity

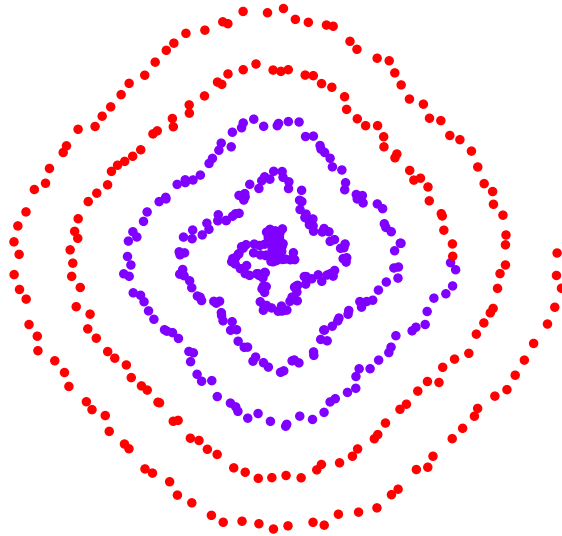Gaussian Similarity

DSC 140B
Representation Learning

Lecture 15 | Part 4

Laplacian Eigenmaps

Idea

▶ Build a similarity graph from points in $\mathbb{R}^2$
  ▶ epsilon neighbors, k-neighbors, or fully connected

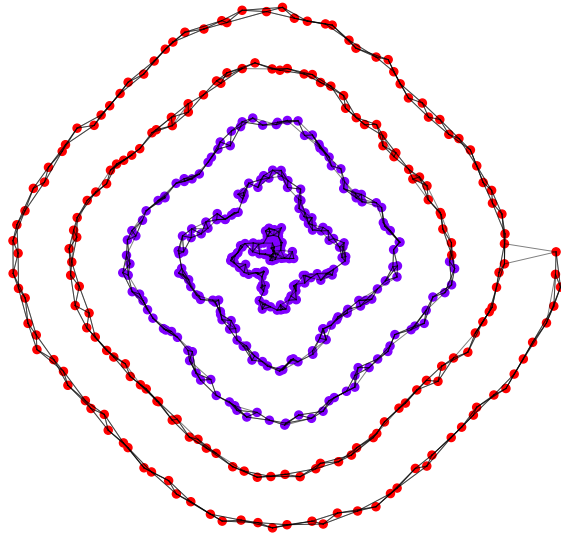▶ Now: use approach from last lecture to embed into $\mathbb{R}^k$

Example 1: Spiral

Example 1: Spiral

$k = 4$

▶ Build a *k*-neighbors graph.
▶ Note: follows the 1-d shape of the data.

Example 1: Spectral Embedding

▶ Let W be the weight matrix (k-neighbor adjacency matrix)

▶ Compute $L = D - W$

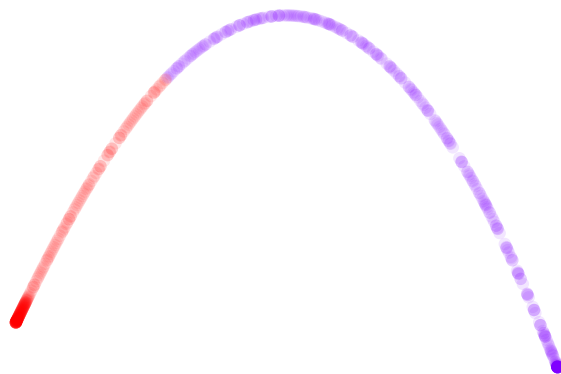▶ Compute bottom k non-constant eigenvectors of L, use as embedding

Example 1: Spiral
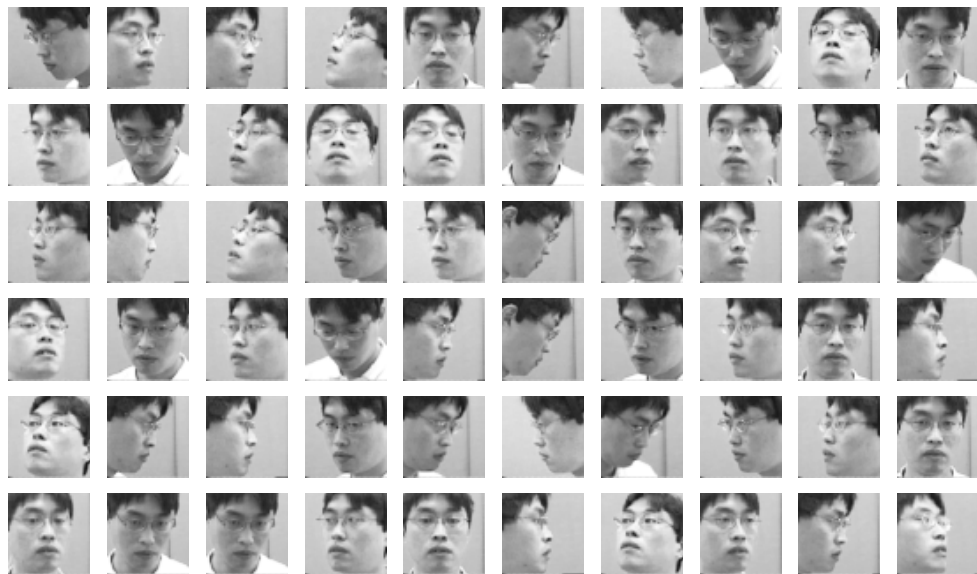
▶ Embedding into $\mathbb{R}^1$

Example 1: Spiral

▶ Embedding into $\mathbb{R}^2$

Example 1: Spiral

```python
import sklearn.neighbors
import sklearn.manifold
W = sklearn.neighbors.kneighbors_graph(
        X, n_neighbors=4
)
embedding = sklearn.manifold.spectral_embedding(
        W, n_components=2
)
```
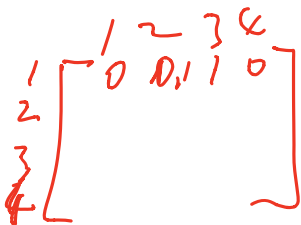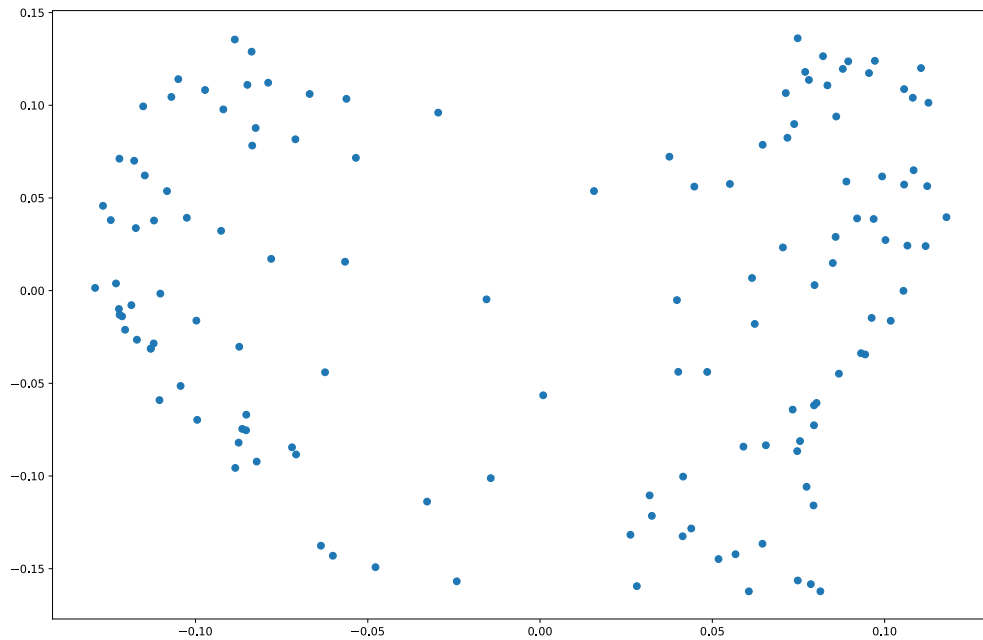
Example 2: Face Pose

Example 2: Face Pose

▶ Construct fully-connected similarity graph with Gaussian similarity

▶ Embed with Laplacian eigenmaps

Example 2: Face Pose

Example 2: Face Pose

DSC 140B
Representation Learning

Lecture 15 | Part 5

Spectral Clustering

Spectral Embeddings

► Useful in multiple tasks:
   ► Feature learning before classification
   ► Visualizing high dimensional data
   ► Clustering

Spectral Clustering

► Problem: k-means assumptions:
  ► Data are vectors (what about graphs?)
  ► Clusters are spherical (what about more complex patterns?)

► One idea:
  1. Embed using, e.g., Laplacian eigenmaps
  2. Run k-means on the embedded points

Demo