

# DSC 140B

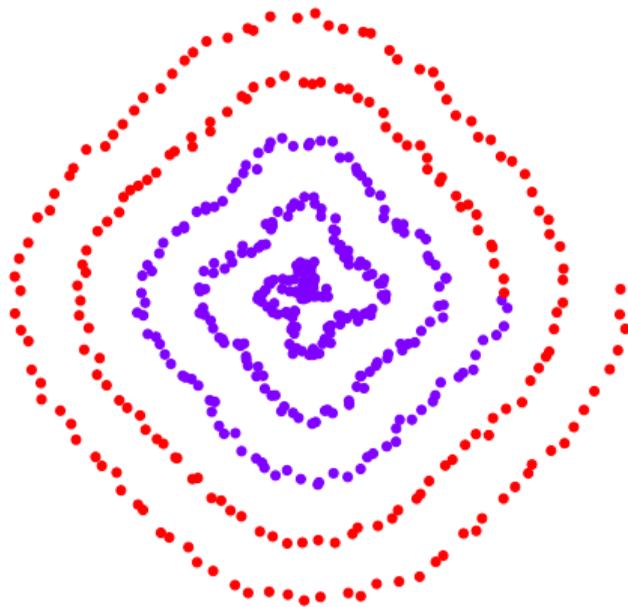
## Representation Learning

Lecture 08 | Part 1

**Recap**

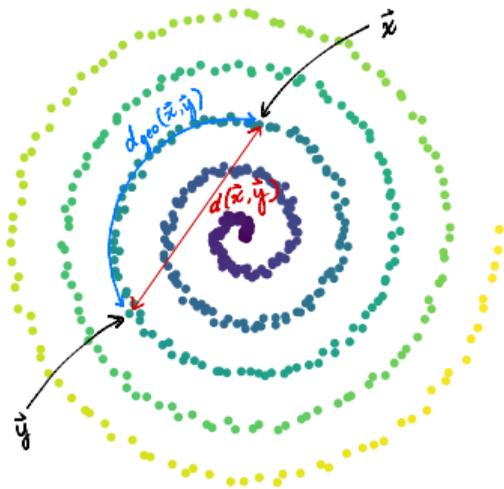
# Last Time

- ▶ PCA is a **linear** dimension reduction technique.
  - ▶ Works well when data lies near a **linear** subspace.
- ▶ What if data lies on a **nonlinear** manifold?



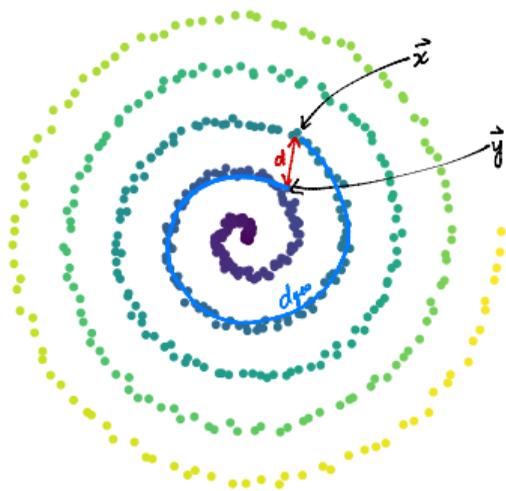
# Euclidean vs. Geodesic Distance

- ▶ **Euclidean distance:** the “straight-line” distance
- ▶ **Geodesic distance:** the distance along the manifold



# Euclidean vs. Geodesic Distance

- ▶ **Euclidean distance:** the “straight-line” distance
- ▶ **Geodesic distance:** the distance along the manifold

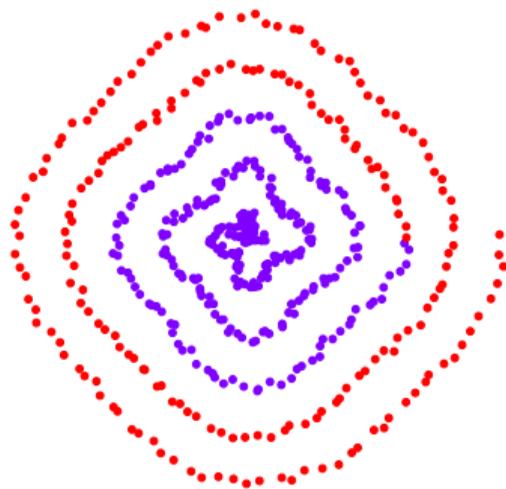


# Non-Linear Dimensionality Reduction

- ▶ **Goal:** Map points in  $\mathbb{R}^d$  to  $\mathbb{R}^k$
- ▶ **Such that:** if  $\vec{x}$  and  $\vec{y}$  are close in **geodesic** distance in  $\mathbb{R}^d$ , they are close in **Euclidean** distance in  $\mathbb{R}^k$

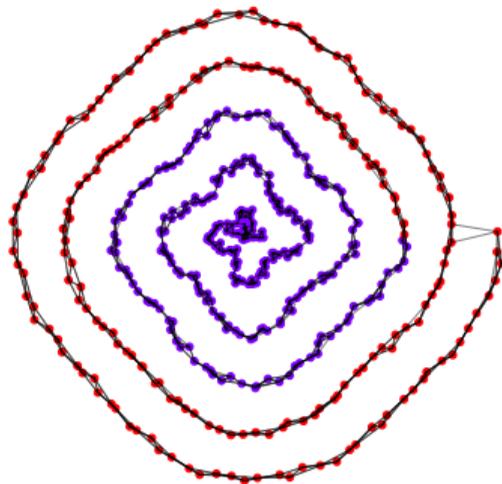
# Computing Geodesic Distance

- ▶ But how do we **compute** geodesic distance?



# Computing Geodesic Distance

- ▶ But how do we **compute** geodesic distance?
- ▶ **Idea:** build a **similarity graph**.
- ▶ Now, geodesic distance  $\approx$  shortest path distance on graph.



# The Plan

- ▶ **Before:** Map points close in geodesic distance to points close in  $\mathbb{R}^k$ .
- ▶ **Now:**
  - ▶ Convert points to a similarity graph.
  - ▶ Map similar nodes to points close in  $\mathbb{R}^k$ .

# DSC 140B

## Representation Learning

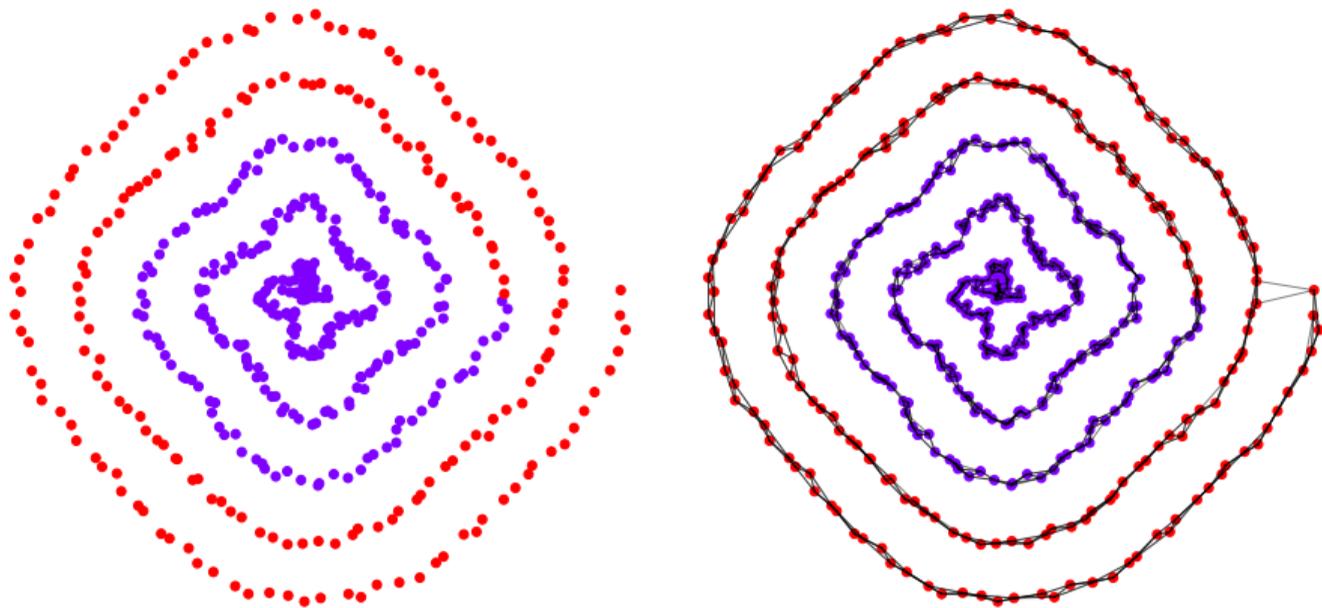
Lecture 08 | Part 2

**From Points to Graphs**

# Goal

- ▶ **Given:** a set of  $n$  points in  $\mathbb{R}^d$ 
  - ▶ We think they're on a (non-linear) manifold
- ▶ **Goal:** construct a **similarity graph** where:
  - ▶ Each point corresponds to a node
  - ▶ Edges connect points close in **geodesic distance**

# Example



# 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  $\epsilon$
- ▶ Create a graph with one node  $i$  per point  $\vec{x}^{(i)}$
- ▶ Add edge between nodes  $i$  and  $j$  if  $\|\vec{x}^{(i)} - \vec{x}^{(j)}\| \leq \epsilon$
- ▶ **Result:** unweighted graph

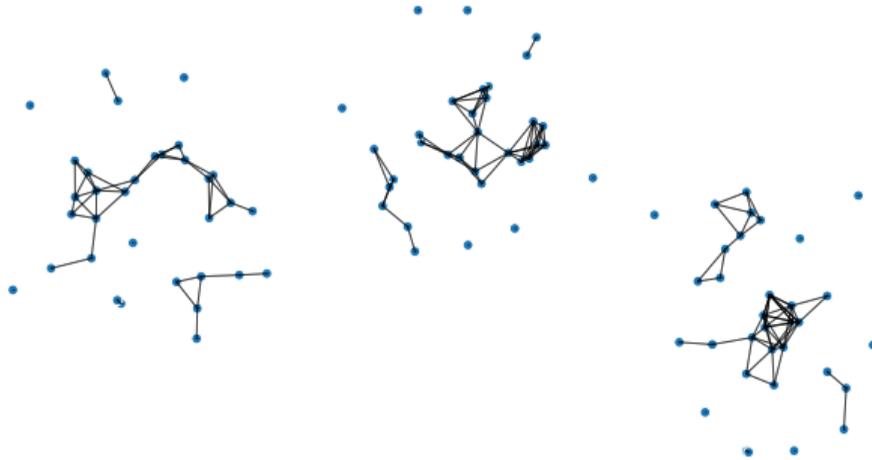


## Exercise

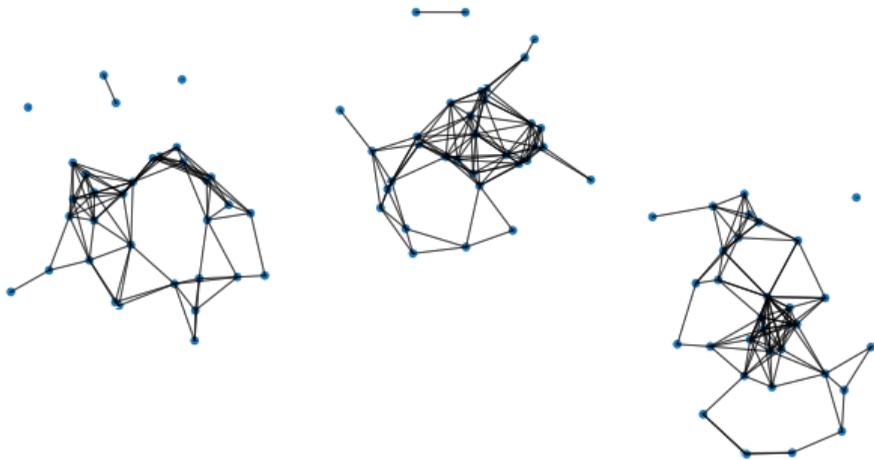
What will the graph look like when  $\varepsilon$  is small? What about when it is large?



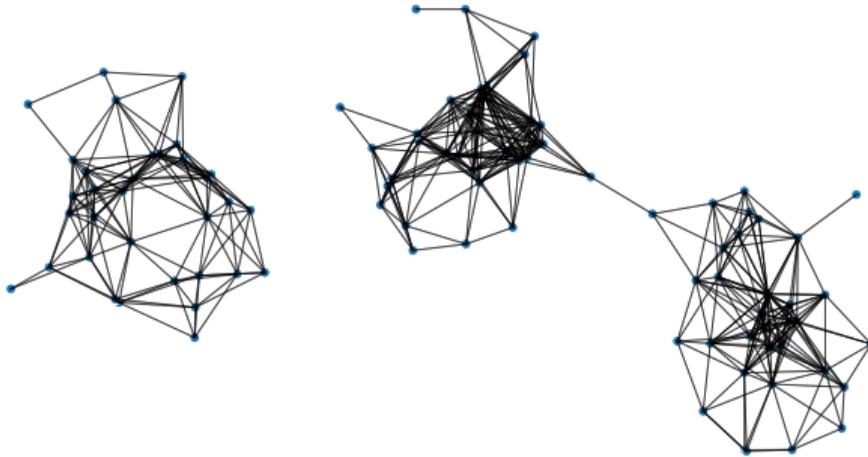
# Epsilon Neighbors Graph



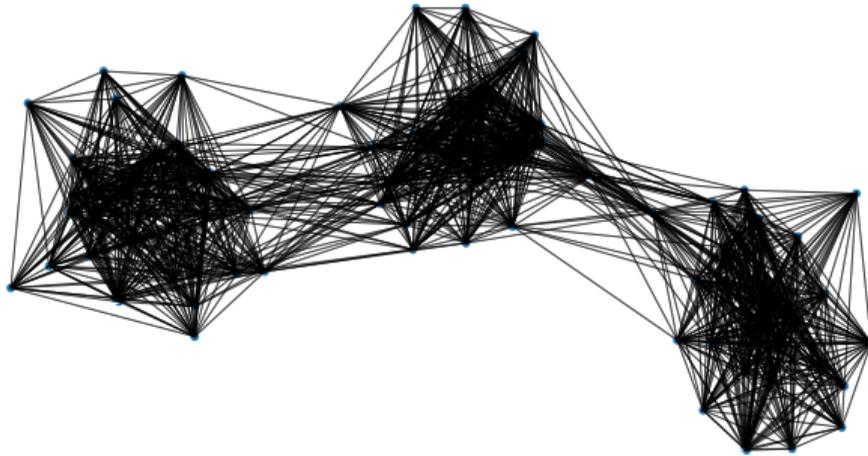
# Epsilon Neighbors Graph



# 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 anywhere.
- ▶ **Key Point:** the graph has no information about where the points were; just which were **close**.

# 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 $\varepsilon$

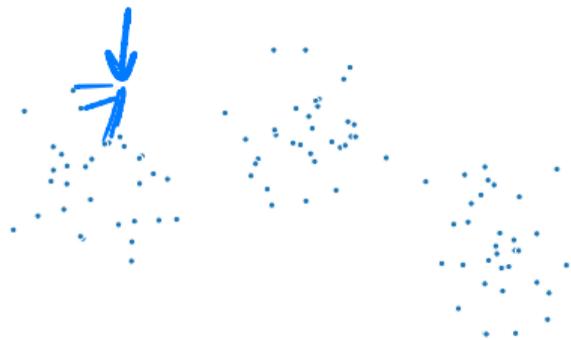
- ▶ If  $\varepsilon$  is **too small**, graph is underconnected
- ▶ If  $\varepsilon$  is **too large**, graph is overconnected
  - ▶ Not capturing the manifold structure
- ▶ If you cannot visualize, test different values and pick one that gives good results

## 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

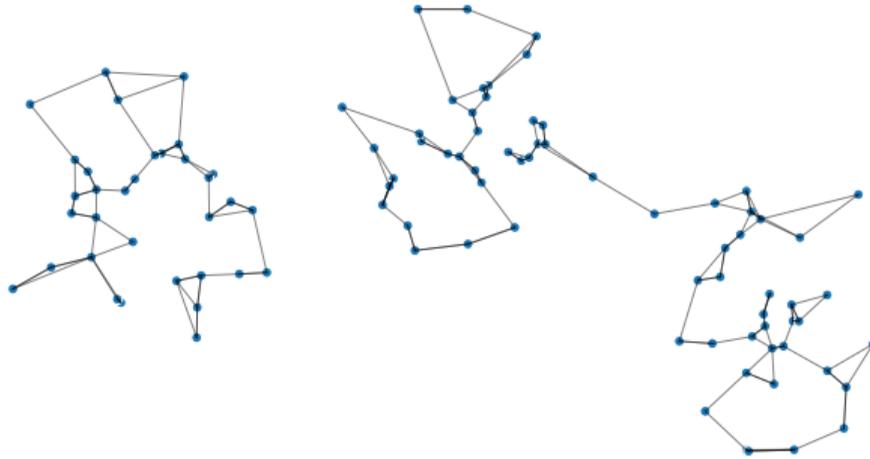
```
# assume the data is in X  
n = len(X)  
adj = np.zeros_like(X)  
for i in range(n):  
    for j in k_closest_neighbors(X, i):  
        adj[i, j] = 1
```

Live Q&A

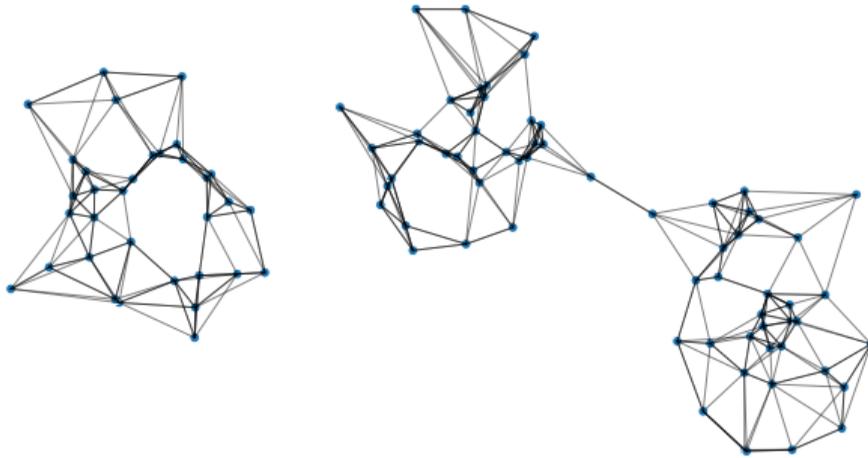
## Exercise

True or False: Is it possible for a  $k$ -neighbors graph to be disconnected?

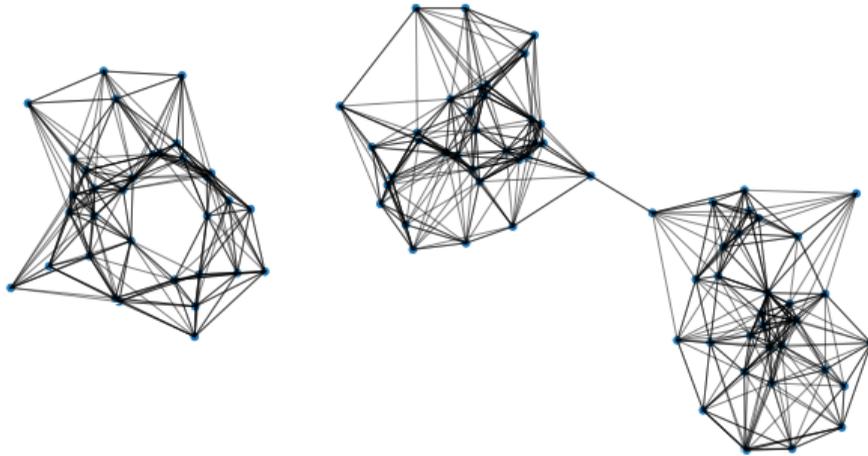
# k-Neighbors Graph



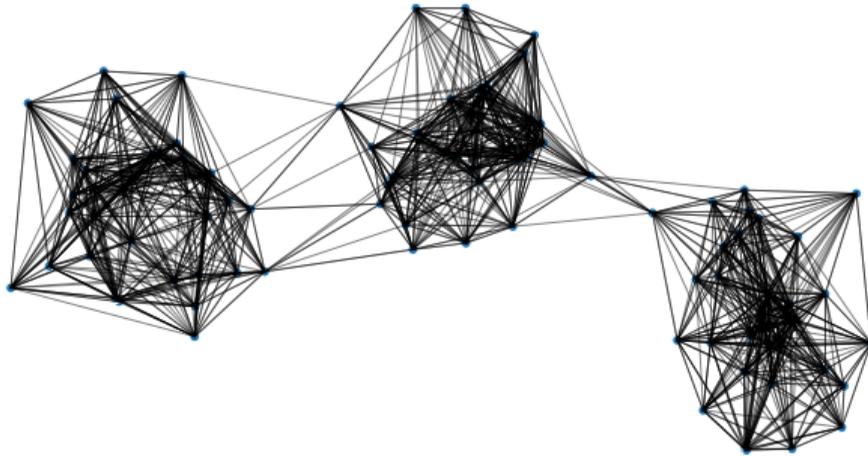
# k-Neighbors Graph



# 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**

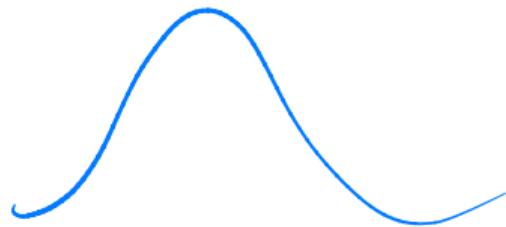


# Similarity Functions

- ▶ A **similarity function**  $h(\vec{x}, \vec{y})$  measures how “similar” two points are
- ▶ A good similarity function should satisfy:
  - ▶ Output is positive:  $h(\vec{x}, \vec{y}) \geq 0$
  - ▶ Symmetry:  $h(\vec{x}, \vec{y}) = h(\vec{y}, \vec{x})$
  - ▶ Larger values = more similar

# Gaussian Similarity

- ▶ A common similarity function: Gaussian<sup>1</sup>
- ▶ Must choose  $\sigma$  appropriately:



if  $\vec{x} = \vec{y} \rightarrow h = 1$

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

$\vec{x}$  &  $\vec{y}$  are very far apart  $\rightarrow h \approx e^{-\text{large \#}} = \frac{1}{e^{\text{large \#}}} \approx 0$

---

<sup>1</sup>Note that this isn't a normalized Gaussian pdf, but we still call it "Gaussian".

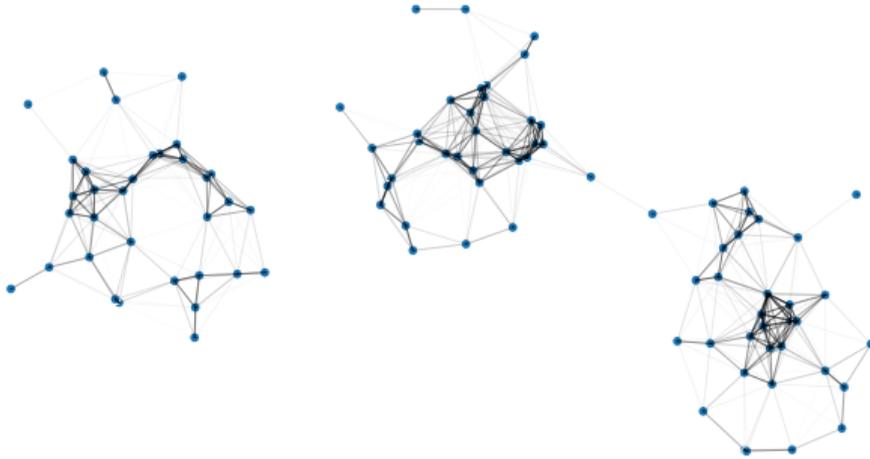
# Fully Connected: Pseudocode

```
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])
```

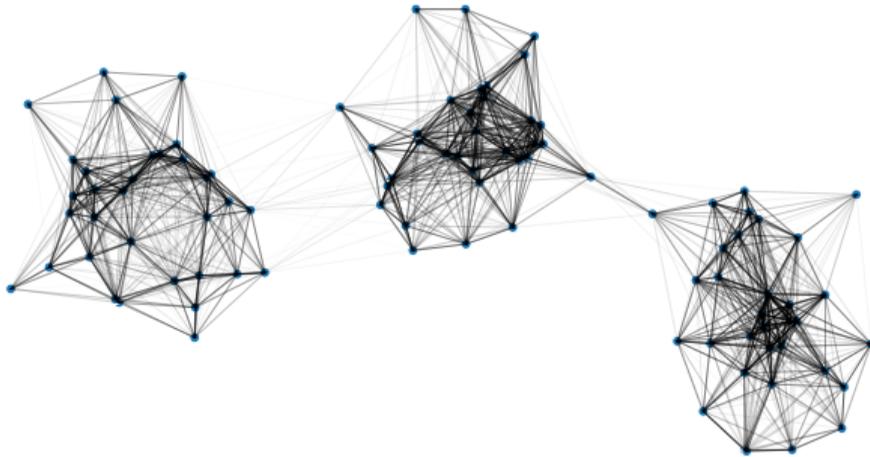
## With SciPy

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

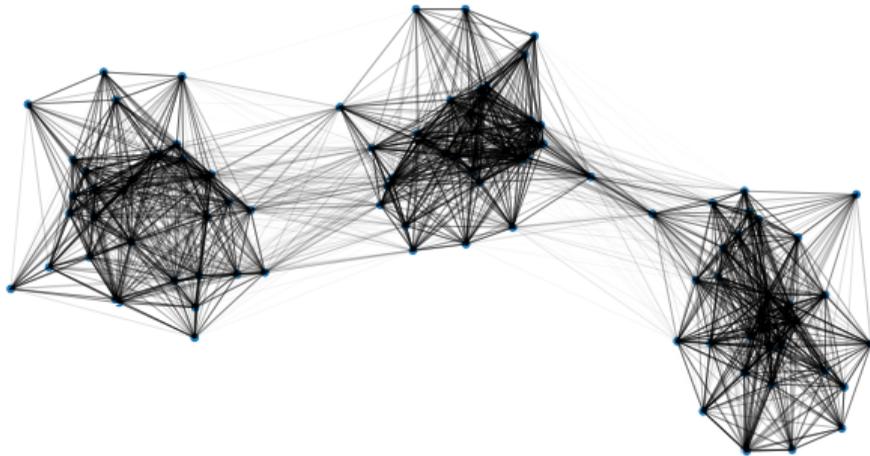
# Gaussian Similarity



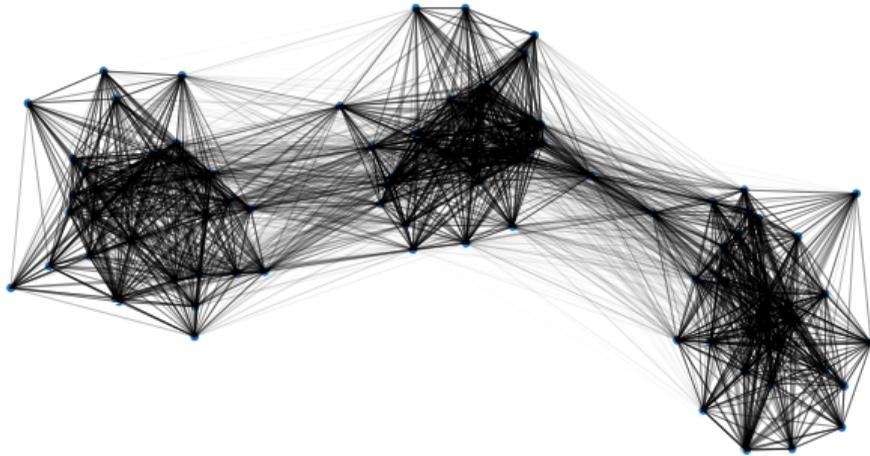
# Gaussian Similarity



# Gaussian Similarity



# Gaussian Similarity



# Next Step

- ▶ **Overall goal:** map points that are close in geodesic distance in  $\mathbb{R}^d$  to points that are close in Euclidean distance in  $\mathbb{R}^k$
- ▶ **First step:** convert points to similarity graph
- ▶ **Next step:** map similar nodes to points close in  $\mathbb{R}^k$

# DSC 140B

## Representation Learning

Lecture 08 | Part 3

**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

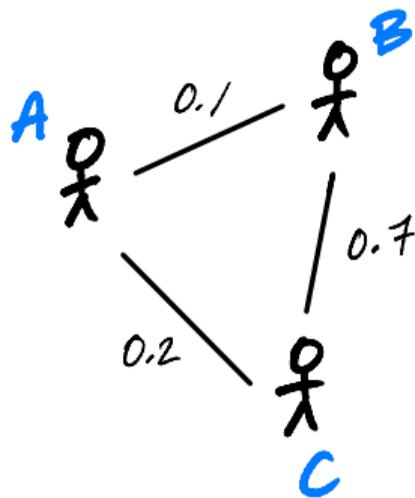
# Idea

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

# Similarity Graphs

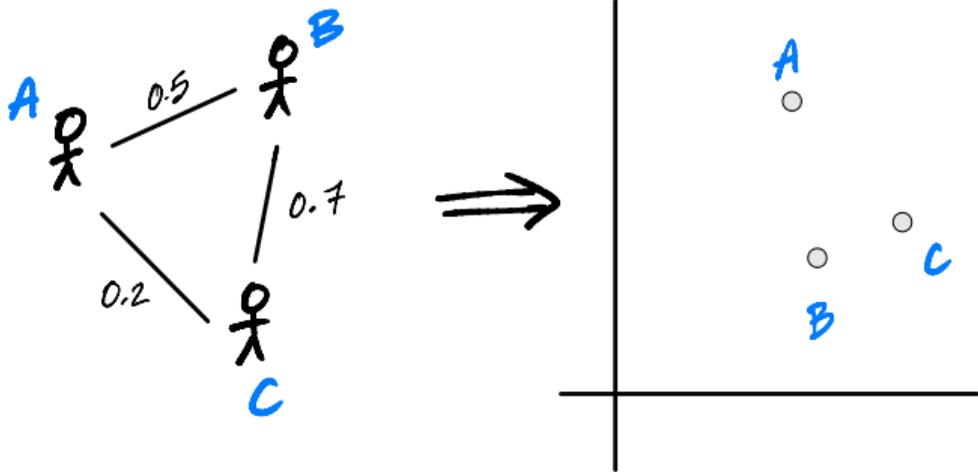
- ▶ Similarity matrices can be thought of as weighted graphs, and *vice versa*.

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



# 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  $n$  nodes into  $\mathbb{R}^k$  so that similar objects are placed nearby

# To Start

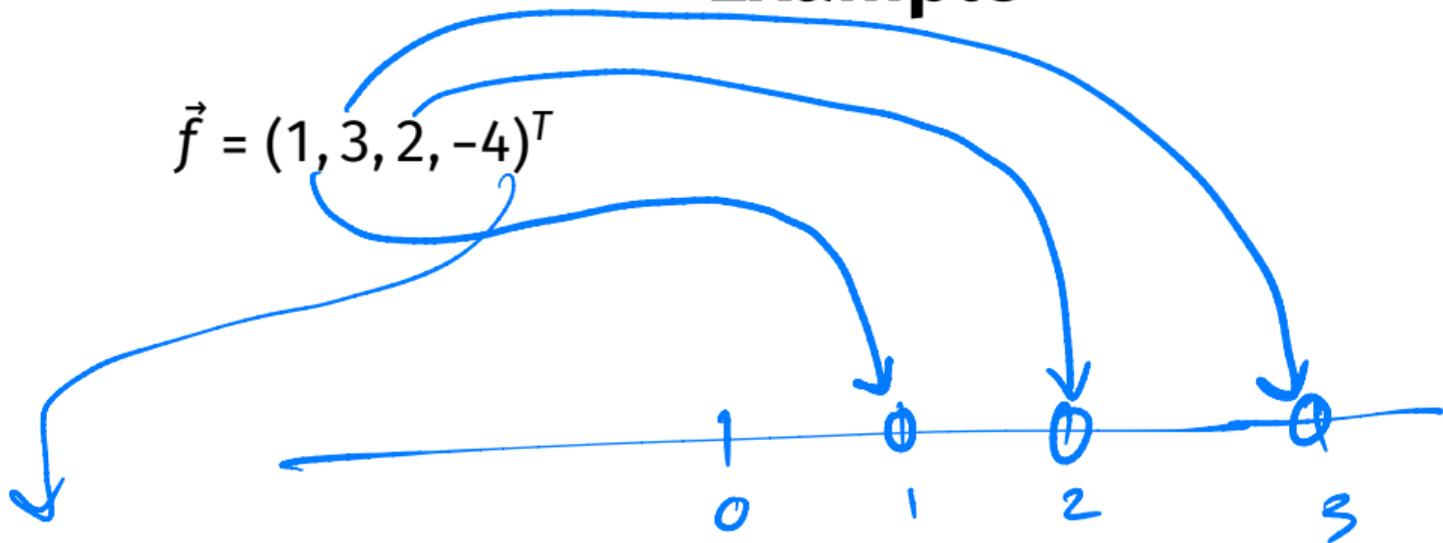
- ▶ **Given:**
  - ▶ A **similarity graph** with  $n$  nodes
  - ▶ a number of dimensions,  $k$
- ▶ **Compute:** an **embedding** of the  $n$  nodes into  $\mathbb{R}^k$  so that similar objects are placed nearby

# 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:  $\vec{f} = (f_1, f_2, \dots, f_n)$
- ▶ **Goal:** find a good set of embeddings,  $\vec{f}$ .

# Example

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



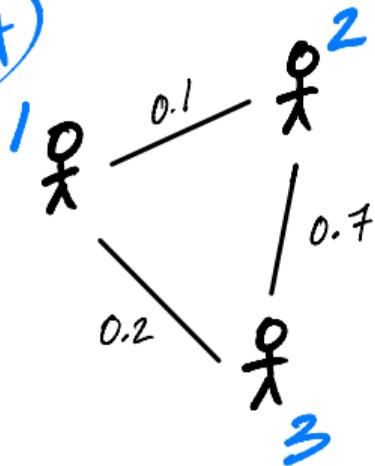
# An Optimization Problem

- ▶ 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

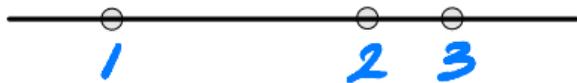
## Exercise

Which of the following embeddings is intuitively the best, given the similarity graph below?

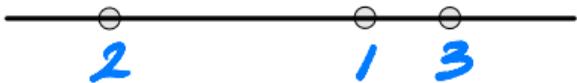
Live Q&A



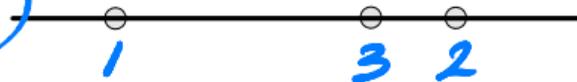
A)



~~B)~~



C)



# Cost Function for Embeddings

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

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

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

# Interpreting the Cost

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

- ▶ 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?

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

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

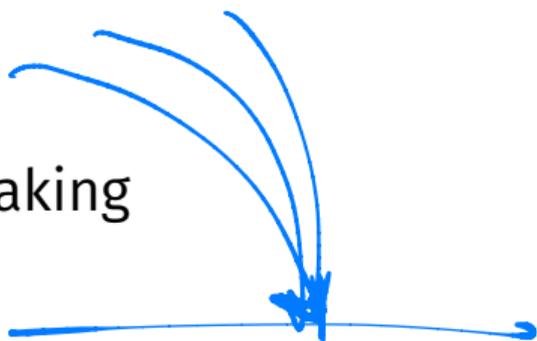
Do you see **another** problem with the cost function, even if we require  $\vec{f}$  to be a unit vector?

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

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

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

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 08 | Part 4

## **The Graph Laplacian**

# The Problem

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

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

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

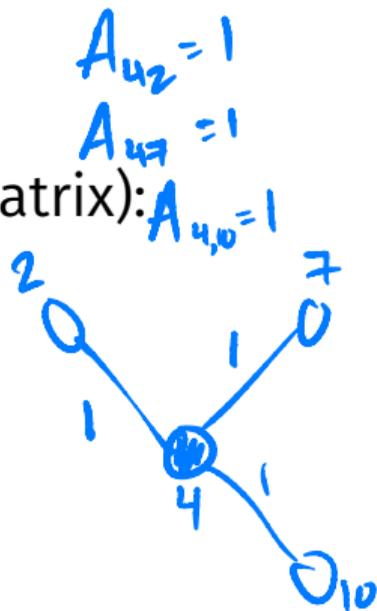
- ▶ Now: write the cost function as a matrix expression.

# 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  $i$

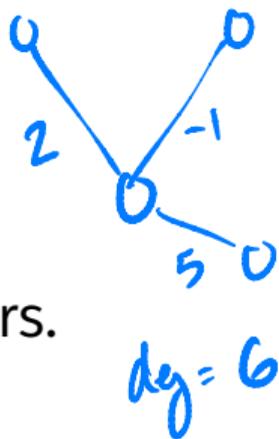


# The Degree Matrix

- ▶ 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:

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

$$\begin{pmatrix} d_{11} & 0 & 0 & \dots \\ 0 & d_{22} & 0 & \dots \\ 0 & 0 & d_{33} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

# The Graph Laplacian

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

# Very Important Fact

► **Claim:**

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

► **Proof:** expand both sides.

# DSC 140B

*Representation Learning*

Lecture 08 | Part 5

**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~~ <sup>A</sup> bottom eigenvector is  $\vec{f} = \frac{1}{\sqrt{n}}(1, 1, \dots, 1)^T$
- ▶ It has associated eigenvalue of 0.
- ▶ That is,  $L\vec{f} = 0\vec{f} = \vec{0}$   $\vec{f}^T L \vec{f} = 0$

## Corollary: Spectral<sup>2</sup> Theorem

### Theorem

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

---

<sup>2</sup>“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**

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

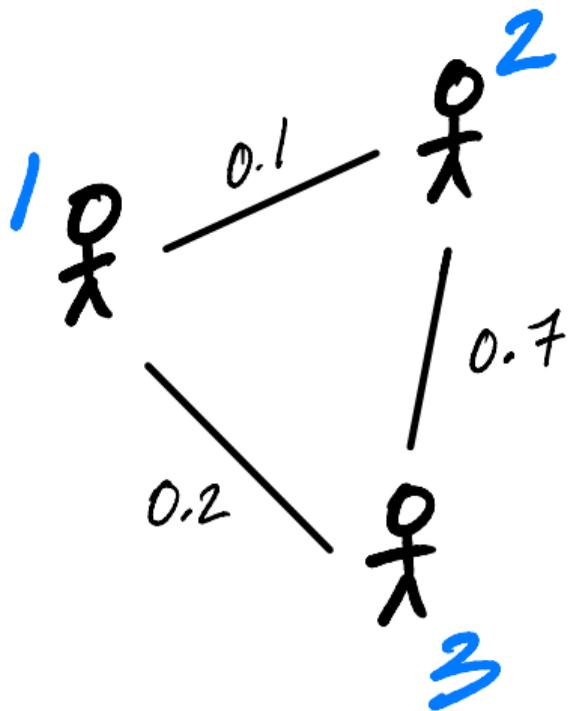
subject to  $\|\vec{f}\| = 1$  and  $\vec{f} \perp (1, 1, \dots, 1)^T$

# 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!

▷ The eigenvalue  $\lambda$  is the cost of the embedding.

# Example



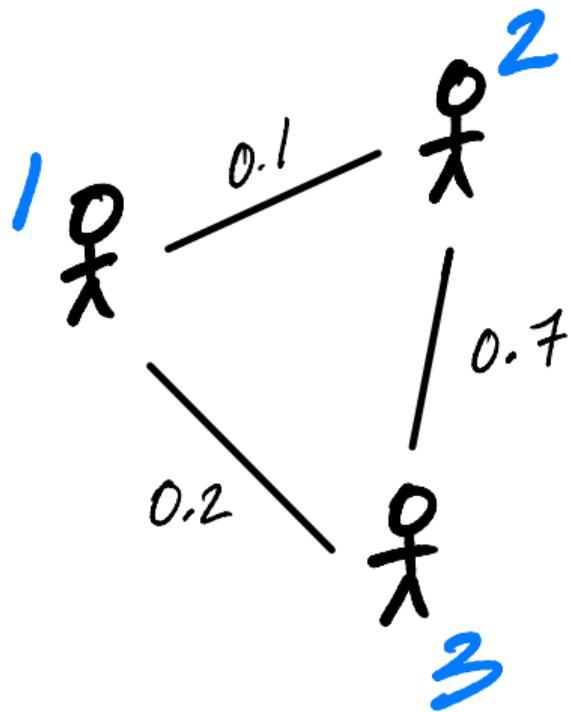
```
W = np.array([
    [1, 0.1, 0.2],
    [0.1, 1, 0.7],
    [0.2, 0.7, 1]
])
```

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

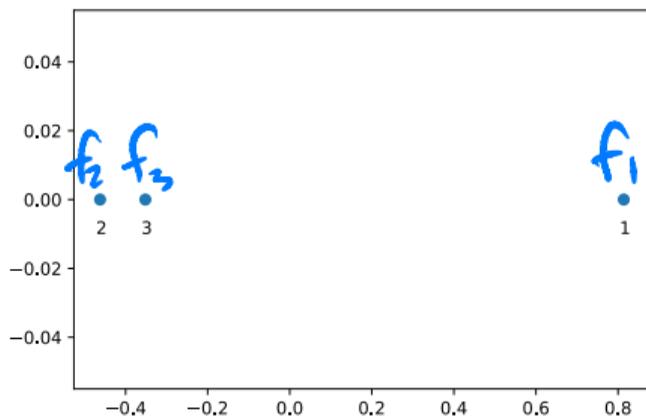
```
vals, vecs = np.linalg.eigh(L)
```

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

# Example



$$\vec{f} \approx (.8, -.5, -.35)$$



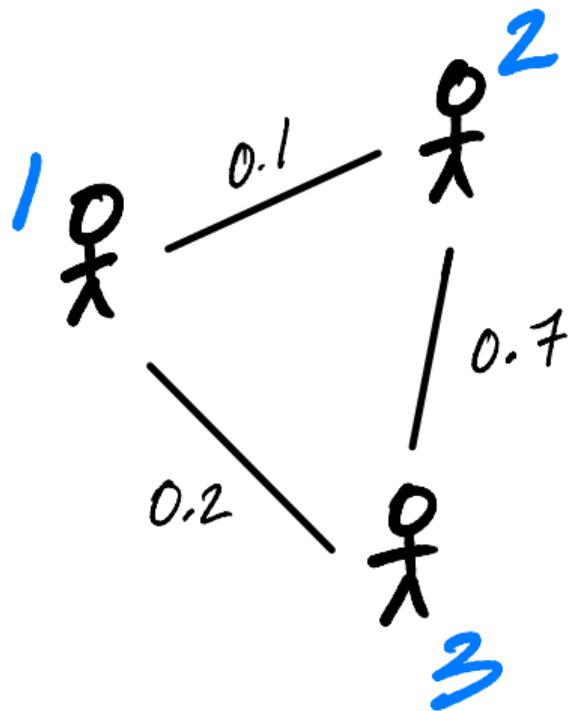
# Embedding into $\mathbb{R}^k$

- ▶ 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  $f_i^{(1)}$ .
  - ▶ Second new coordinate is  $f_i^{(2)}$ .
  - ▶ Third new coordinate is  $f_i^{(3)}$ .
  - ▶  $\vdots$

# Example



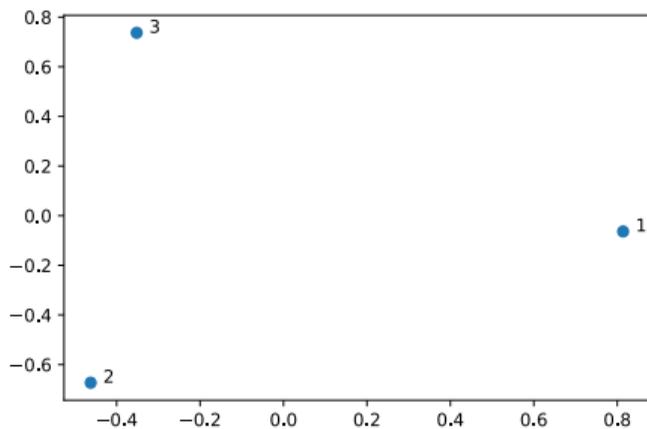
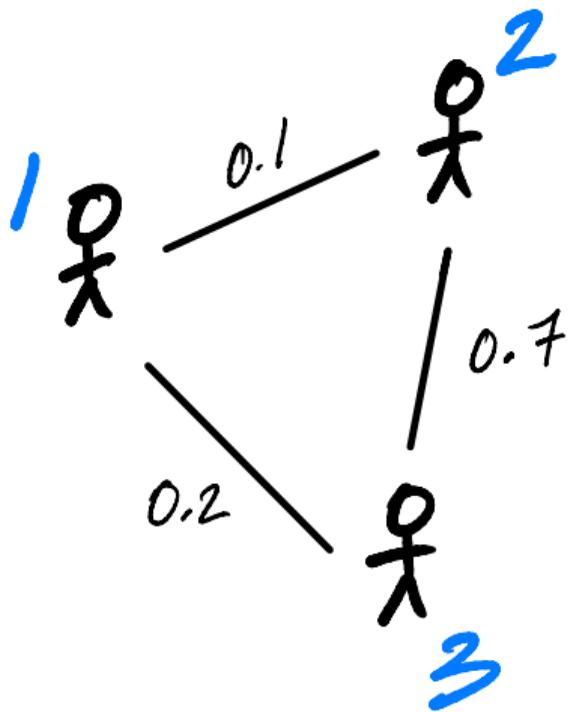
```
W = np.array([
    [1, 0.1, 0.2],
    [0.1, 1, 0.7],
    [0.2, 0.7, 1]
])
```

```
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 Belkin<sup>3</sup> and Partha Niyogi
- ▶ It is a type of **spectral embedding**

---

<sup>3</sup>Now 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  $L_{\text{norm}}$ .

# 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 08 | Part 6

**Laplacian Eigenmaps**

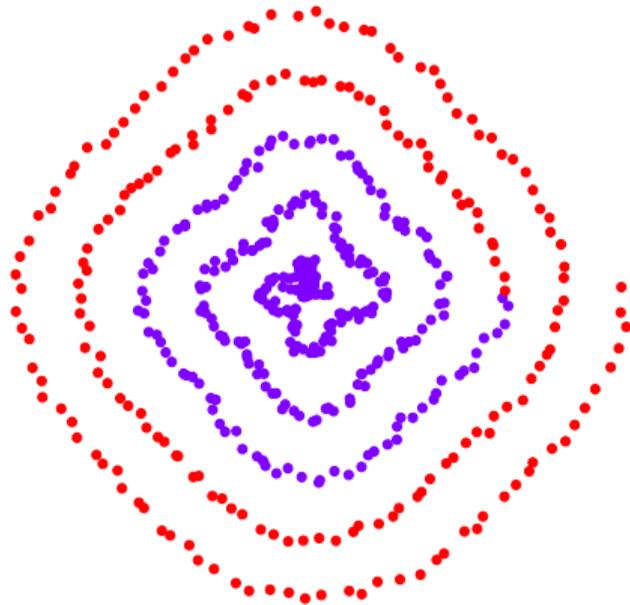
# Problem: Non-linear Dimensionality Reduction

- ▶ **Given:** points in  $\mathbb{R}^d$ , target dimension  $k$
- ▶ **Goal:** **embed** the points in  $\mathbb{R}^k$  so that points that were close in geodesic distance are close after

# Idea

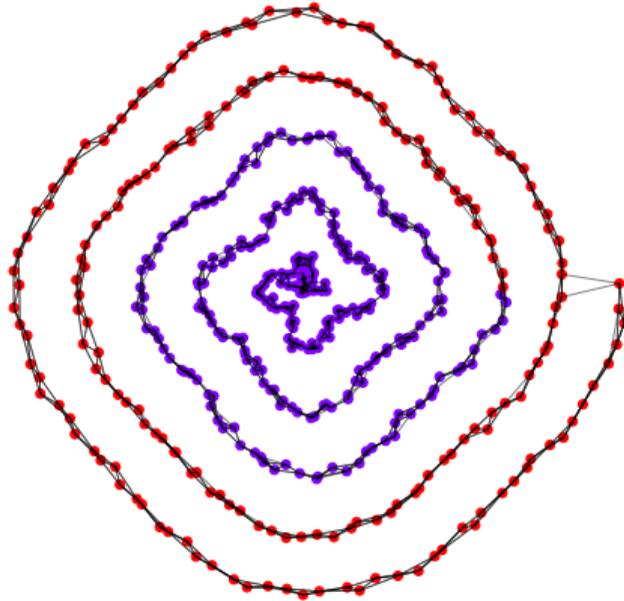
- ▶ Build a similarity graph from points in  $\mathbb{R}^d$ 
  - ▶ epsilon neighbors,  $k$ -neighbors, or fully connected
- ▶ Embed the similarity graph in  $\mathbb{R}^k$  using eigenvectors of graph Laplacian

# Example 1: Spiral



# Example 1: Spiral

- ▶ 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

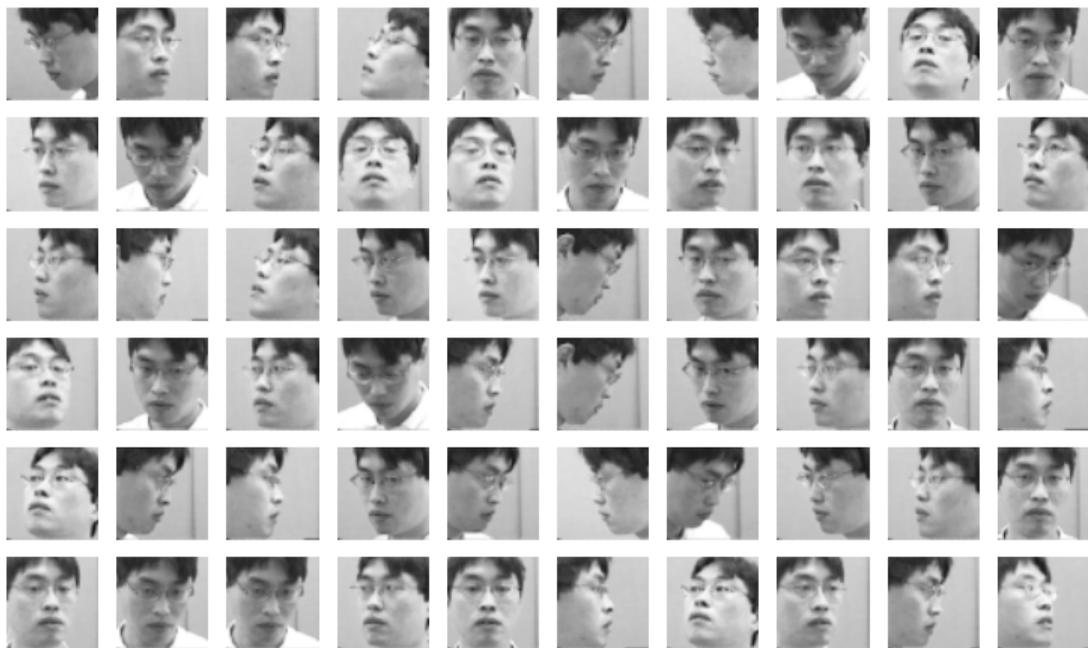
- ▶ Embedding into  $\mathbb{R}^2$



# Example 1: Spiral

```
import sklearn.neighbors
import sklearn.manifold
W = sklearn.neighbors.kneighbors_graph(
    X, n_neighbors=4
)
embedding = sklearn.manifold.spectral_embedding(
    W, n_components=2
)
```

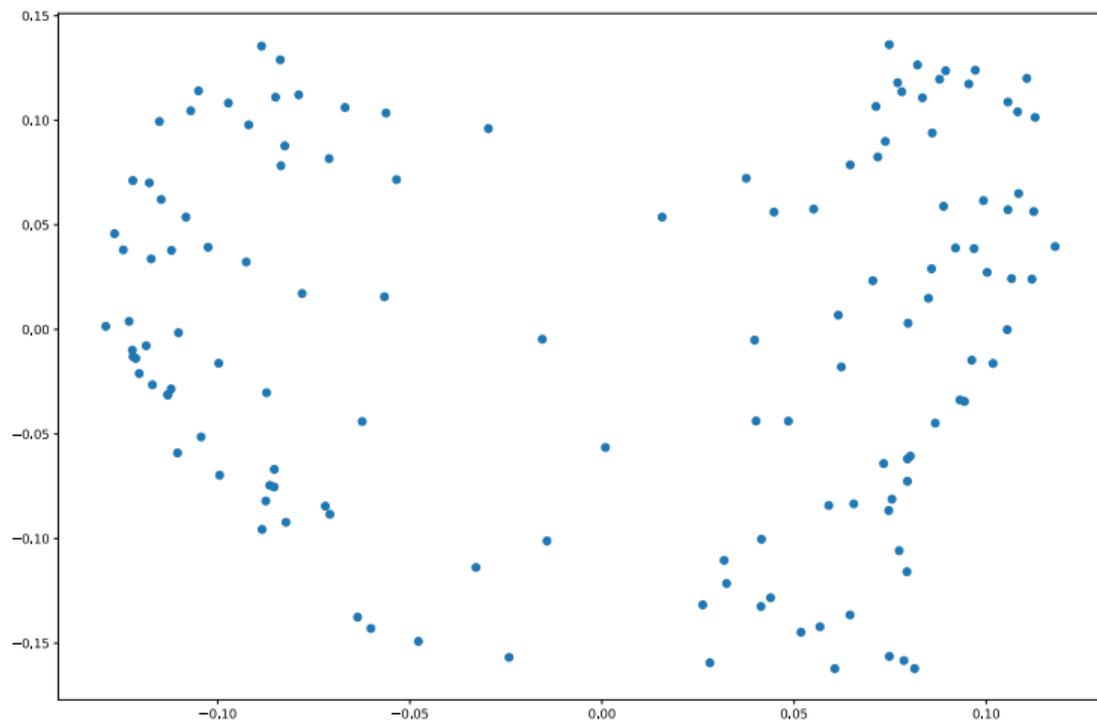
## Example 2: Face Pose



## Example 2: Face Pose

- ▶ Construct fully-connected similarity graph with Gaussian similarity
- ▶ Embed with Laplacian eigenmaps

# Example 2: Face Pose



# Example 2: Face Pose

