

DSC 140B

Representation Learning

Lecture 09 | Part 1

Embedding Similarities

Similar Netflix Users

- ▶ Suppose you are a data scientist at Netflix
- ▶ You're given an $n \times n$ **similarity matrix** W of users
 - ▶ entry (i, j) tells you how *similar* user i and user j are
 - ▶ 1 means “very similar”, 0 means “not at all”
- ▶ Goal: visualize to find patterns

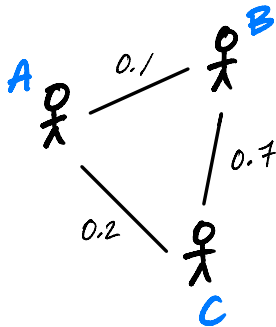
Idea

- ▶ We like scatter plots. Can we make one?
- ▶ Users are **not** vectors / points!
- ▶ They are nodes in a **similarity graph**

Similarity Graphs

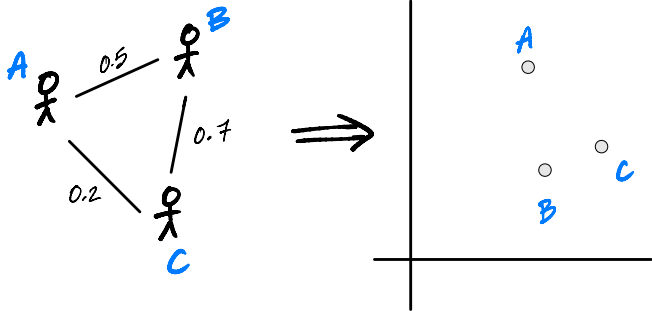
- Similarity matrices can be thought of as weighted graphs, and vice versa.

$$\begin{matrix} & \begin{matrix} A & B & C \end{matrix} \\ \begin{matrix} A \\ B \\ C \end{matrix} & \begin{pmatrix} 1 & 0.1 & 0.2 \\ 0.1 & 1 & 0.7 \\ 0.2 & 0.7 & 1 \end{pmatrix} \end{matrix}$$



Goal

- ▶ **Embed** nodes of a similarity graph as points.
- ▶ Similar nodes should map to nearby points.



Today

- ▶ We will design a graph embedding approach:
 - ▶ **Spectral embeddings** via **Laplacian eigenmaps**

More Formally

- ▶ **Given:**
 - ▶ A **similarity graph** with n nodes
 - ▶ a number of dimensions, k
- ▶ **Compute:** an **embedding** of the n points into \mathbb{R}^k so that similar objects are placed nearby

To Start

- ▶ **Given:**
 - ▶ A **similarity graph** with n nodes
- ▶ **Compute:** an **embedding** of the n points into \mathbb{R}^1 so that similar objects are placed nearby

Vectors as Embeddings into \mathbb{R}^1

- ▶ Suppose we have n nodes (objects) to embed
- ▶ Assume they are numbered $1, 2, \dots, n$
- ▶ Let $f_1, f_2, \dots, f_n \in \mathbb{R}$ be the embeddings
- ▶ We can pack them all into a vector: \vec{f} .
- ▶ Goal: find a good set of embeddings, \vec{f} .

Example

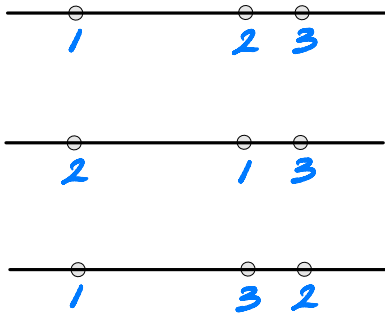
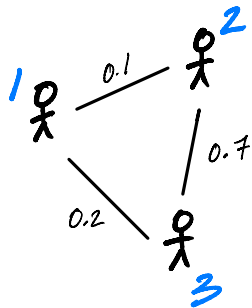
$$\vec{f} = (1, 3, 2, -4)^T$$

An Optimization Problem

- ▶ We'll turn it into an optimization problem:
- ▶ **Step 1:** Design a cost function quantifying how good a particular embedding \vec{f} is
- ▶ **Step 2:** Minimize the cost

Example

- Which is the best embedding?



Cost Function for Embeddings

- ▶ Idea: cost is low if similar points are close
- ▶ Here is one approach:

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

- ▶ where w_{ij} is the weight between i and j .

Interpreting the Cost

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

- ▶ If $w_{ij} \approx 0$, that pair can be placed very far apart without increasing cost
- ▶ If $w_{ij} \approx 1$, the pair should be placed close together in order to have small cost.

Exercise

Do you see a problem with the cost function?

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

Hint: what embedding \vec{f} minimizes it?

Problem

- ▶ The cost is **always** minimized by taking $\vec{f} = 0$.
- ▶ This is a “**trivial**” solution. Not useful.
- ▶ **Fix:** require $\|\vec{f}\| = 1$
 - ▶ Really, any number would work. 1 is convenient.

Exercise

Do you see **another** problem with the cost function, even if we require \vec{f} to be a unit vector?

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

Hint: what other choice of \vec{f} will **always** make this zero?

Problem

- ▶ The cost is **always** minimized by taking $\vec{f} = \frac{1}{\sqrt{n}}(1, 1, \dots, 1)^T$.
- ▶ This is a “trivial” solution. Again, not useful.
- ▶ **Fix:** require \vec{f} to be orthogonal to $(1, 1, \dots, 1)^T$.
 - ▶ Written: $\vec{f} \perp (1, 1, \dots, 1)^T$
 - ▶ Ensures that solution is not close to trivial solution
 - ▶ Might seem strange, but it will work!

The New Optimization Problem

- ▶ **Given:** an $n \times n$ similarity matrix W
- ▶ **Compute:** embedding vector \vec{f} minimizing

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

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

How?

- ▶ This looks difficult.
- ▶ Let's write it in matrix form.
- ▶ We'll see that it is actually (hopefully) familiar.

DSC 140B

Representation Learning

Lecture 09 | Part 2

The Graph Laplacian

The Problem

- **Compute:** embedding vector \vec{f} minimizing

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

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

- Now: write the cost function as a matrix expression.

The Degree Matrix

- ▶ Recall: in an unweighted graph, the degree of node i equals number of neighbors.
- ▶ Equivalently (where A is the adjacency matrix):

$$\text{degree}(i) = \sum_{j=1}^n A_{ij}$$

- ▶ Since $A_{ij} = 1$ only if j is a neighbor of i

The Degree Matrix

- ▶ In a weighted graph, define **degree** of node i similarly:

$$\text{degree}(i) = \sum_{j=1}^n w_{ij}$$

- ▶ That is, it is the total weight of all neighbors.

The Degree Matrix

- ▶ The **degree matrix** D of a weighted graph is the diagonal matrix where entry (i, i) is given by:

$$\begin{aligned}d_{ii} &= \text{degree}(i) \\ &= \sum_{j=1}^n w_{ij}\end{aligned}$$

The Graph Laplacian

- ▶ Define $L = D - W$
 - ▶ D is the degree matrix
 - ▶ W is the similarity matrix (weighted adjacency)
- ▶ L is called the **Graph Laplacian** matrix.
- ▶ It is a very useful object

Very Important Fact

► Claim:

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

► Proof: expand both sides ¹

¹Note that there was originally a $\frac{1}{2}$ in front of $\vec{f}^T L \vec{f}$, but this was not correct as written. See Problem 06 in the Midterm 02 practice for a longer explanation.

Proof

DSC 140B

Representation Learning

Lecture 09 | Part 3

Solving the Optimization Problem

A New Formulation

- ▶ **Given:** an $n \times n$ similarity matrix W
- ▶ **Compute:** embedding vector \vec{f} **minimizing**

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

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**

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

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

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

Claim

- ▶ The bottom eigenvector is $\vec{f} = \frac{1}{\sqrt{n}}(1, 1, \dots, 1)^T$
- ▶ It has associated eigenvalue of 0.
- ▶ That is, $L\vec{f} = 0\vec{f} = \vec{0}$

Spectral² Theorem

Theorem

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

²“Spectral” not in the sense of specters (ghosts), but because the eigenvalues of a transformation form the “spectrum”

The Fix

- ▶ Remember: we wanted \vec{f} to be orthogonal to $\frac{1}{\sqrt{n}}(1, 1, \dots, 1)^T$.
 - ▶ i.e., should be orthogonal to bottom eigenvector of L .
- ▶ Fix: take \vec{f} to be eigenvector of L with with smallest eigenvalue $\neq 0$.
- ▶ Will be $\perp \frac{1}{\sqrt{n}}(1, 1, \dots, 1)^T$ 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**

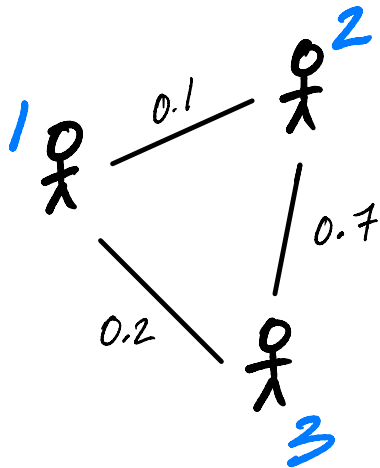
$$\text{Cost}(\vec{f}) = \sum_{i=1}^n \sum_{j=1}^n w_{ij} (f_i - f_j)^2 = \vec{f}^T L \vec{f}$$

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

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



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

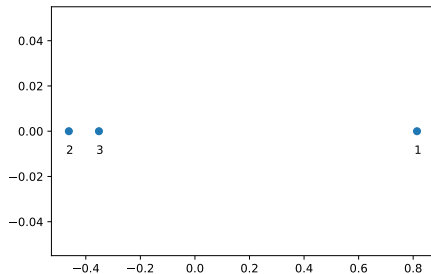
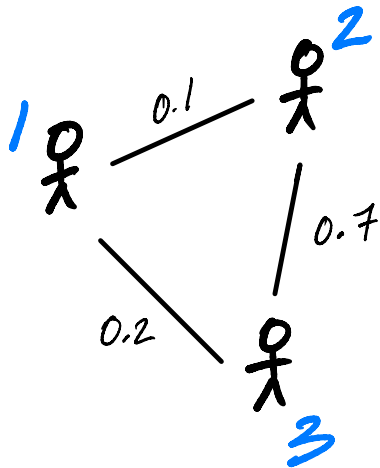
```
D = np.diag(W.sum(axis=1))
```

```
L = D - W
```

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

```
f = vecs[:,1]
```

Example



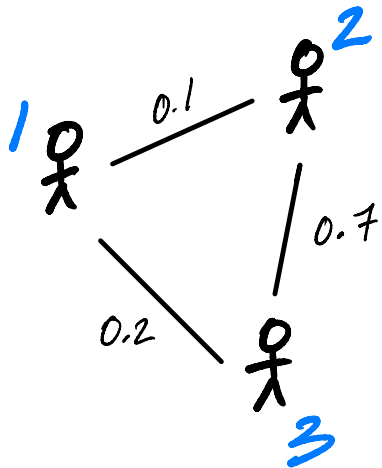
Embedding into \mathbb{R}^k

- ▶ This embeds nodes into \mathbb{R}^1 .
- ▶ What about embedding into \mathbb{R}^k ?
- ▶ Natural extension: find bottom k eigenvectors with eigenvalues > 0

New Coordinates

- ▶ With k eigenvectors $\vec{f}^{(1)}, \vec{f}^{(2)}, \dots, \vec{f}^{(k)}$, each node is mapped to a point in \mathbb{R}^k .
- ▶ Consider node i .
 - ▶ First new coordinate is $\vec{f}_i^{(1)}$.
 - ▶ Second new coordinate is $\vec{f}_i^{(2)}$.
 - ▶ Third new coordinate is $\vec{f}_i^{(3)}$.
 - ▶ \vdots

Example



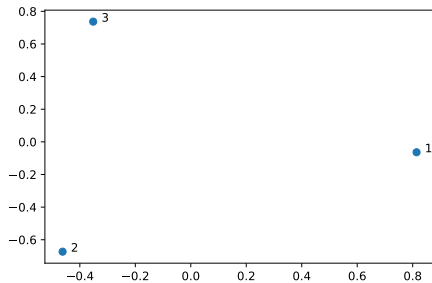
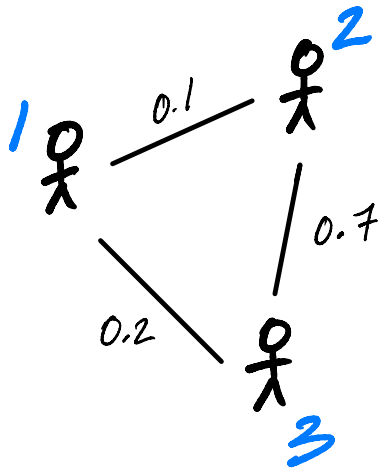
```
W = np.array([
    [1, 0.1, 0.2],
    [0.1, 1, 0.7],
    [0.2, 0.7, 1]
])
```

```
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]
```

Example



Laplacian Eigenmaps

- ▶ This approach is part of the method of “**Laplacian eigenmaps**”
- ▶ Introduced by Mikhail Belkin³ and Partha Niyogi
- ▶ It is a type of **spectral embedding**

³Now at HDSI

A Practical Issue

- ▶ The Laplacian is often **normalized**:

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

where $D^{-1/2}$ is the diagonal matrix whose i th 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

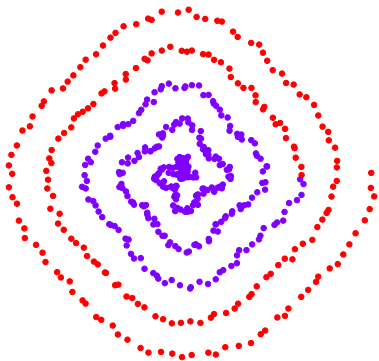
Representation Learning

Lecture 09 | Part 4

Nonlinear Dimensionality Reduction

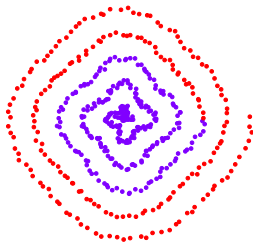
Scenario

- ▶ 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

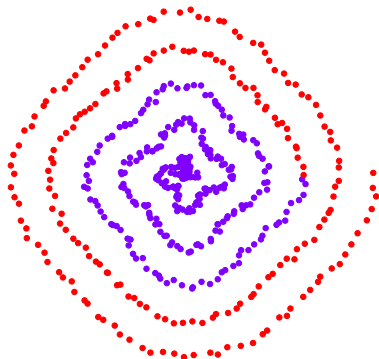
- ▶ Non-linear dimensionality reduction via **spectral embeddings**.

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 09 | Part 5

From Points to Graphs

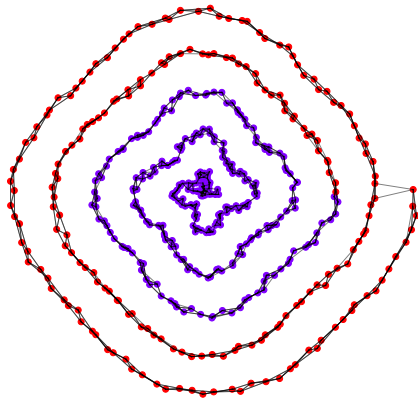
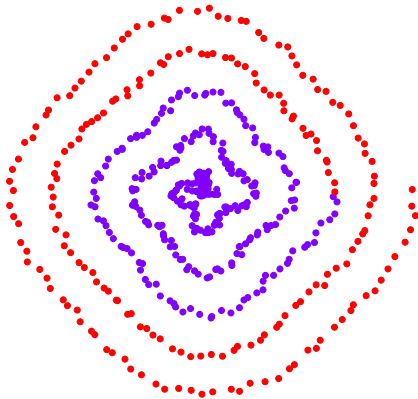
Dimensionality Reduction

- ▶ **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

- ▶ Input: vectors $\vec{x}^{(1)}, \dots, \vec{x}^{(n)}$,
a number ε
- ▶ Create a graph with one
node i per point $\vec{x}^{(i)}$
- ▶ Add edge between nodes i
and j if $\|\vec{x}^{(i)} - \vec{x}^{(j)}\| \leq \varepsilon$
- ▶ Result: **unweighted** graph

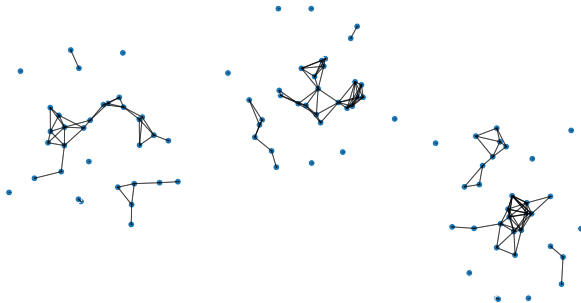


Exercise

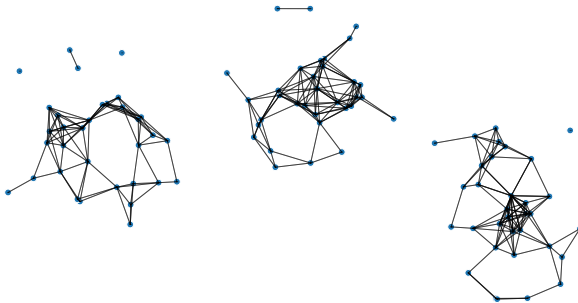
What will the graph look like when ϵ is small? What about when it is large?



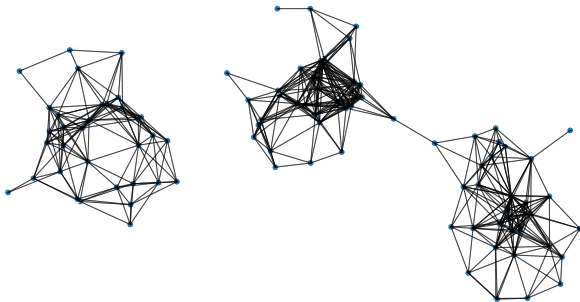
Epsilon Neighbors Graph



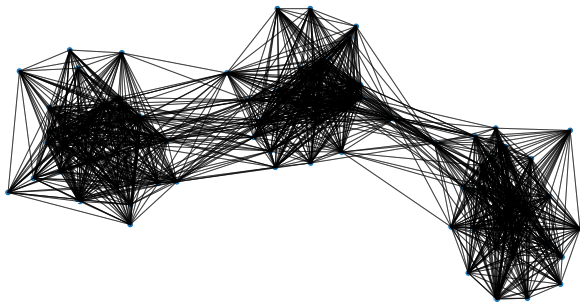
Epsilon Neighbors Graph



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

```
# 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

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

k-Neighbors Graph

- ▶ Input: vectors $\vec{x}^{(1)}, \dots, \vec{x}^{(n)}$, 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



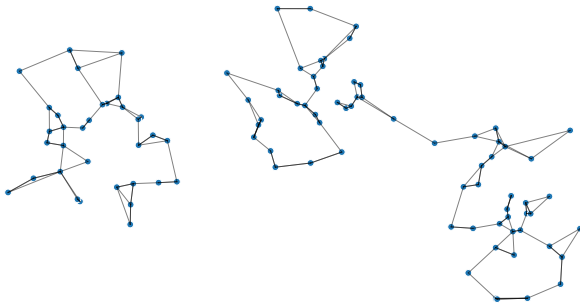
k-Neighbors: Pseudocode

```
# 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
```

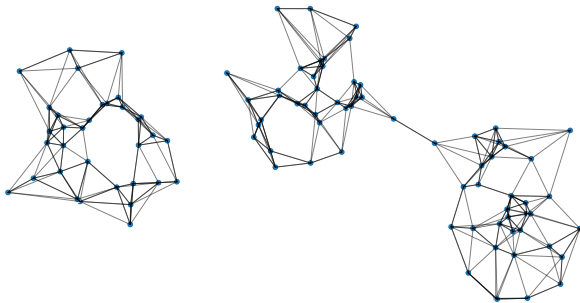

Exercise

Is it possible for a k -neighbors graph to be disconnected?

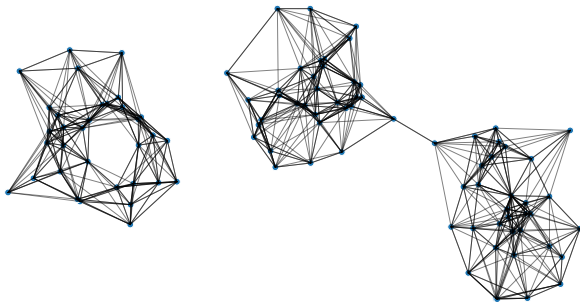
k-Neighbors Graph



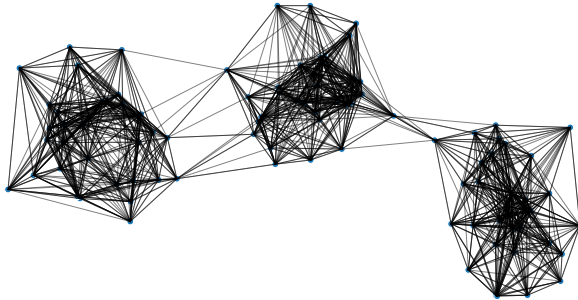
k-Neighbors Graph



k-Neighbors Graph



k-Neighbors Graph



With scikit-learn

```
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



Gaussian Similarity

- ▶ A common similarity function: Gaussian
- ▶ Must choose σ appropriately

$$h(\vec{x}, \vec{y}) = e^{-\|\vec{x}-\vec{y}\|^2/\sigma^2}$$

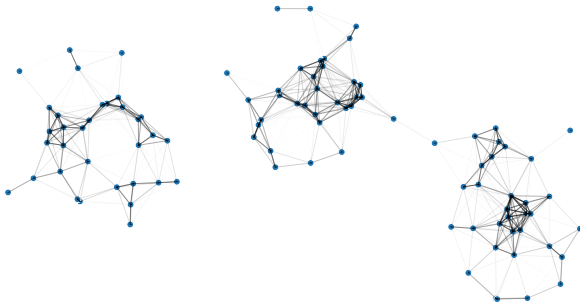
Fully Connected: Pseudocode

```
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)  
for i in range(n):  
    for j in range(n):  
        w[i, j] = h(X[i], X[j])
```

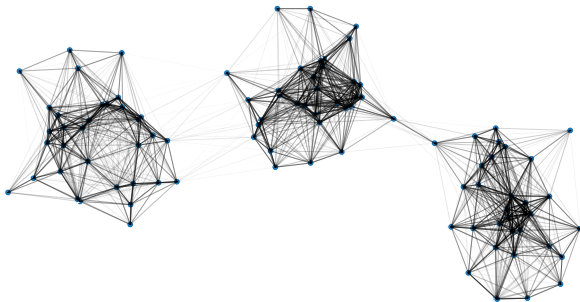
With SciPy

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

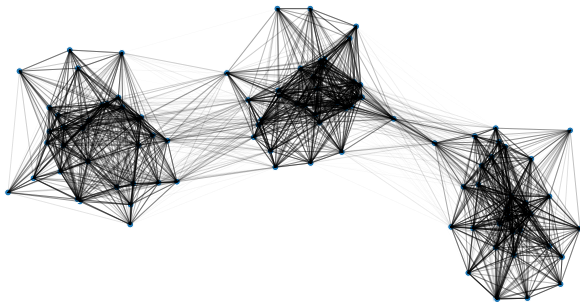
Gaussian Similarity



Gaussian Similarity



Gaussian Similarity



Gaussian Similarity

