HANDS-ON NETWORK MACHINE LEARNING

by
Alexander Russell Loftus

A thesis submitted to The Johns Hopkins University in conformity with the requirements for the degree of Master of Science in Engineering

Baltimore, Maryland
December, 2021

Abstract

This thesis is a general overview of spectral methods on networks, and how you can use tools from a network's eigenspace to understand and explain the network more deeply.

Networks are some of the most fundamental building blocks of the universe. Atoms and molecules are connected to each other with chemical bonds. Neurons connect to each other through synapses, and the different parts of the brain connect to each other through groups of neurons interacting with each other. At a larger level, we are interconnected with other humans through social networks, and our economy is a global, interconnected trade network. The Earth's food chain is an ecological network, and larger still, every object with mass in the universe is connected to every other object through a gravitational network.

This thesis covers the fundamentals of spectral methods with respect to network data science, focusing on developing intuition on networks as statistical objects, while paired with relevant Python code.

While you can read this thesis without picking up your laptop, I highly recommend you experiment with the code examples available online as Jupyter notebooks at https://loftusa.github.io/thesis

Primary Reader and Advisor: Joshua Vogelstein

Secondary Readers: Avanti Athreya, Carey Priebe

Acknowledgements

Big thanks to everybody who has been reading the thesis as I write and giving feedback. This list includes Dax Pryce, Ross Lawrence, Geoff Loftus, Alexandra McCoy, Olivia Taylor Peter Brown, Sambit Panda, Eric Bridgeford, Josh Vogelstein, and Ali Sad-Aldin.

I am grateful to my advisor, Joshua Vogelstein, for his insights and strong feedback. The value he puts on clarity and simplicity in any mathematical model has been an enormous help throughout this process.

I am also especially grateful to Eric Bridgeford, who has been giving me constant feedback throughout the writing process. I would be lost in a sea of papers without his help.

This thesis is dedicated to my father, Geoffrey Loftus, for teaching me the value of rigor in science and for being a resoundingly positive role model throughout my life; to my mother, Susan Loftus, for teaching me to never give up in the face of adversity; to my sister, Emma Loftus, for accepting me no matter what; to my stepfather, Matthew Voorsanger, for showing me when to be pragmatic and when to use humor; to my stepmother, Willa Rose, for always being there to listen; and to my stepsiblings for illuminating and enlarging my life.

Contents

List of Figures

Chapter 1

Matrix Representations Of Networks

When we work with networks, we need a way to represent them mathematically and in our code. A network itself lives in network space, which is just the set of all possible networks. Network space is kind of abstract and inconvenient if we want to use traditional mathematics, so we'd generally like to represent networks with groups of numbers to make everything more concrete.

More specifically, we would often like to represent networks with matrices. In addition to being computationally convenient, using matrices to represent networks lets us bring in a surprising amount of tools from linear algebra and statistics. Programmatically, using matrices also lets us use common Python tools for array manipulation like numpy.

The most common matrix representation of a network is called the Adjacency Matrix, and we'll learn about that first.

The Adjacency Matrix

The beating heart of matrix representations for networks throughout this book is the adjacency matrix. The idea is pretty straightforward: Let's say you have a network with n nodes. You give each node an index – usually some value between 0 and n –

and then you create an $n \times n$ matrix. If there is an edge between node i and node j, you fill the $(i, j)_{th}$ value of the matrix with an entry, usually 1 if your network has unweighted edges. In the case of undirected networks, you end up with a symmetric matrix with full of 1's and 0's, which completely represents the topology of your network.

Let's see this in action. We'll make a network with only three nodes, since that's small and easy to understand, and then we'll show what it looks like as an adjacency matrix (fig 1.1).

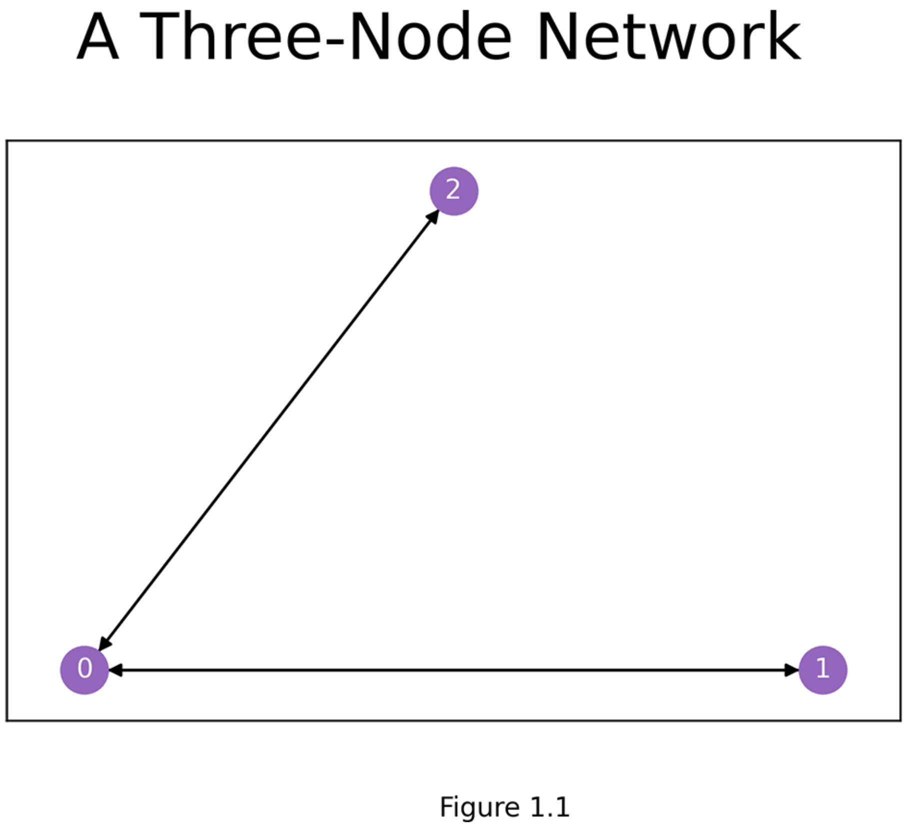


Figure 1-1. A Three-Node Network

Our network has three nodes, labeled 1, 2, and 3. Each of these three nodes is either connected or not connected to each of the two other nodes. We'll make a square matrix X, with 3 rows and 3 columns, so that each node has its own row and column associated to it.

So, let's fill out the matrix. We start with the first row, which corresponds to the first node, and move along the columns. If there is an edge between the first node and the node whose index matches the current column, put a 1 in the current location. If the two nodes aren't connected, add a 0. When you're done with the first row, move on to the second. Keep going until the whole matrix is filled with 0's and 1's.

The end result looks like the matrix in figure 1.2. Since the second and third nodes aren't connected, there is a 0 in locations $A_{2,1}$ and $A_{1,2}$. There are also zeroes along the diagonals, since nodes don't have edges with themselves.

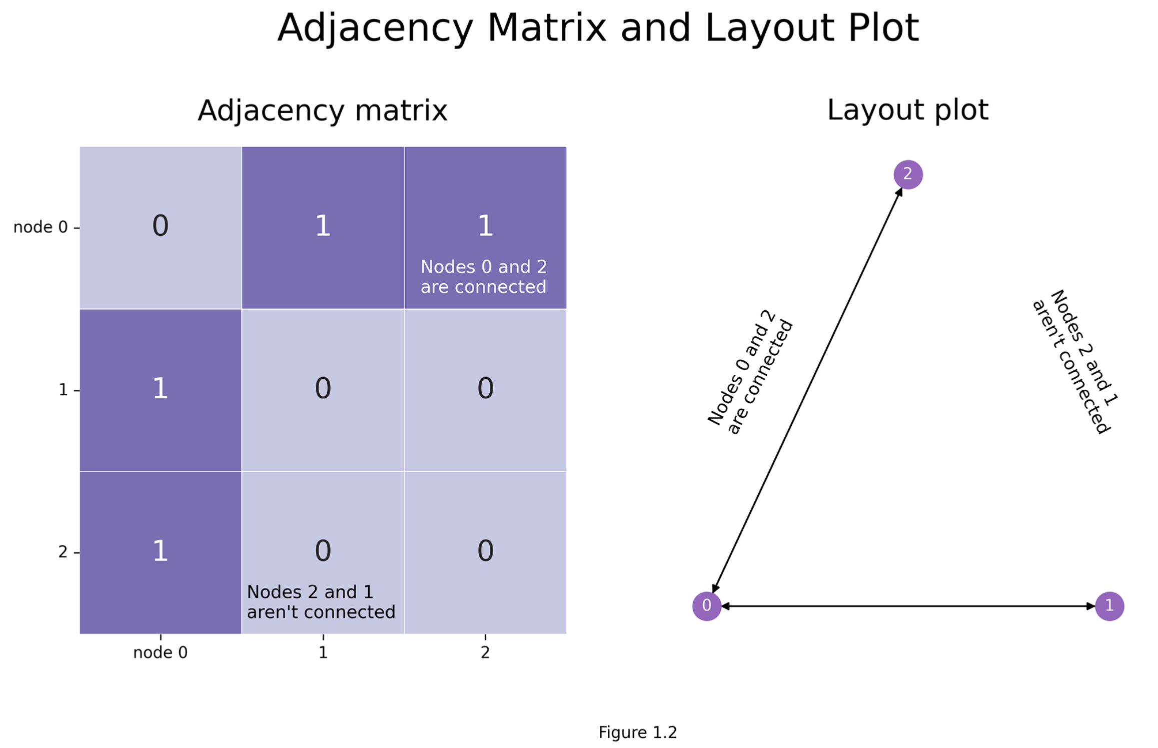

Adjacency Matrix and Layout Plot
Adjacency matrix
Layout plot
node 0
1
2
Nodes 0 and 2 are connected
Nodes 2 and 1 aren't connected
Nodes 0 and 2 are connected
Nodes 2 and 1 aren't connected
Figure 1.2

Figure 1-2. Adjacency Matrix and Layout Plot

Although the adjacency matrix is straightforward and easy to understand, it isn't the only way to represent networks. The Incidence Matrix is a larger, sparser representation

The Incidence Matrix

Instead of having values in a symmetric matrix represent possible edges, like with the Adjacency Matrix, we could have rows represent nodes and columns represent edges. This is called the *Incidence Matrix*, and it's useful to know about – although it won't appear too much in this book. If there are n nodes and m edges, you make an $n \times m$ matrix. Then, to determine whether a node is a member of a given edge, you'd go to that node's row and the edge's column. If the entry is nonzero (1 if the network is unweighted), then the node is a member of that edge, and if there's a 0, the node is not a member of that edge.

You can see the incidence matrix for our network in fig 1-3. Notice that with incidence plots, edges are (generally arbitrarily) assigned indices as well as nodes.

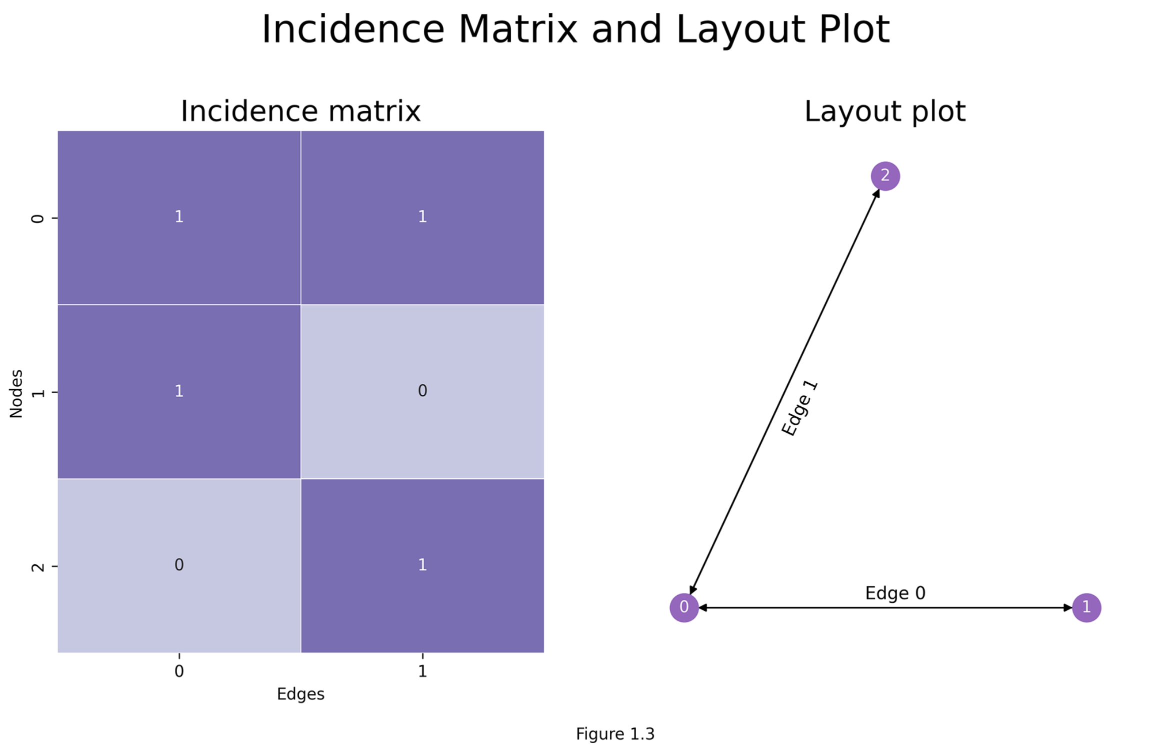


Figure 1-3. Incidence Matrix and Layout Plot

When networks are large, incidence matrices tend to be extremely sparse – meaning, their values are mostly 0's. This is because each column must have exactly two nonzero values along its rows: one value for the first node its edge is connected to, and another for the second. Because of this, incidence matrices are usually represented in Python computationally as scipy's *sparse matrices* rather than as numpy arrays, since this data type is much better-suited for matrices which contain mostly zeroes.

You can also add orientation to incidence matrices, even in undirected networks, which we'll discuss next.

The Oriented Incidence Matrix

The oriented incidence matrix is extremely similar to the normal incidence matrix, except that you assign a direction or orientation to each edge: you define one of its nodes as being the head node, and the other as being the tail. For undirected networks, you can assign directionality arbitrarily. Then, for the column in the incidence matrix corresponding to a given edge, the tail node has a value of -1, and the head node has a value of 0. Nodes who aren't a member of a particular edge are still assigned values of 0. You can see what it looks like in fig 1-4.

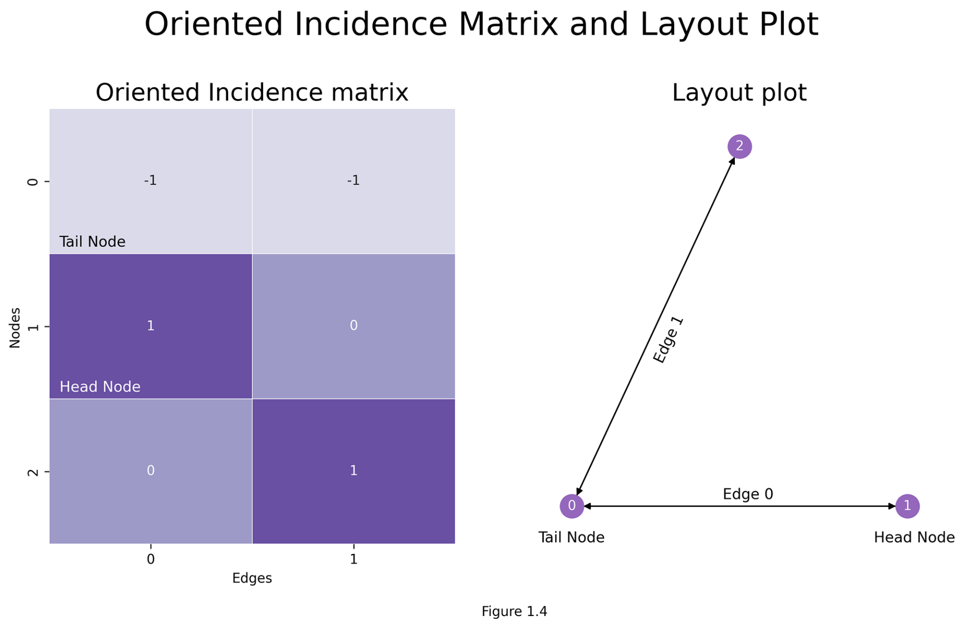


Figure 1-4. Oriented Incidence Matrix and Layout Plot

Although we won't use incidence matrices, oriented or otherwise, in this thesis too much, we introduced them because there's a deep connection between incidence matrices, adjacency matrices, and a matrix representation that we haven't introduced yet called the Laplacian.

The Laplacian Matrix

The standard, cookie-cutter Laplacian Matrix L is just the adjacency matrix A subtracted from the the degree matrix D.

$L = D - A$

Since the only nonzero values of the degree matrix is along its diagonals, and because the diagonals of an adjacency matrix never contain zeroes if its network doesn't have nodes connected to themselves, the diagonals of the Laplacian are just the degree of each node. The values on the non-diagonals work similarly to the adjacency matrix: they contain a -1 if there is an edge between the two nodes, and a 0 if there is no edge.

Figure 1-5 below shows you what the Laplacian looks like. Since each node has exactly two edges, the degree matrix is just a diagonal matrix of all twos. The Laplacian looks like the degree matrix, but with -1's in all the locations where an edge exists between nodes i and j.

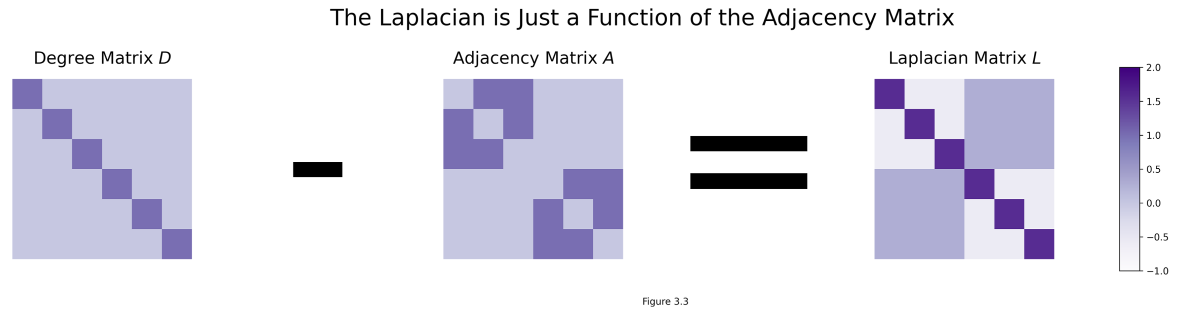


Figure 1-5. The Laplacian is Just a Function of the Adjacency Matrix

We use the Laplacian in practice because it has a number of interesting mathematical properties, which tend to be useful for analysis. For instance, the magnitude of its second-smallest eigenvalue, called the Fiedler eigenvalue, tells you how well-connected your network is – and the number of eigenvalues equal to zero is the number of connected components your network has (a connected component is a group of nodes in a network which all have a path to get to each other – think of it as an island of nodes and edges). Laplacians and adjacency matrices will be used throughout this thesis, and details about when and where one or the other will be used are in the spectral embedding chapter.

Chapter 2

Why Embed Networks?

Networks by themselves can have interesting properties, but a network is not how we traditionally organize data in machine learning. In almost any ML algorithm - whether you're using a neural network or a decision tree, whether your goal is to classify observations or to predict values using regression - you'll see data organized into a matrix, where the rows represent observations and the columns represent features, or variables. Each observation, or row of the matrix, is traditionally represented as a single point in d-dimensional space (if there are d columns in the matrix). If you have two columns, for instance, you could represent data organized in this way on an x/y coordinate plane. The first column would represent the x-axis, and the second column would represent the y-axis.

For example, the data in fig 2-1 is organized traditionally. On the left is the data matrix; each observation has its own row, with two features across the columns. The x-column contains the first feature for each observation, and the y-column contains the second feature for each observation. We can see the two clusters of data numerically, through the color mapping.

On the right is the same data, but plotted in Euclidean space. Each column of the data matrix gets its own axis in the plot. The x and y axis location of the i^{th} point in the scatterplot is the same as the x and y values of the i^{th} row of the data matrix.

We can see the two clusters of data geometrically, through the location of the points.

You can see the code which generated this data below.

```
from sklearn.datasets import make_blobs
import pandas as pd
import numpy as np

# make the data
centers = np.array([[-2, -2],
                    [2, 2]])
X, labels = make_blobs(n_samples=10, cluster_std=0.5,
                  centers=centers, shuffle=False)

# convert data into a DataFrame
data = pd.DataFrame(X, columns=["x", "y"])
```

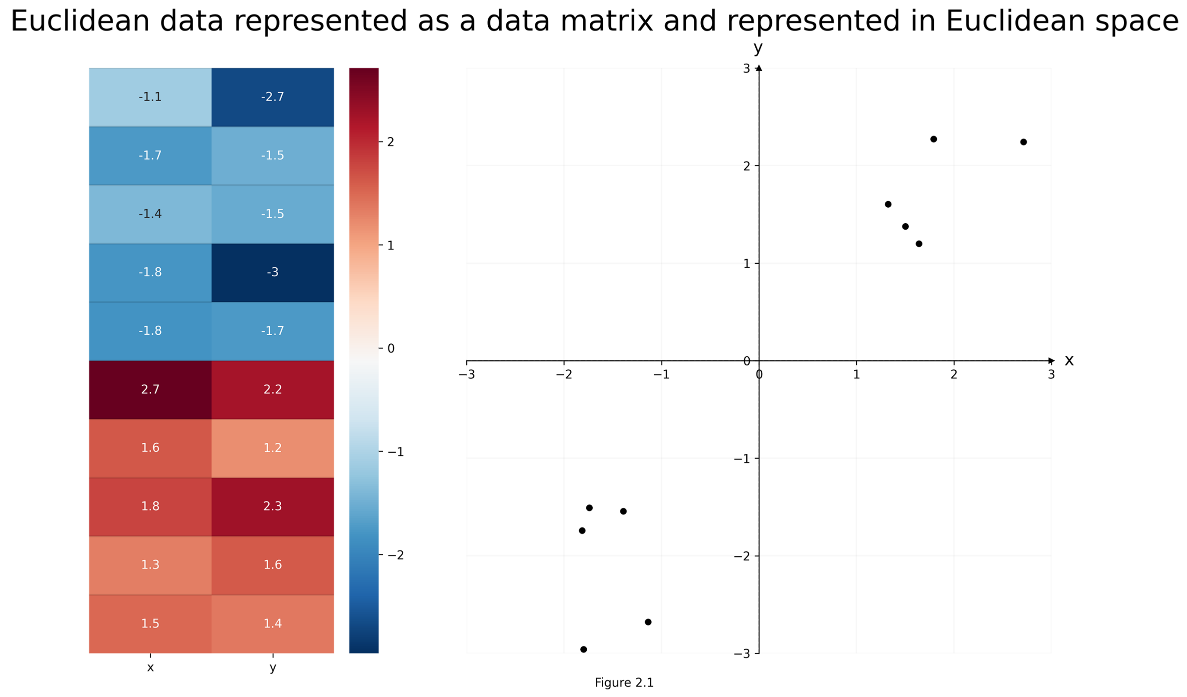


Figure 2-1. Euclidean Data Represented as a Data Matrix and in Euclidean Space

It's often useful for our data to be organized like this, since it opens the door to a wide variety of machine learning methods. With the data above, for example, we could use scikit-learn to perform simple K-Means Clustering to find the two clusters of observations. Below, we import scikit-learn's K-Means clustering algorithm. K-Means finds a pre-determined number of clusters in your data by setting randomly determined starting-points, and then iterating to get those points closer to the true cluster means.

It outputs the community membership labels for each observation, which you can see below.

```
from sklearn.cluster import KMeans

predicted_labels = KMeans(n_clusters=2).fit_predict(X)
print("Predicted␣labels:␣", predicted_labels)

>>> Predicted labels: [1 1 1 1 1 0 0 0 0 0]
```

You can see the data colored by predicted label in figure 2-1 below.

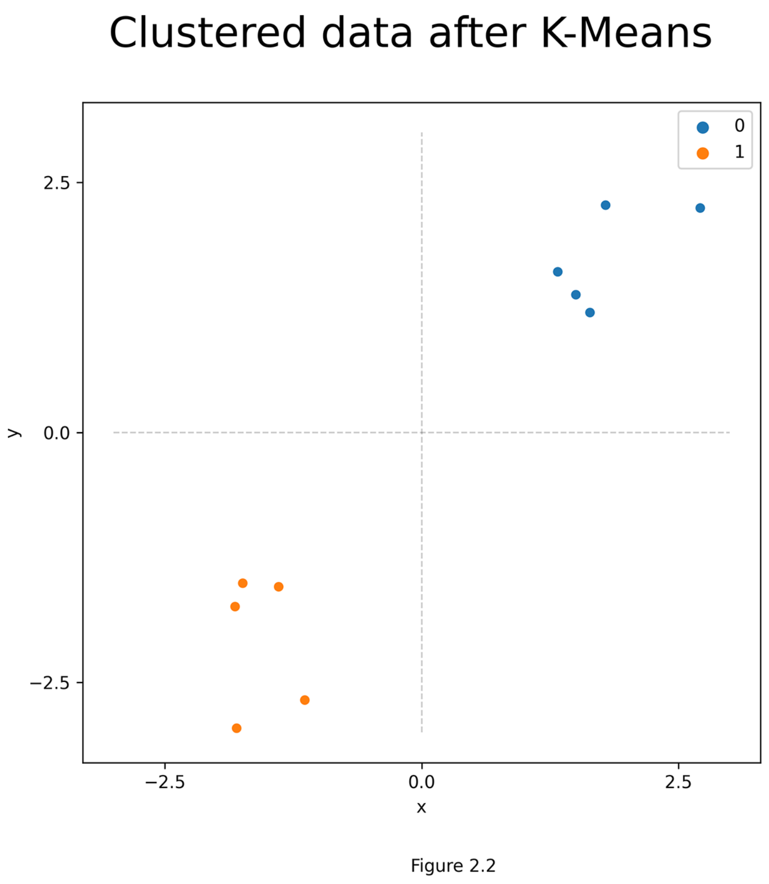


Figure 2-2. Clustered Data after K-Means

Network-valued data are different. Take the observation of a stochastic block model below, shown as both a layout plot and an adjacency matrix. Say your goal is to view the nodes as particular observations, and you'd like to cluster the data in the same way you clustered the Euclidean data above. Intuitively, you'd expect to find two groups: one for the first set of heavily connected nodes, and one for the second set. Unfortunately, traditional machine learning algorithms won't work on data represented as a network: it doesn't live in the traditional rows-as-observations, columns-as-features format.

```
import networkx as nx
from graspologic.simulations import sbm
np.random.seed(1)

p = np.array([[.9, .1],
              [.1, .9]])
A, labels = sbm([25, 25], p, return_labels=True)
```

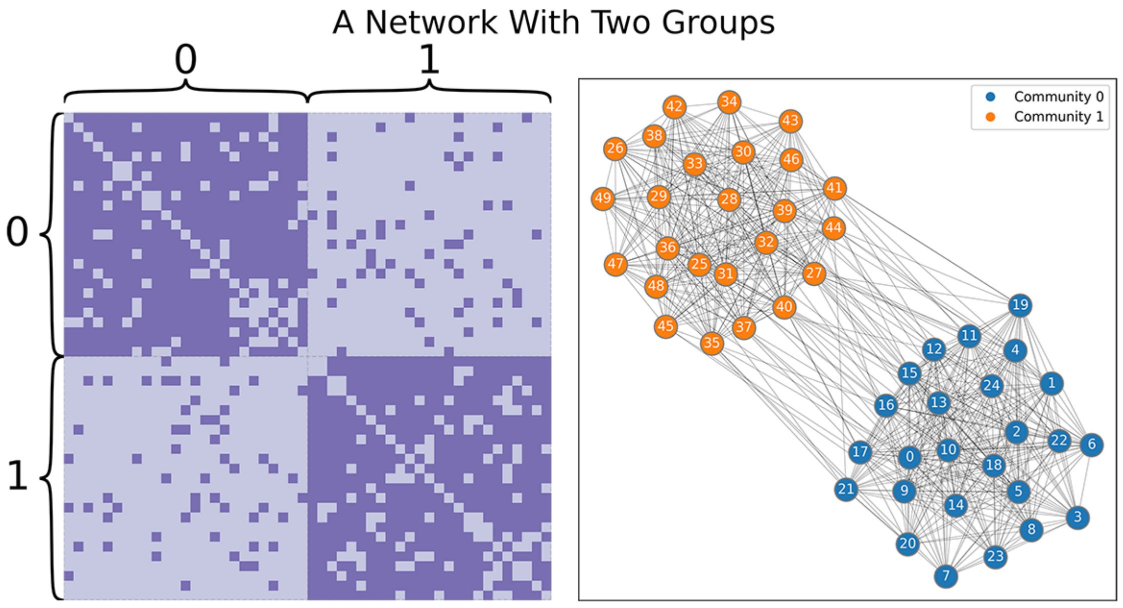


Figure 2-3. A Network with Two Groups

You, of course, can make up methods which work directly on networks - algorithms which run by traversing along edges, for instance, or which use network statistics like node degree to learn, and so on - and data scientists have developed many algorithms like this. But to be able to use the entire toolbox that machine learning offers, you'd like to be able to figure out a way to *represent* networks in Euclidean space as tabular data. This is why having good embedding methods, like Spectral Embedding (which we'll learn about soon), is useful. There's another problem with networks that make embedding into lower-dimensional space useful.

High Dimensionality of Network Data

The other problem with network data is its high dimensionality. You could view each element of an adjacency matrix as its own (binary, for unweighted networks) dimension, for instance – although you could also make the argument that talking about dimensionality doesn't even make *sense* with network data, since it doesn't live in Euclidean space. Regardless, if you were to view the elements of the adjacency matrix as their own dimensions, you can get to a fairly unmanageable number of dimensions fairly quickly. Many dimensions can generally be unmanageable largely because of a machine learning concept called the *curse of dimensionality*.

Our intuition often fails when observations have a lot of features – meaning, observations that, when you think of them geometrically, are points in very high-dimensional space.

For example, pick a point randomly in a 10,000-dimensional unit hypercube (meaning, a $1 \times 1 \times \cdots \times 1$ cube, with ten thousand 1s). You can also just think of this point as a vector with 10,000 elements, each of which has a value between 0 and 1. There's a probability greater than 99.999999% that the point will be located a distance less than .001 from a border of the hypercube. This probability is only 0.4% in a unit square. This actually makes intuitive sense: if you think about measuring a lot of attributes of an object, there's a decent chance it'll be extreme in at least one of those attributes. Take yourself, for example. You're probably normal in a lot of ways, but I'm sure you can think of a part of yourself which is extreme compared to other people.

An even bigger shocker: if you pick two random points in a unit square with two dimensions, they'll be on average 0.52 units of distance away from each other. However, if you pick two random points in a unit hypercube with a million dimensions, they'll be around 408 units away from each other. This implies that, on average, any set of points that you generate from some random process when you're in high dimensions will be extremely far away from each other.

What this comes down to is that almost every point in ultra-high dimensions is extremely lonely, hugging the edge of the space it lives in, all by itself. These facts mess with many traditional machine learning methods which use relative distances, or averages (very few observations in high-dimensional space will actually be anywhere near their average!)

Latent Estimation

The embedding methods which we'll explore the most in this book are the spectral methods. These methods pull heavily from linear algebra to keep only the information about our network which is useful - and use that information to place nodes in Euclidean space. We'll explore other methods as well. It's worth it to know a bit of linear algebra review here, particularly on concepts like eigenvectors and eigenvalues, as well as the properties of symmetric matrices. We'll guide you as clearly as possible through the math in future sections.

Spectral embedding methods in particular, which we'll talk about in the next section, will estimate the latent position matrix. This estimate is an $n \times d$ matrix (where this are n rows, one for each node, and d dimensions for each row). The estimated latent position matrix is thus organized like a traditional data table, with nodes corresponding to observations, and you could plot the rows as points in Euclidean space.

The Estimated Latent Position Matrix

Let's refresh ourselves on latent positions, and then explore how we can interpret an estimated latent position matrix.

Assuming you're viewing your network as some type of random dot product graph (remember that this can include SBMs, ER networks, and more), you can think of every node as being secretly associated with a position in Euclidean space. This position (relative to the positions associated with other nodes) tells you about the probability that one node will have an edge with another node.

Let's call the latent position matrix X. Remember that X has n rows (the number of nodes) and d columns (the number of dimensions). Although in practice you almost never know what the latent position matrix actually is, you can *estimate* it by embedding your network.

We're going to cheat a bit and use an embedding method called adjacency spectral embedding before we've discussed it, just to show what this looks like. In the next section, you'll learn how this embedding is happening, and what's going on under the hood. For now, just think of it as a way to estimate the latent positions for the nodes of a network and move from network space to Euclidean space.

Below we make a network, which in this case is an SBM. From the network, we can estimate a set of latent positions, where $n = 20$ rows for each node and $d = 2$ dimensions. Usually when something is an estimation for something else in statistics, you put a hat over it: $\hat{X}$. We'll do that here.

```python
from graspologic.simulations import sbm
from graspologic.embed import AdjacencySpectralEmbed as ASE
import numpy as np

# make a network
B = np.array([[0.8, 0.1],
              [0.1, 0.8]])
n = [10, 10]
A, labels = sbm(n=n, p=B, return_labels=True)

# embed
ase = ASE(n_components=2)
X = ase.fit_transform(A)
```

It's good to emphasize here that we're modeling our networks as *random dot-product graphs* (RDPGs). One implication of this is that we can think of our network

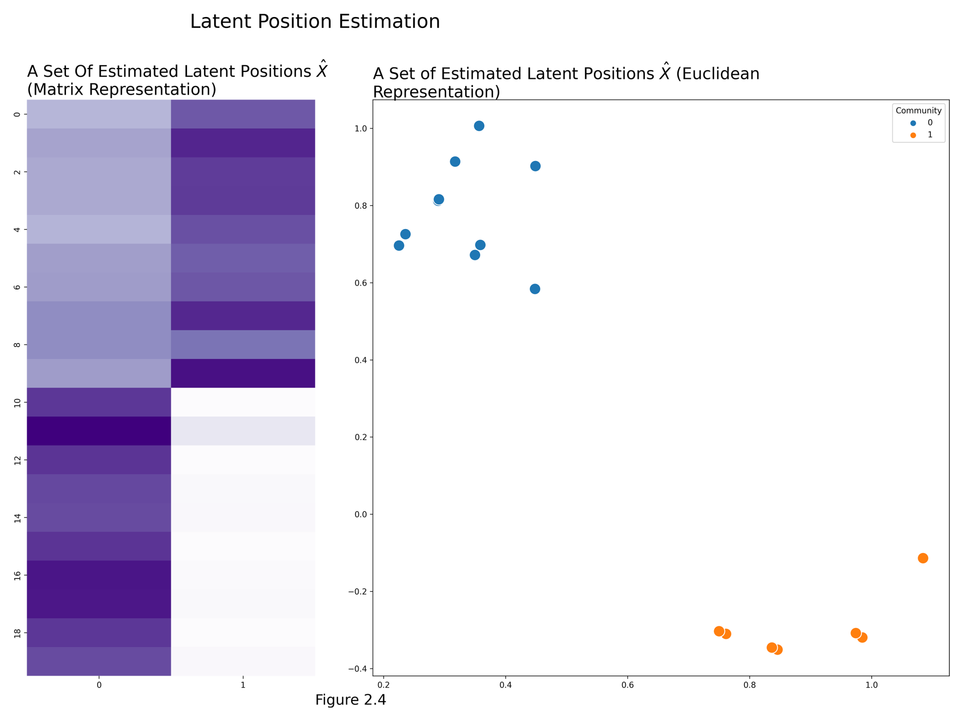


Figure 2-4. Latent Position Estimation

as having some underlying probability distribution, and any specific network is one of many possible realizations of that distribution. It also means that each edge in our network has some *probability* of existing: nodes 0 and 3, for instance, may or may not have an edge. The concept of a latent position only works under the assumption that the network is observed from an RDPG.

Edge Probability Matrix Estimation

We mentioned before that the relative locations of latent positions tell you about edge probabilities, but it's good to be a bit more specific. If you take the dot product (or the weighted sum) of row i of the latent position matrix X with row j, you'll get the probability that nodes i and j have an edge between them. Incidentally, this means that the dot product between any two rows of the latent position matrix has to be bound between 0 and 1.

Estimating a Block Probability Matrix From the Estimated Latent Positions

Similarly, you can estimate the block probability matrix P for your network using the latent position estimates. How would you generate $\hat{P}$ from $\hat{X}$?

Well, you'd just multiply it by its transpose: $\hat{P} = \hat{X}\hat{X}^\top$. This operation will take the dot product between every row of $\hat{X}$ and put it in the result. $(\hat{X}\hat{X}^\top)_{ij}$ will just be the dot product between rows i and j of the estimated latent position matrix (which is the estimated probability that nodes i and j will be connected). So, $\hat{X}\hat{X}^\top$ is just an estimate for the $n \times n$ block probability matrix.

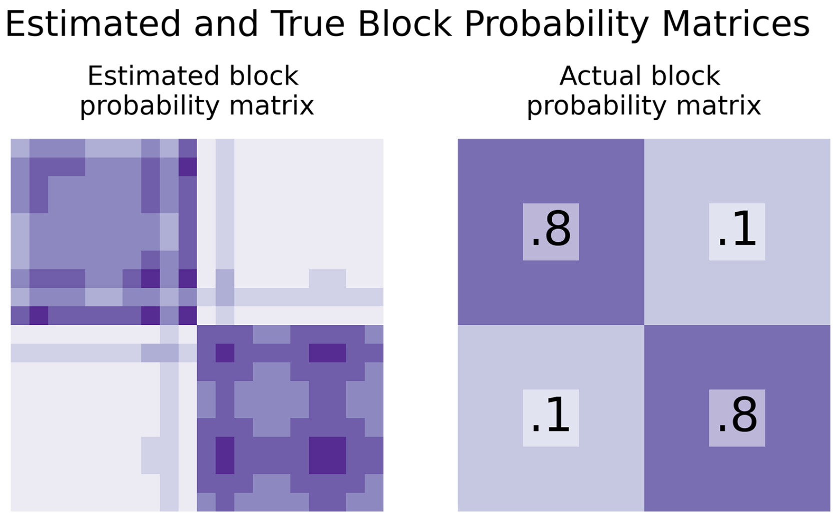


Figure 2.5

Figure 2-5. Estimated and True Block Probability Matrix

Thinking about Latent Positions Geometrically

You can also think about this stuff geometrically. The dot product between any two vectors u_i and u_j, geometrically, is their lengths multiplied together and then weighted by the cosine of the angle between them. Smaller angles have cosines close to 1, and larger angles have cosines close to 0. So, nodes whose latent positions have larger angles between them tend to have lower edge probabilities, and nodes whose latent positions have smaller angles between them tend to have higher edge probabilities. This is the core intuition you need to understand why you can find communities and do downstream inference with latent position matrices: two nodes whose latent positions are further apart will have a smaller probability of having an edge between them!

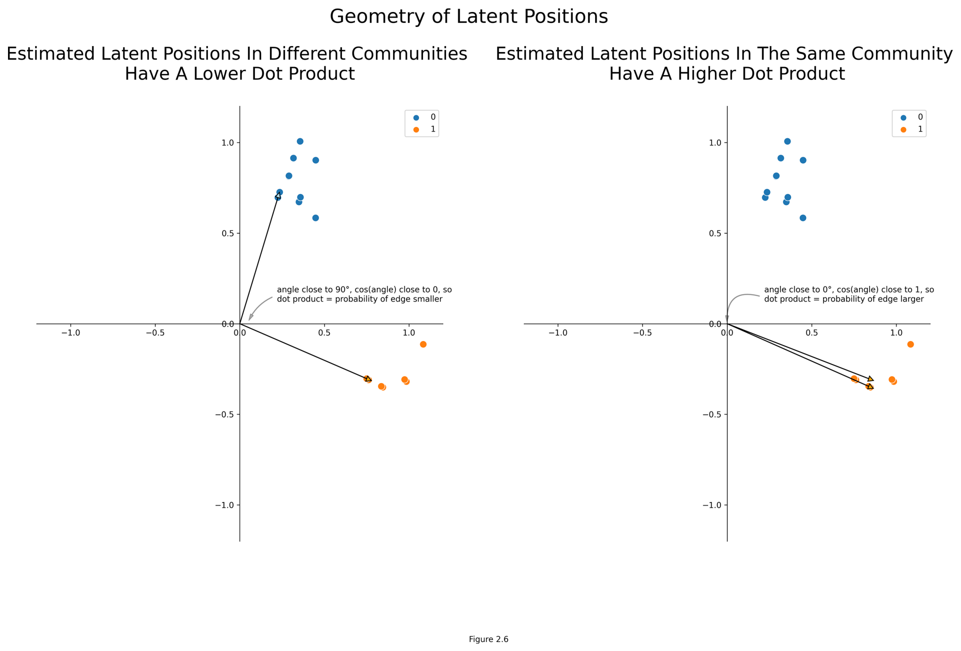


Figure 2-6. Geometry of Latent Position Estimates

Keep in mind that the two figures in fig 2-6 contain *estimates* for the latent positions, not the true ones. The true latent positions would have sets of point-masses in the same location for each community, since every node in an SBM in the same community has the same latent position. There's math that shows that you get a pretty good estimate for the block probability matrix using these estimates as well. In practice, that's what you're actually doing: getting an estimate of the latent positions with spectral embedding, then using those to do more downstream tasks or estimating block probability matrices.

Chapter 3

Spectral Embedding Methods

One of the primary embedding tools we'll use in this book is a set of methods called spectral embedding. You'll see spectral embedding and variations on it repeatedly, both throughout this section and when we get into applications, so it's worth taking the time to understand spectral embedding deeply. If you're familiar with Principal Component Analysis (PCA), this method has a lot of similarities. We'll need to get into a bit of linear algebra to understand how it works.

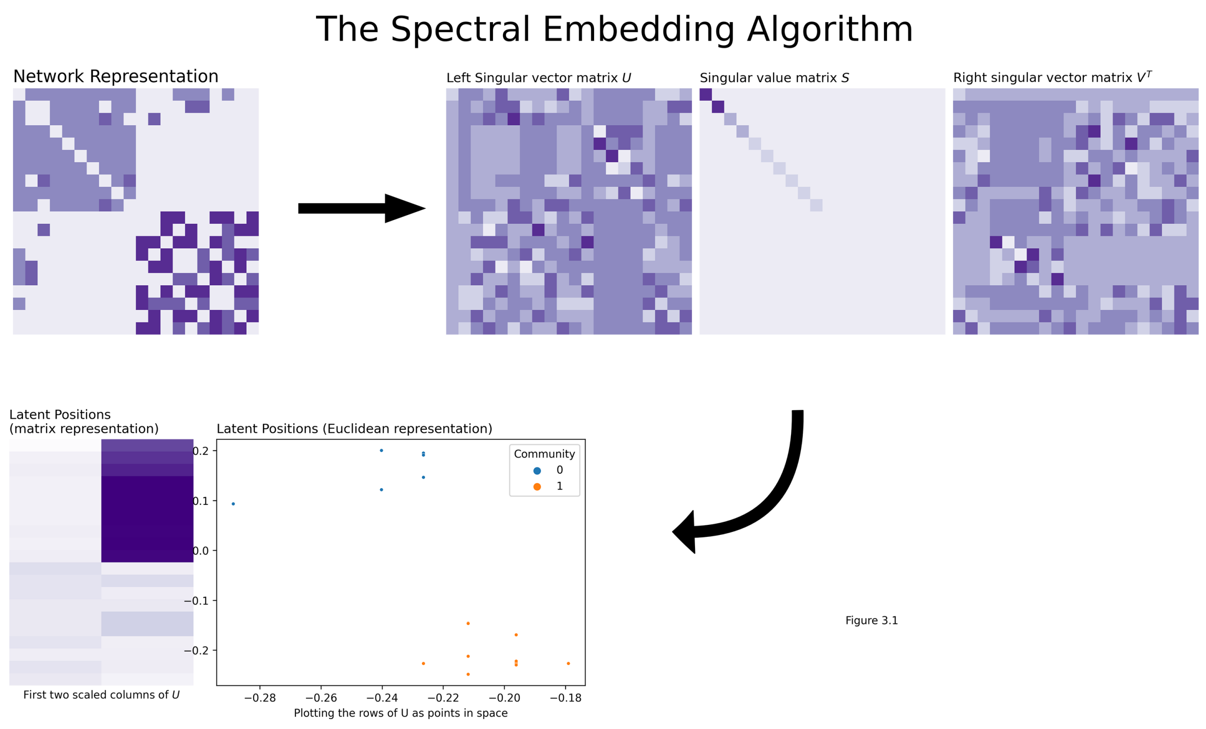
Remember that the basic idea behind any network embedding method is to take the network and put it into Euclidean space - meaning, a nice data table with rows as observations and columns as features (or dimensions), which you can then plot on an x-y axis. In this section, you'll see the linear algebra-centric approach that spectral embedding uses to do this.

Spectral methods are based on a bit of linear algebra, but hopefully a small enough amount to still be understandable. The overall idea has to do with eigenvectors, and more generally, something called "singular vectors" - a generalization of eigenvectors. It turns out that the biggest singular vectors of a network's adjacency matrix contain the most information about that network - and as the singular vectors get smaller, they contain less information about the network (we're glossing over what 'information' means a bit here, so just think about this as a general intuition). So if you represent a network in terms of its singular vectors, you can drop the smaller ones and still retain most of the information. This is the essence of what spectral embedding is about (here "biggest" means "the singular vector corresponding to the largest singular value").

If you don't know what singular values and singular vectors are, don't worry about it. You can think of them as a generalization of eigenvalues/vectors (it's also ok if you don't know what those are): all matrices have singular values and singular vectors, but not all matrices have eigenvalues and eigenvectors. In the case of square, symmetric matrices with positive eigenvalues, the eigenvalues/vectors and singular values/vectors are the same thing.

If you want some more background information on spectral theory, there are some explanations in the Math Refresher section in the introduction. They're an important set of vectors associated with matrices with a bunch of interesting properties. A lot of linear algebra is built around exploring those properties.

You can see visually how Spectral Embedding works in fig 3-1. We start with an observation of a 20-node stochastic block model with two communities, and then find its singular values and vectors. It turns out that because there are only two communities, only the first two singular vectors contain information – the rest are just noise! (you can see this if you look carefully at the first two columns of the eigenvector matrix). So, we took these two columns and scaled them by their singular values. The final embedding is that scaled matrix, and the plot you see takes the rows of that matrix and puts them into Euclidean space (an x-y axis) as points. This is all stuff we've seen before - we've created an estimate of the SBM's latent position matrix, and the embeddings for the nodes are the latent positions. Underneath the figure is a step-by-step explanation for spectral embedding.



The Spectral Embedding Algorithm
Network Representation
Left Singular vector matrix U
Singular value matrix S
Right singular vector matrix V^T
Latent Positions (matrix representation)
First two scaled columns of U
Latent Positions (Euclidean representation)
Community
0
1
Plotting the rows of U as points in space
Figure 3.1

Figure 3-1. The Spectral Embedding Algorithm

```
Algorithm 1: The Spectral Embedding Algorithm
1 Take a network's adjacency matrix. Optionally take its Laplacian as a network
   representation.
2 Decompose it into a a singular vector matrix, a singular value matrix, and the
   singular vector matrix's transpose.
3 Remove every column of the singular vector matrix except for the first $k$
   vectors, corresponding to the $k$ largest singular values.
4 Scale the $k$ remaining columns by their corresponding singular values to create
   the embedding.
5 The rows of the matrix we created are estimates for the latent positions for the
   nodes of the SBM we observed the network from.
```

We need to dive into a few specifics to understand spectral embedding better. We need to figure out how to find our network's singular vectors, for instance, and we also need to understand why those singular vectors can be used to form a representation of our network. To do this, we'll explore a few concepts from linear algebra like matrix rank, and we'll see how understanding these concepts connects to understanding spectral embedding.

Let's scale down and make a simple network, with only six nodes. We'll take its Laplacian just to show what that optional step looks like, and then we'll find its singular vectors with a technique we'll explore called singular value decomposition. Then, we'll explore why we can use the first k singular values and vectors to find an embedding. Let's start with creating the simple network.

Data Generation

Say we have the simple network below. There are six nodes total, numbered 0 through 5, and there are two distinct connected groups (called "connected components" in network theory land). Nodes 0 through 2 are all connected to each other, and nodes 3 through 5 are also all connected to each other.

```
from itertools import combinations
import numpy as np

def add_edge(A, edge: tuple):
    """
    Add an edge to an undirected graph.
    """
    i, j = edge
    A[i, j] = 1
    A[j, i] = 1
    return A

A = np.zeros((6, 6))

for edge in combinations([0, 1, 2], 2):
    add_edge(A, edge)

for edge in combinations([3, 4, 5], 2):
    add_edge(A, edge)
```

You can see the adjacency matrix and layout plot for the network in fig 3-2. Notice that there are two distinct blocks in the adjacency matrix: in its upper-left, you can see the edges between the first three nodes, and in the bottom right, you can see the edges between the second three nodes.

A Simple Network

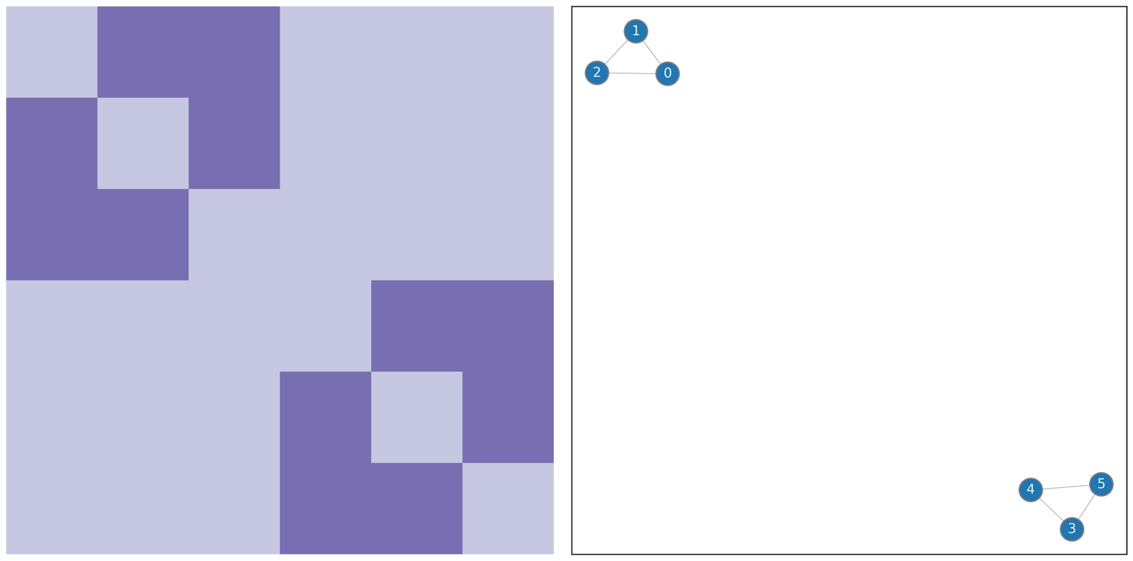


Figure 3.2

Figure 3-2. The Spectral Embedding Algorithm

The Laplacian Matrix

With spectral embedding, we'll either find the singular vectors of the Laplacian or the singular vectors of the Adjacency Matrix itself. Since we already have the adjacency matrix, let's take the Laplacian just to see what that looks like.

Remember from chapter four that there are a few different types of Laplacian matrices. By default, for undirected networks, Graspologic uses the normalized Laplacian $L = D^{-1/2}AD^{-1/2}$, where D is the degree matrix. Remember that the degree matrix has the degree, or number of edges, of each node along the diagonals. Variations on the normalized Laplacian are generally what we use in practice, but for simplicity and illustration, we'll just use the basic, cookie-cutter version of the Laplacian $L = D - A$.

Here is the degree matrix D.

```
# Build the degree matrix D
degrees = np.count_nonzero(A, axis=0)
D = np.diag(degrees)
```

```
# Build the Laplacian matrix L
L = D - A
```

Singular Vectors and Singular Value Decomposition

Now that we have a Laplacian matrix, we'll want to find its singular vectors. To do this, we'll need to use a technique called *Singular Value Decomposition*, or SVD.

SVD is a way to break a single matrix apart (also known as factorizing) into three distinct new matrices – In our case, the matrix we'll factorize will be the Laplacian we just built. These three new matrices correspond to the singular vectors and singular values of the original matrix: the algorithm will collect all of the singular vectors as columns of one matrix, and the singular values as the diagonals of another matrix.

In the case of the Laplacian (as with all symmetric matrices that have real, positive eigenvalues), remember that the singular vectors/values and the eigenvectors/values are the same thing. For more technical and generalized details on how SVD works, or for explicit proofs, we would recommend a Linear Algebra textbook [Trefethan, LADR]. Here, we'll look at the SVD with a bit more detail here in the specific case where we start with a matrix which is square, symmetric, and has real eigenvalues.

singular value decomposition Suppose you have a square, symmetrix matrix X with real eigenvalues. In our case, X corresponds to the Laplacian L (or the adjacency matrix A).

$$\begin{bmatrix} x_{11} & & & '' \\ & x_{22} & & \\ & & \ddots & \\ '' & & & x_{nn} \end{bmatrix}$$

Then, you can find three matrices - one which rotates vectors in space, one which scales them along each coordinate axis, and another which rotates them back - which, when you multiply them all together, recreate the original matrix X. This is the essence of singular value decomposition: you can break down any linear transformation into a rotation, a scaling, and another rotation. Let's call the matrix which rotates U (this type of matrix is called "orthogonal"), and the matrix that scales S.

$$X = USV^T$$

Since U is a matrix that just rotates any vector, all of its column-vectors are orthogonal (all at right angles) from each other and they all have the unit length of 1. These columns are more generally called the *singular vectors* of X. In some specific cases, these are also called the eigenvectors. Since S just scales, it's a diagonal matrix: there are values on the diagonals, but nothing (0) on the off-diagonals. The amount that each coordinate axis is scaled are the values on the diagonal entries of S, σ_i. These are *singular values* of the matrix X, and, also when some conditions are met, these are also the eigenvalues. Assuming our network is undirected, this will be the case with the Laplacian matrix, but not necessarily the adjacency matrix.

$$X = \begin{bmatrix} \uparrow & \uparrow & & \uparrow \\ u_1 & \vec{u}_2 & \dots & \vec{u}_n \\ \downarrow & \downarrow & & \downarrow \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix} \begin{bmatrix} \leftarrow & \vec{u}_1^T & \rightarrow \\ \leftarrow & \vec{u}_2^T & \rightarrow \\ & \vdots & \\ \leftarrow & \vec{u}_n^T & \rightarrow \end{bmatrix}$$

Breaking Down the Laplacian

Now we know how to break down any random matrix into singular vectors and values with SVD, so let's apply it to our toy network. We'll break down our Laplacian matrix into U, S, and $V^\top$. The Laplacian is a special case where the singular values and singular vectors are the same as the eigenvalues and eigenvectors, so we'll just refer to them as eigenvalues and eigenvectors from here on, since those terms are more common. For similar (actually the same) reasons, in this case $V^\top = U^\top$.

Here, the leftmost column of U (and the leftmost eigenvalue in S) correspond to the eigenvector with the highest eigenvalue, and they're organized in descending order (this is standard for Singular Value Decomposition). We decomposed the Laplacian with scipy's linalg module.

```
from scipy.linalg import svd
U, S, Ut = svd(L)
```

You can see the matrices below in fig 3-4.

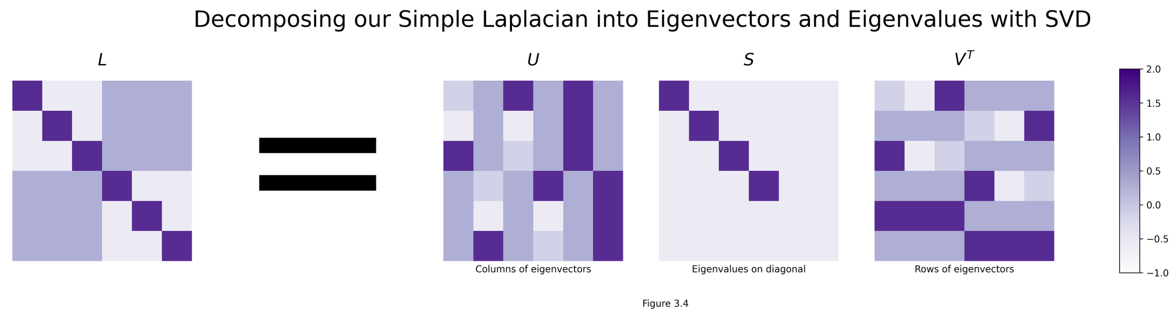


Figure 3-3. Decomposing our Simple Laplacian into Eigenvalues and Eigenvectors with SVD

So now we have a collection of eigenvectors organized into a matrix with U, and a collection of their corresponding eigenvalues organized into a matrix with S. Remember that with Spectral Embedding, we keep only the largest eigenvalues/vectors and "clip" columns off of U.

Why exactly do these matrices reconstruct our Laplacian when multiplied together? Why does the clipped version of U give us a lower-dimensional representation of our network? To answer that question, we'll need to start talking about a concept in linear algebra called the *rank* of a matrix.

The essential idea is that you can turn each eigenvector/eigenvalue pair into a low-information matrix instead of a vector and number. Summing all of these matrices lets you reconstruct L. Summing only a few of these matrices lets you get *close* to L. In fact, if you were to unwrap the two matrices into single vectors, the vector you get from summing the two is as close in Euclidean space as you possibly can get to L

given the information you deleted when you removed the smaller eigenvectors.

Let's dive into it!

Matrix Rank

When we embed anything to create a new representation, we're essentially trying to find a simpler version of that thing which preserves as much information as possible. This leads us to the concept of *matrix rank*.

Matrix Rank: The rank of a matrix X, defined $rank(X)$, is the number of linearly independent rows and columns of X.

At a very high level, we can think of the matrix rank as telling us just how "simple" X is. A matrix which is rank 1 is very simple: all of its rows or columns can be expressed as a weighted sum of just a single vector. On the other hand, a matrix which has "full rank", or a rank equal to the number of rows (or columns, whichever is smaller), is a bit more complex: no row nor column can be expressed as a weighted sum of other rows or columns.

There are a couple ways that the rank of a matrix and the singular value decomposition interact which are critical to understand: First, you can make a matrix from your singular vectors and values (eigenvectors and values, in our Laplacian's case), and summing all of them recreates your original, full-rank matrix. Each matrix that you add to the sum increases the rank of the result by one. Second, summing only a few of them gets you to the best estimation of the original matrix that you can get to, given the low-rank result. Let's explore this with a bit more depth.

We'll be using the Laplacian as our examples. For the adjacency matrix, this theory all still works, but you'd just have to replace $\vec{u}_i\vec{u}_i^\top$ with $\vec{u}_i\vec{v}_i^\top$ throughout (the adjacency matrices' SVD is $A = USV^\top$, since the right singular vectors might be different than the left singular vectors).

Sums of Rank One Matrices

You can actually create an $n \times n$ matrix using any one of the original Laplacian's eigenvectors $\vec{u}_i$ by taking its outer product $\vec{u}_i\vec{u}_i^T$. This creates a rank one matrix which only contains the information stored in the first eigenvector. Since we're using eigenvalues instead of singular values with the Laplacian, let's call an eigenvalue λ_i. Scale the matrix $\vec{u}_i\vec{u}_i^T$ by λ_i to make $\lambda_i\vec{u}_i\vec{u}_i^T$ and you have something that feels suspiciously similar to how we take the first few singular vectors of U and scale them in the spectral embedding algorithm.

It turns out that we can express the Laplacian (and in general any matrix, if we use $\vec{v}_i^\top$ instead of $\vec{u}_i^\top$) as the sum of all of these rank one matrices. Take $\vec{u}_i$, the i^{th} column of U and the i^{th} eigenvector of our Laplacian. Its corresponding eigenvalue is the i^{th} element of the diagonal eigenvalue matrix S. You can make a rank one matrix from this eigenvalue/eigenvector pair by taking the outer product and scaling the result by the eigenvalue: $\lambda_i\vec{u}_i\vec{u}_i^T$.

It turns out that when we take the sum of all of these rank 1 matrices–each one corresponding to a particular eigenvalue/eigenvector pair–we'll recreate the original matrix.

$$X = \sum_{i=1}^{n} \lambda_i \vec{u}_i \vec{u}_i^T = \lambda_1 \begin{bmatrix} \uparrow \\ \vec{u}_1 \\ \downarrow \end{bmatrix} \begin{bmatrix} \leftarrow & \vec{u}_1^T & \rightarrow \end{bmatrix} + \lambda_2 \begin{bmatrix} \uparrow \\ \vec{u}_2 \\ \downarrow \end{bmatrix} \begin{bmatrix} \leftarrow & \vec{u}_2^T & \rightarrow \end{bmatrix} + ... + \lambda_n \begin{bmatrix} \uparrow \\ \vec{u}_n \\ \downarrow \end{bmatrix} \begin{bmatrix} \leftarrow & \vec{u}_n^T & \rightarrow \end{bmatrix}$$

Here are all of the $\lambda_i\vec{u}_i\vec{u}_i^T$ for our Laplacian L. Since there were six nodes in the original network, there are six eigenvalue/vector pairs, and six rank 1 matrices.

```
n_nodes = U.shape[0]

# For each eigenvector/value,
# find its outer product,
# and append it to a list.
low_rank_matrices = []
for node in range(n_nodes):
    ui = np.atleast_2d(U[:, node]).T
    low_rank_matrix = S[node] * ui @ ui.T
    low_rank_matrices.append(low_rank_matrix)

# Take the elementwise sum of every matrix in the list.
laplacian_sum = np.array(low_rank_matrices).sum(axis=0)
```

You can see the result of the sum in fig 3-5 below. On the left are all of the low-rank matrices - one corresponding to each eigenvector - and on the right is the sum of all of them. You can see that the sum is just our Laplacian!

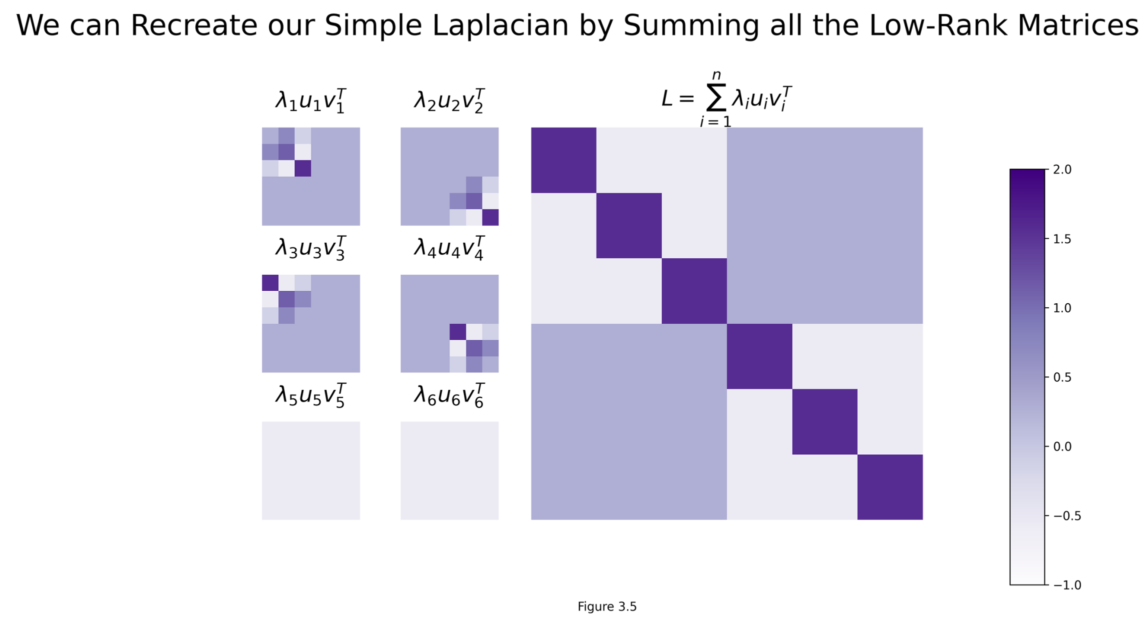


Figure 3-4. We can Recreate our Simple Laplacian by Summing all the Low-Rank Matrices

Next up, we'll estimate the Laplacian by only taking a few of these matrices. You can already kind of see in the figure above that this'll work - the last two matrices don't even have anything in them (they're just 0)!

Laplacian Approximation Through Summation

When you sum the first few of these low-rank $\lambda_i u_i u_i^T$, you can approximate your original matrix.

This tells us something interesting about spectral embedding: the information in the first few eigenvectors of a high rank matrix lets us find a more simple approximation to it. You can take a matrix that's extremely complicated (high-rank) and project it down to something which is much less complicated (low-rank).

Look at fig 3-6. In each plot, we're summing more and more of these low-rank matrices. By the time we get to the fourth sum, we've totally recreated the original Laplacian.

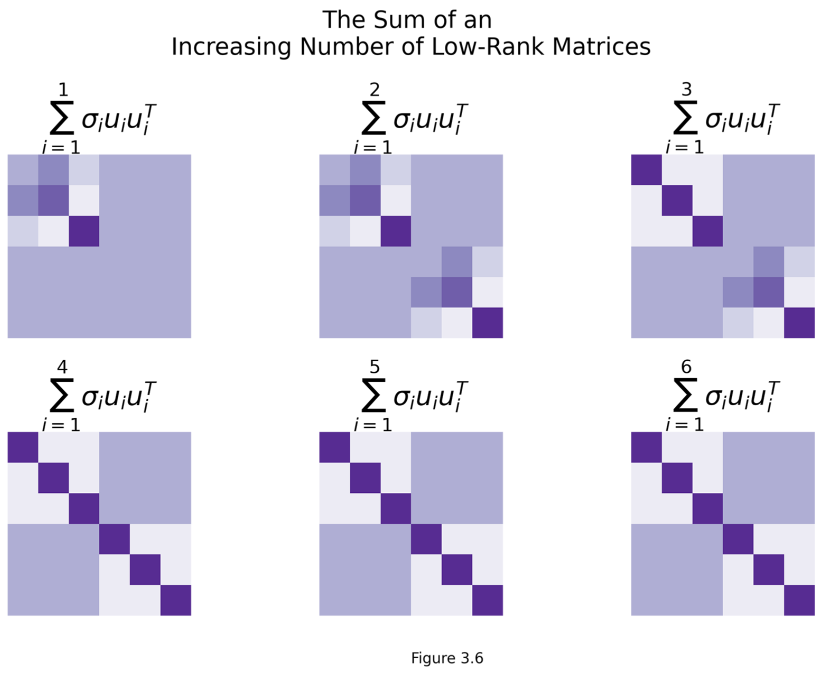


Figure 3-5. The Sum of an Increasing Number of Low-Rank Matrices

Increased Usefulness of Approximation with Larger Networks

This becomes even more useful when we have huge networks with thousands of nodes, but only a few communities. It turns out, especially in this situation, we can usually sum a very small number of low-rank matrices and get to an excellent approximation for our network that uses much less information.

Take the network below, for example. It's generated from a stochastic block model with 1000 nodes total (500 in one community, 500 in another). We took its normalized Laplacian (remember that this means $L = D^{-1/2}AD^{-1/2}$), decomposed it, and summed the first two low-rank matrices that we generated from the eigenvector columns.

The result is not exact, but it looks pretty close. And we only needed the information from the first two singular vectors instead of all of the information in our full $n \times n$ matrix!

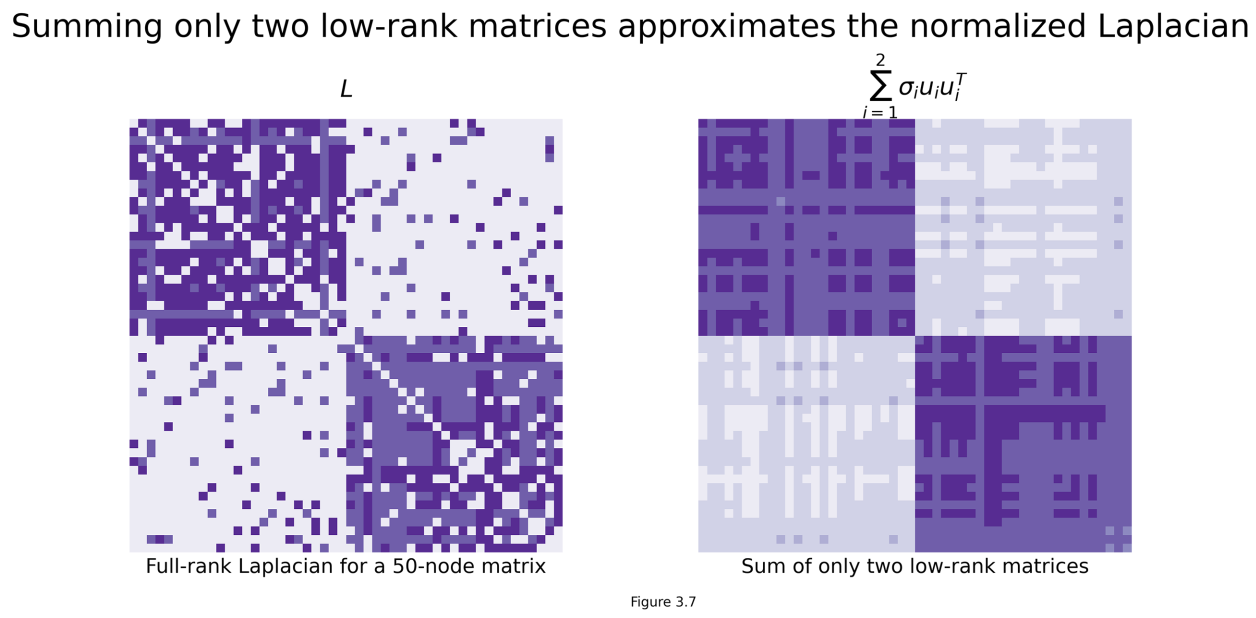


Figure 3-6. The Sum of an Increasing Number of Low-Rank Matrices

This is where a lot of the power of an SVD comes from: you can approximate extremely complicated (high-rank) matrices with extremely simple (low-rank) matrices.

Matrix Rank and Spectral Embedding

Remember the actual spectral embedding algorithm: we take a network, decompose it with Singular Value Decomposition into its singular vectors and values, and then cut out everything but the top k singular vector/value pairs. Once we scale the columns of singular vectors by their corresponding values, we have our embedding. That embedding is an estimate of the latent position matrix.

Let's go back to our original, small (six-node) network and estimate the latent position matrix from it. We'll embed down to three dimensions.

```
k = 3
U_cut = U[:, :k]
```

```
E_cut = E[:k]

latents_small = U_cut @ np.diag(E_cut)
```

You can see what the estimated latent position matrix we just made looks like in fig 3-8.

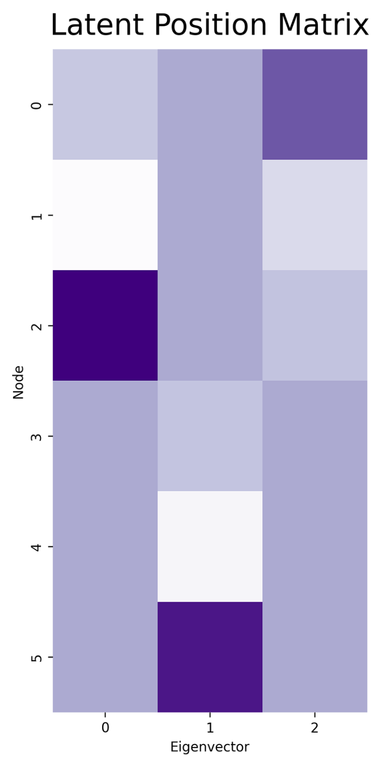


Figure 3.8

Figure 3-7. Estimated Latent Position Matrix

How does all this stuff about matrix rank help us understand spectral embedding?

Well, each column of the latent position matrix for the Laplacian is the i^{th} eigenvector scaled by the i^{th} eigenvalue: $\lambda_i \vec{u_i}$. If we right-multiplied one of those columns by its unscaled transpose $\vec{u_i}^\top$, we'd have one of our rank one matrices. This means that you can think of our rank-one matrices as essentially just fancy versions of the columns of an estimated latent position matrix (our embedding). They contain all the same information - they're just matrices instead of vectors!

In fact, you can express the sum we did earlier - our lower-rank estimation of L

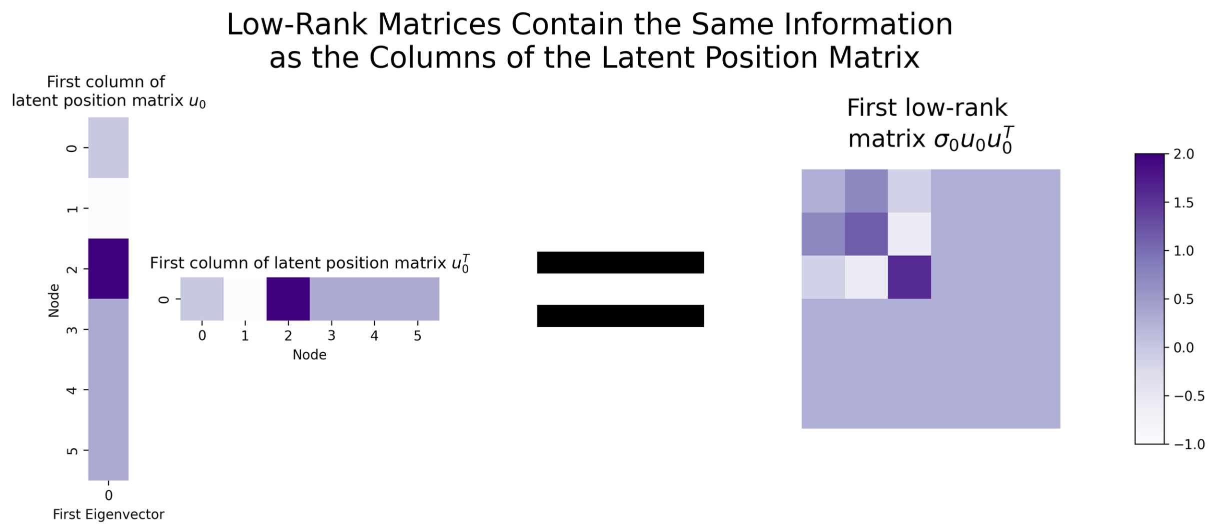


Figure 3.9

Figure 3-8. Low-Rank Matrices Contain the Same Information as the columns of the Latent Position Matrix

- with just our estimated latent position matrix! Remember that U_k is the first k eigenvectors of our Laplacian, and S_k is the diagonal matrix with the first k eigenvalues (and that we named them λ_1 through λ_k). In fig 3-10 below, you can see the same result as before – the sum of the first two low-rank matrices – but created an entirely different way.

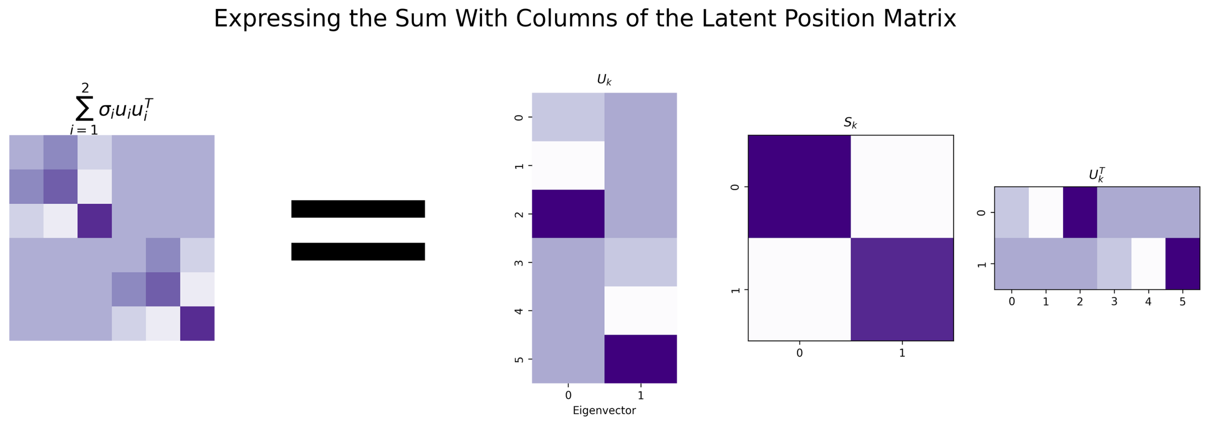


Figure 3.10

Figure 3-9. Expressing the Sum with Columns of the Estimated Latent Position Matrix

This helps gives an intuition for why our estimated latent position matrix gives a representation of our network. You can take columns of it, turn those columns into matrices, and sum those matrices, and then estimate the Laplacian for the network. That means the columns of our embedding network contain all of the information necessary to estimate the network!

Dimensionality Estimation

One thing we haven't addressed is how to figure out how many dimensions to embed our network down to. We've generally been embedding into two dimensions throughout this chapter (mainly because it's easier to visualize), but you can embed into as many dimensions as you want.

If you don't have any prior information about the "true" dimensionality of your latent positions, by default you'd just be stuck guessing. Fortunately, there are some rules-of-thumb to make your guess better, and some methods people have developed to make fairly decent guesses automatically.

The most common way to pick the number of embedding dimensions is with something called a scree plot. Essentially, the intuition is this: the top singular vectors of an adjacency matrix contain the most useful information about your network, and as the singular vectors have smaller and smaller singular values, they contain less information and so are less important (this is why it's reasonable to cut out the smallest $n - k$ singular vectors in the spectral embedding algorithm).

The scree plot just plots the singular values by their indices: the first (biggest) singular value is in the beginning, and the last (smallest) singular value is at the end.

You can see the scree plot for the Laplacian we made earlier in fig 3-11. We're only plotting the first ten singular values for demonstration purposes.

You'll notice that there's a marked area called the "elbow". This is an area where singular values stop changing in magnitude as much when they get smaller: before the elbow, singular values change rapidly, and after the elbow, singular values barely

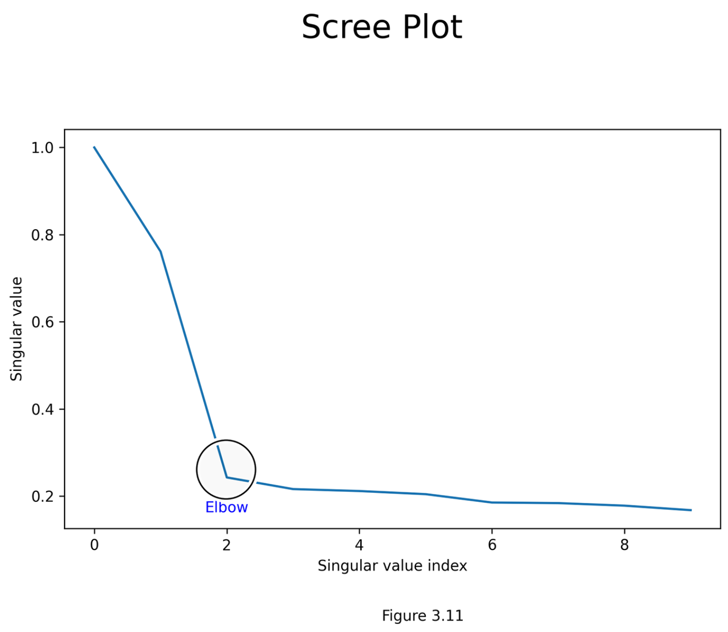


Figure 3-10. Scree Plot

change at all. (It's called an elbow because the plot kind of looks like an arm, viewed from the side!)

The location of this elbow gives you a rough estimate for how many "true" dimensions the true, underlying latent positions have. The singular values after the elbow are quite close to each other and have singular vectors which are largely noise, and don't tell you very much about your data. It looks from the scree plot that we should be embedding down to two dimensions, and that adding more dimensions would probably just mean adding noise to our embedding.

One drawback to this method is that a lot of the time, the elbow location is pretty subjective - real data will rarely have a nice, pretty elbow like the one you see above. The advantage is that it still generally works pretty well; embedding into a few more dimensions than you need isn't too bad, since you'll only have a few noise dimensions and there still may be *some* signal there.

In any case, Graspologic automates the process of finding an elbow using a popular method developed in 2006 by Mu Zhu and Ali Ghodsi at the University of Waterloo.

We won't get into the specifics of how it works here, but you can usually find fairly good elbows automatically.

Using Graspologic to embed networks

It's pretty straightforward to use graspologic's API to embed a network. The setup works like an SKlearn class: you instantiate an AdjacencySpectralEmbed class, and then you use it to transform data. You set the number of dimensions to embed to (the number of eigenvector columns to keep!) with 'n_components'.

Adjacency Spectral Embedding

Adjacency Spectral Embedding (ASE) performs the spectral embedding on the adjacency matrix. ASE is notable for directly estimating the latent position matrix in its purest form: as the matrix X such that $XX^\top$ estimates the block probability matrix P. You can perform ASE in Graspologic with the AdjacencySpectralEmbed class, shown below.

```python
from graspologic.embed import AdjacencySpectralEmbed as ASE

# Generate a network from an SBM
B = np.array([[0.8, 0.1],
              [0.1, 0.8]])
n = [25, 25]
A, labels = sbm(n=n, p=B, return_labels=True)

# Instantiate an ASE model and find the embedding
ase = ASE(n_components=2)
embedding = ase.fit_transform(A)
```

You can see the results of this embedding below, in fig 3-12.

Laplacian Spectral Embedding

Laplacian Spectral Embedding (LSE) works exactly the same as ASE, except it does SVD on the Laplacian matrix rather than the adjacency matrix. You can see the results of using Graspologic for this embedding below.

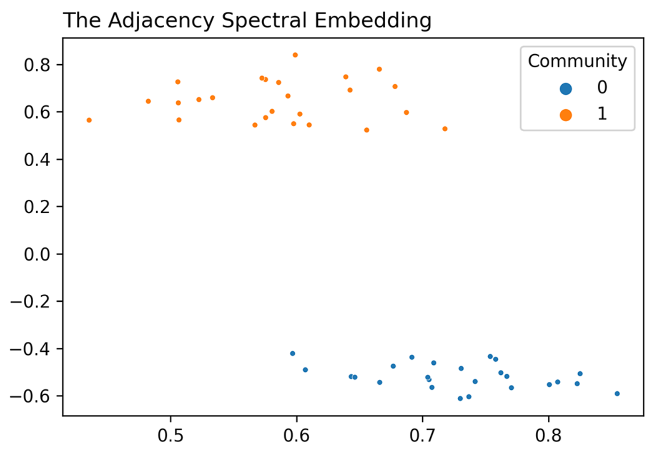


Figure 3.12

Figure 3-11. The Adjacency Spectral Embedding

```
from graspologic.embed import LaplacianSpectralEmbed as LSE

embedding = LSE(n_components=2).fit_transform(A)
```

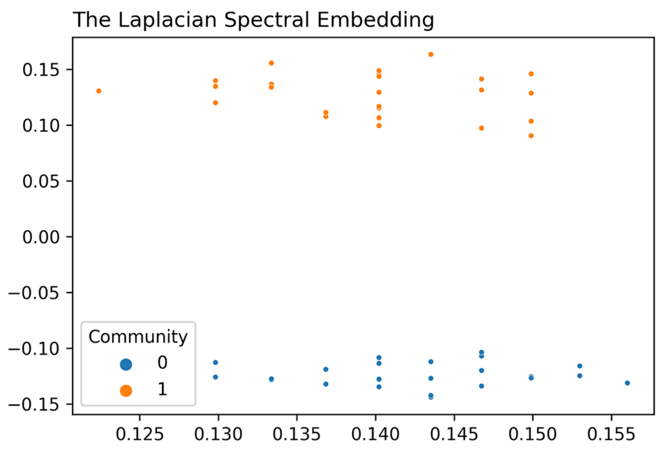


Figure 3.13

Figure 3-12. The Laplacian Spectral Embedding

The Two-Truths Phenomenon

Throughout this article, we've primarily used LSE, since Laplacians have some nice properties (such as having singular values being the same as eigenvalues) that make stuff like SVD easier to explain. However, you can embed the same network with either ASE or LSE, and you'll get two different (but equally true) embeddings.

Since both embeddings will give you a reasonable clustering, how are they different? When should you use one compared to the other?

Well, it turns out that LSE and ASE capture different notions of "clustering". Carey Priebe and collaborators at Johns Hopkins University investigated this recently - in 2018 - and discovered that LSE lets you capture "affinity" structure, whereas ASE lets you capture "core-periphery" structure (their paper is called "On a two-truths phenomenon in spectral graph clustering" - it's an interesting read for the curious). The difference between the two types of structure is shown in fig 3-14 below.

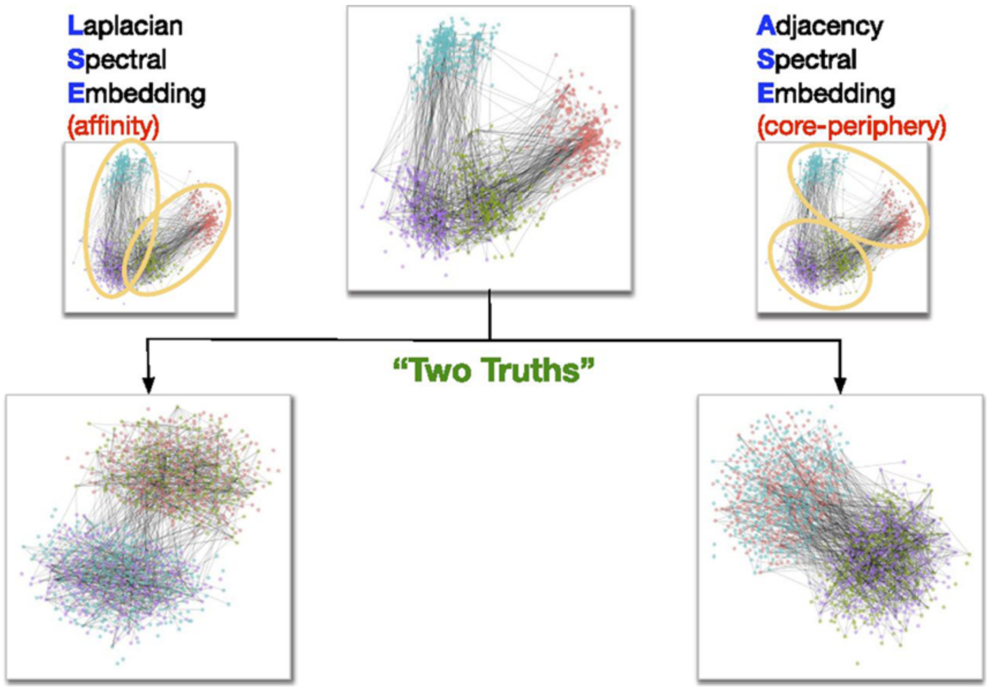


Figure 3-13. Affinity vs. Core-Periphery Structure

Chapter 4

Multiple-Network Representation Learning

Say you're a brain researcher, and you have a bunch of scans of brains - some are scans of people, and some are scans of aliens. You have some code that estimates networks from your scans, so you turn all your scans into networks. The nodes represent the brain regions which are common to both humans and aliens (isn't evolution amazing?), and the edges represent communication between these brain regions. You want to know if the human and alien networks share a common grouping of regions (your research topic is titled, "Do Alien Brains Have The Same Hemispheres That We Do?"). What do you do? How do you even deal with situations in which you have a lot of networks whose nodes all represent the same objects, but whose edges might come from totally different distributions?

Well, if your goal is to find the shared grouping of regions between the human and alien networks, you could try embedding your networks and then seeing what those embeddings look like. This would serve the dual purpose of having less stuff to deal with and having some way to directly compare all of your networks in the same space. Finding an embedding is also simply useful in general, because embedding a network or group of networks opens the door to machine learning methods designed for tabular data.

Data Generation

For example, say you have four alien networks and four human networks. Since alien brain networks aren't currently very accessible, we'll just simulate our human and alien networks with Stochastic Block Models. The communities that we're trying to group all of the brain regions into are the two hemispheres of the brain. We'll design the human brains to have strong connections within hemispheres, and we'll design the alien brains to have strong connections between hemispheres – but the same regions still correspond to the same hemispheres.

we'll use a relatively small number of nodes and fairly small block probabilities. You can see the specific parameters in the code below.

```python
import numpy as np
from graspologic.simulations import sbm

# Generate networks from an SBM, given some parameters
def make_sbm(*probs, n=100, return_labels=False):
    pa, pb, pc, pd = probs
    P = np.array([[pa, pb],
                  [pc, pd]])

    return sbm([n, n], P, return_labels=return_labels)

# make nine human networks
# and nine alien networks
p1, p2, p3 = .12, .06, .03
n = 100
labels = [0]*n + [1]*n
humans = [make_sbm(p1, p3, p3, p1, n=n) for i in range(4)]
aliens = [make_sbm(p3, p1, p1, p3, n=n) for i in range(4)]
```

The human and alien networks come from very different distributions. As you can see from the stochastic block model structure in fig 4-1, the regions in the human and the alien brains can both be separated into two communities. These communities represent the two hemispheres of the brain (who knew aliens also have bilateralized brains!). Although both humans and aliens have the same regions belonging to their respective hemispheres, as we planned, the alien networks have a strange property: their brain regions have more connections with regions in the opposite hemisphere than the same one.

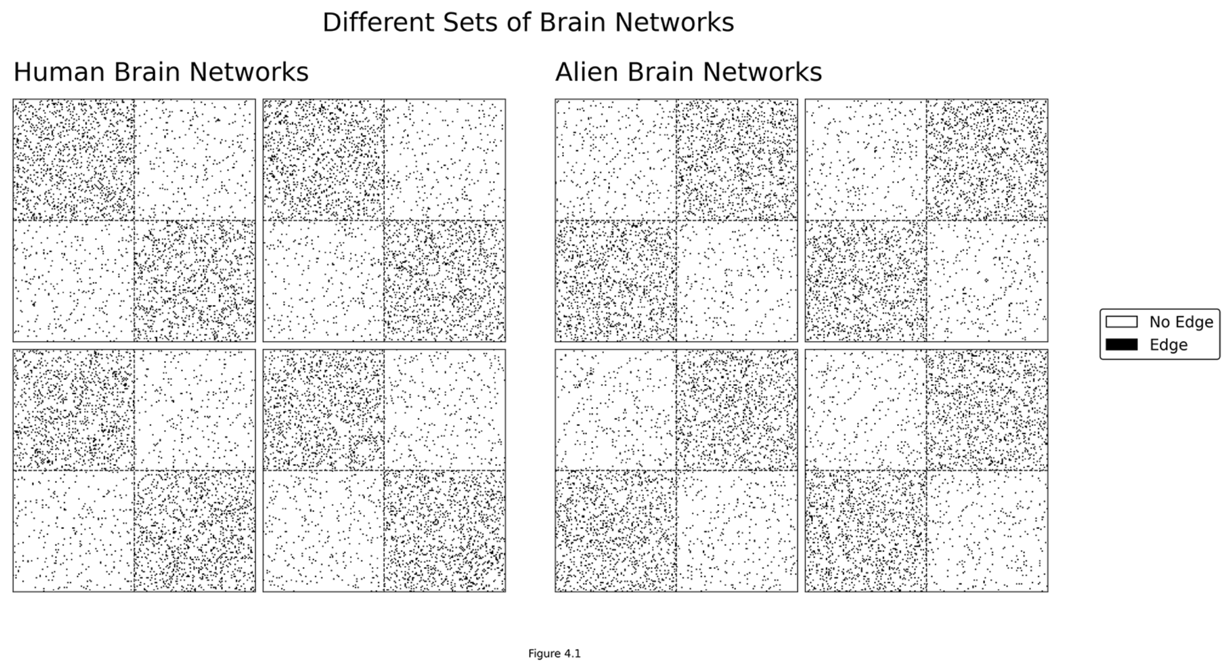


Figure 4-1. Different Sets of Brain Networks

Simple Embedding Methods with Multiple Networks

Remember, our goal is to find community structure common to both humans and aliens, and in our case that community structure is the brain hemispheres. We're going to try to to embed our brain networks into some lower-dimensional space - that way, we can use standard clustering methods from machine learning to figure out which regions are grouped. Try to think about how you might find a lower-dimensional embedding where the location of each node's estimated latent positions uses information from all of the networks.

Averaging Separately

The first idea you might come up with is to average your networks together, and then embed the result of that averaging with Spectral Embedding. It turns out that this is actually the right idea in the very special case where all of your networks come from the same probability distribution. In our case, we'll try averaging our groups of networks separately: we'll treat the human networks as one group, and the alien networks as another group, and we'll average each independently. In the end, we'll have two separate embeddings.

```
from graspologic.embed import AdjacencySpectralEmbed as ASE

# Compute the average adjacency matrix for
# human brains and alien brains
human_mean_network = np.array(humans).mean(axis=0)
alien_mean_network = np.array(aliens).mean(axis=0)

# Embed both matrices
ase = ASE(n_components=2)
human_latents = ase.fit_transform(human_mean_network)
alien_latents = ase.fit_transform(alien_mean_network)
```

Below, in fig 4-2, you can see what happens when we embed the averaged human and alien networks separately. Like all of our embedding plots, each dot represents the estimated latent position for a particular node.

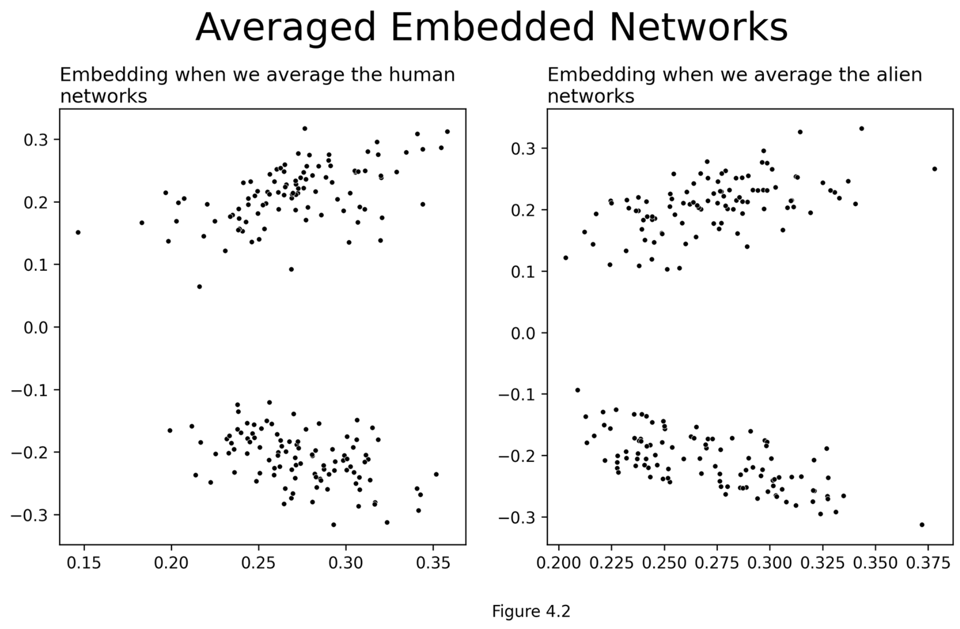


Figure 4-2. Averaged Embedded Networks

Both of these embeddings have clear clustering: there are two communities of nodes in both the human and the alien networks. We can recover the labels for these communities fairly easily using our pick of unsupervised clustering method. We know that the estimated latent positions in each community of an Adjacency Spectral Embedding are normally distributed under this simulation setting, and we have two communities. That means that the above embeddings are distributed according to a Gaussian Mixture. Here, "Gaussian" just means "normal", and a gaussian mixture just means that we have groups of normally distributed data clusters. As a result, it makes sense to cluster these data using scikit-learn's GaussianMixture implementation.

```python
from sklearn.mixture import GaussianMixture as GMM

# Predict labels for the human and alien brains
human_labels = GMM(n_components=2).fit_predict(human_latents)
alien_labels = GMM(n_components=2).fit_predict(alien_latents)
```

You can see a plot that predicts our community structure below. Success! When we embed the human and the alien networks separately, averaging them clearly lets us cluster the brain regions by hemisphere. However, as you can see, the colors are flipped: the communities are in different places relative to each other. This is because the alien networks are drawn from a different distribution than the human networks.

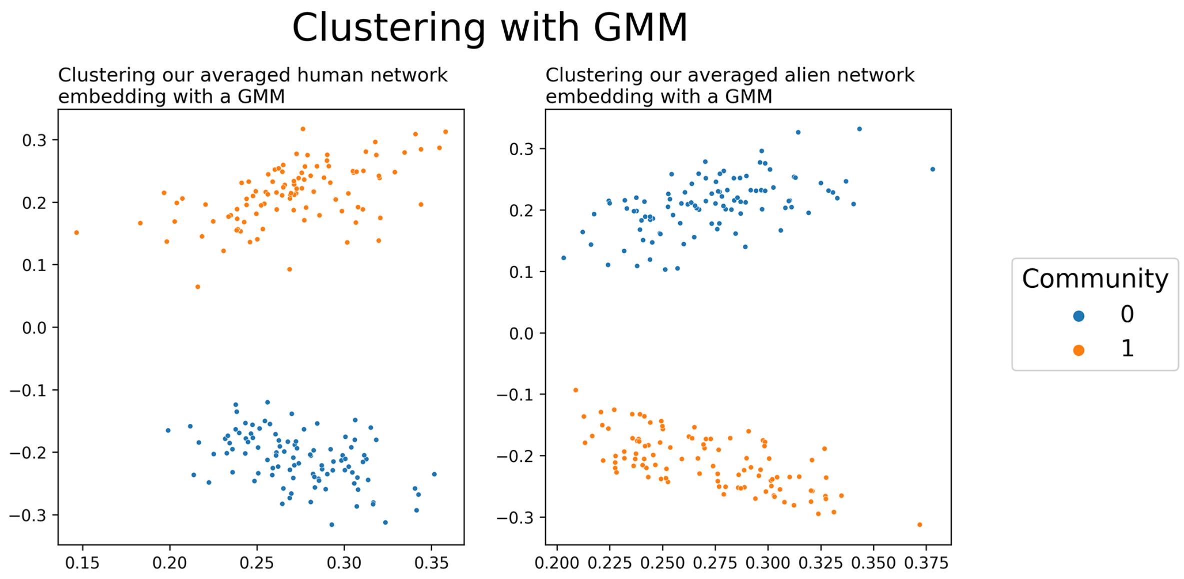


Figure 4.3

Figure 4-3. Clustering with GMM

Averaging Together

But what if you wanted to embed *all* of the networks into the same space, both the human and the alien networks, so that there's only one plot? Let's try it. We'll take all of the networks and then average them together, and then do an Adjacency Spectral Embedding. This will result in a single plot, with each point representing a single brain region. Do you think we'll still find this nice community separation?

```
total_mean_matrix = np.array(humans + aliens).mean(axis=0)
all_latents = ase.fit_transform(total_mean_matrix)
```

You can see what happens when we do this in fig 4-4.

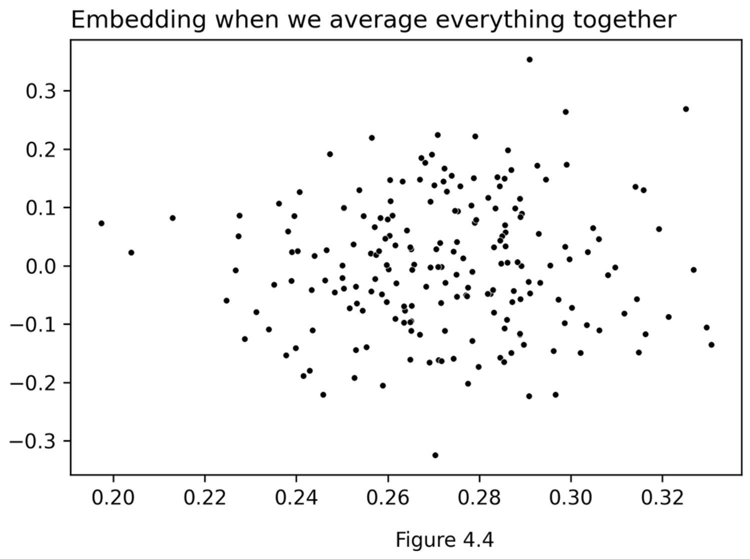


Figure 4-4. Embedding when we average everything together

Nope, bummer. Our community separation into discrete hemispheres is gone - the human networks and the alien networks cancelled each other out. As far as anybody can tell, our estimated latent positions have just become meaningless noise, so we can't cluster and find communities like we did before.

Why Did Averaging Together Fail?

Why did this happen? Well, let's go back and compare one human brain network with one alien brain network. Fig 4-5 shows a single one of our human brain networks compared with a single alien network.

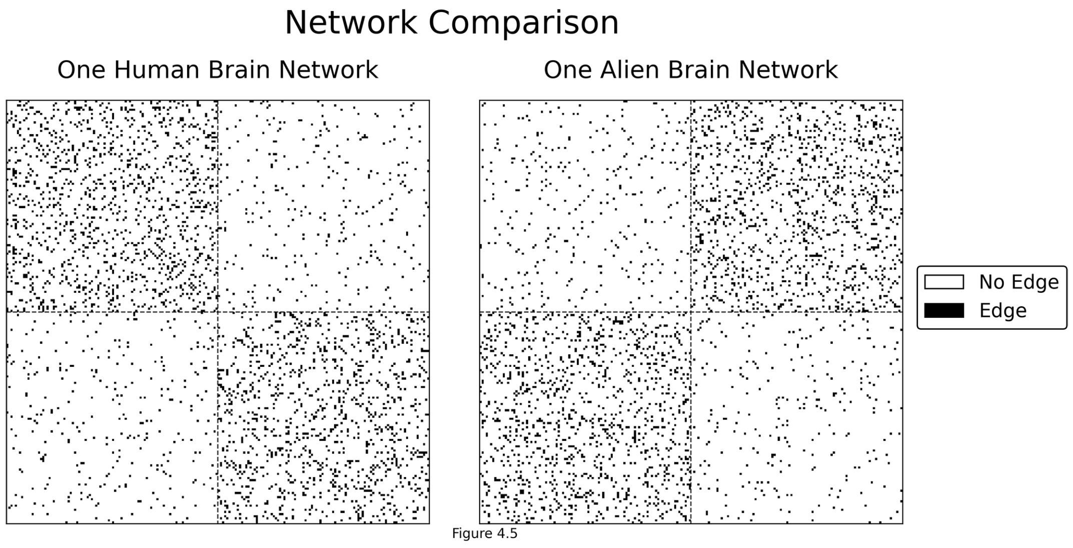


Figure 4-5. Network Comparison

The human network has more edges in the upper-left and lower-left quadrants of the heatmap. This implies that two regions in the same hemisphere are more likely to be connected for humans than two regions in opposite hemispheres.

The alien network tells a different story. For aliens, two regions in opposite hemispheres are more likely to be connected than two regions in the same hemisphere.

But what happens when you average these two adjacency matrices together?

```
combined = np.array([humans[0], aliens[0]])
averaged = np.mean(combined, axis=0)
```

By averaging, we've lost all of the community structure used to exist. That's why our big averaged embedding failed.

We've just discovered that even though it's often a great idea to simply average all of your networks together - for example, if they were drawn from the same distribution

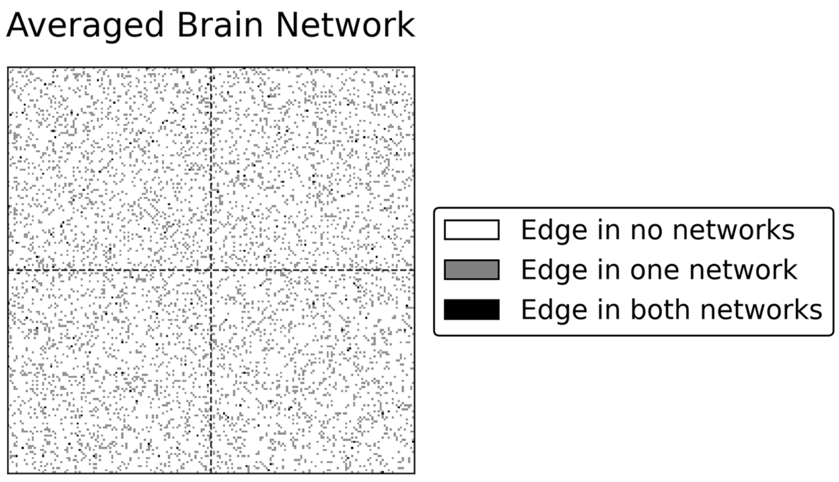


Figure 4.6

Figure 4-6. Averaged Brain Network

- it's often a horrible idea to average all of your networks if they might come from different distributions. This is a case of averaging networks which are "heterogeneous": Not only are your networks slightly different, but they're *should* to be different because their edge probabilities aren't the same. Sampling a lot of heterogeneous networks and then averaging them, as you can see from our exploration above, can result in losing the community signal you might have had.

We'd like to find a way to compare these heterogeneous networks directly, so that we can embed all of our networks into the same space and still keep that nice community structure. Figuring out the best way to do this is a topic under active research, and the set of techniques and tools that have developed as a result are together called multiple-network representation learning.

Different Types of Multiple-Network Representation Learning

Let's take a moment to explore some of the possible general approaches we could take in multiple-network representation learning. At some point we need to combine the many individual representations of our networks into one, and there are at least three possible places where we could do this: combining the networks together, combining the networks separately, and combining the embeddings. Each of these eventually results in an estimated latent position representation for our networks. It's important to note that in all of these approaches, we're simply learning representations for our groups of networks. You can do whatever you want with these representations; in our case, we'll illustrate that we can use them to classify our nodes.

Combining the Networks Together

With this approach, you'll start with a set of networks, and then you'll combine them all into a single network prior to doing anything else. You can then embed and classify this network directly. What we did before, averaging the human and alien networks, was an example of combining our networks – we just averaged all of our adjacency matrices, and then we embedded the result.

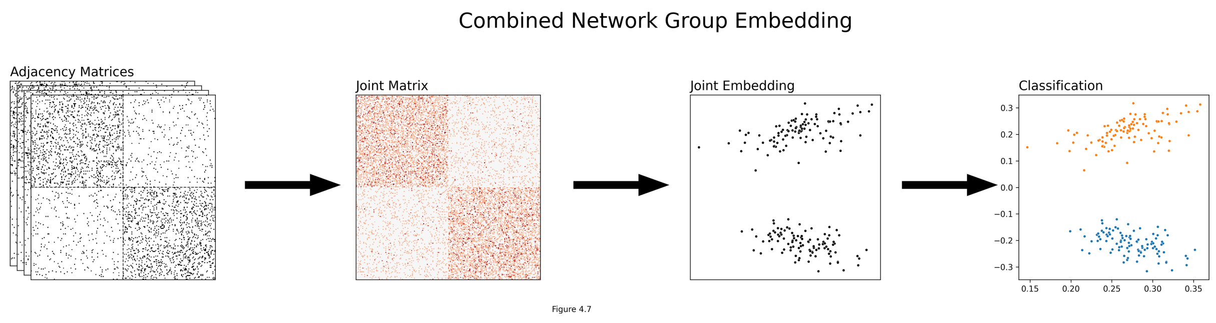


Figure 4-7. Combined Network Group Embedding

Combining the Networks Separately

The above approach is nice for collapsing our information into a single embedding – with each point in our final embedding representing a single node of our network. However, there are situations in which we might want to keep our embeddings separate, but make sure that they're in the same latent space – meaning, the embeddings aren't rotations of each other. That way, we can directly compare the embeddings of our separate embeddings.

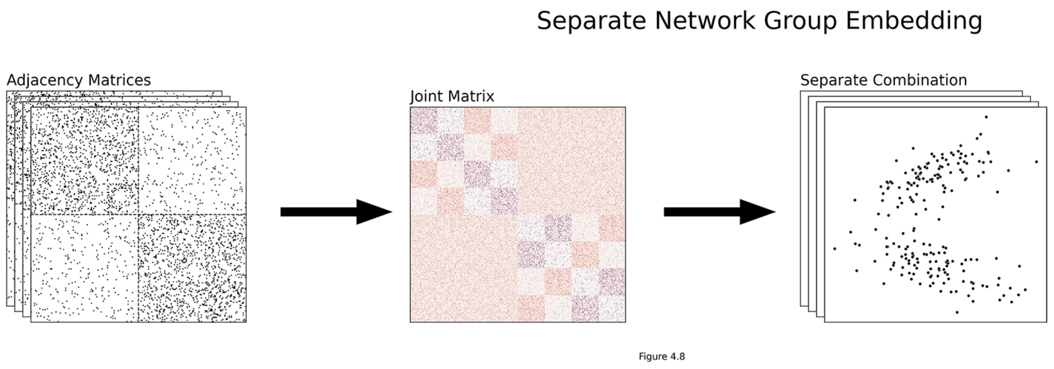


Figure 4-8. Separate Network Group Embedding

Combining the Embeddings

The final approach to multiple-network representation learning that we'll talk about is combining the embeddings themselves. With this approach, you're waiting until you've already embnedded all of your networks separately before you combine them, either with Adjacency Spectral Embedding or with some other single-network embedding method. Multiple Adjacency Spectral Embedding, which we'll be talking about soon, is an example of this approach.

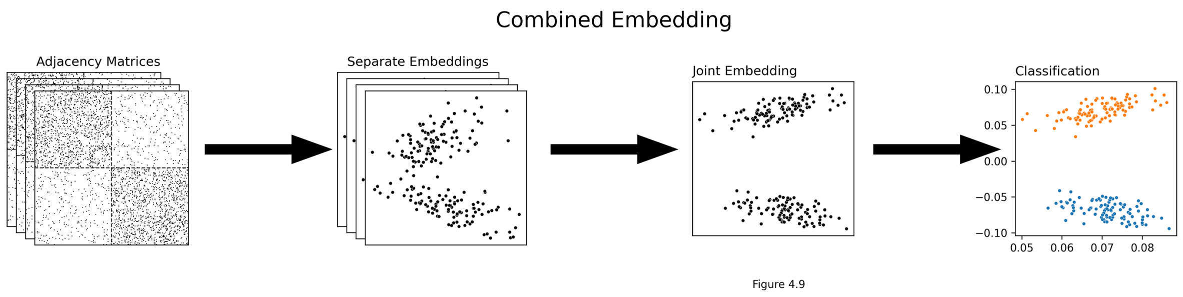


Figure 4-9. Combined Embedding

For the rest of this section, we'll explore the strengths and weaknesses of different particular techniques which use these approaches. The first we'll look at is combines the embeddings, like above. It's called Multiple Adjacency Spectral Embedding, or MASE for short.

Multiple Adjacency Spectral Embedding

MASE is a technique which combines embeddings by concatennating and re-embedding the separate latent position estimates into a single space. It's nice because you don't actually need each network to be generated from the same distribution - you only need the nodes of the different networks to be aligned and for them to belong to the same communities.

MASE is probably the easiest to understand if you know how Adjacency Spectral Embeddings work. Say you have some number of networks, and (like we said above) their nodes are aligned. The goal of MASE is to embed the networks into a single space, with each point in that space representing a single node - but, unlike simply averaging, MASE lets you combine networks which aren't necessarily drawn from the same distribution. MASE is based on the common subspace independent-edge (COSIE) model from the multi-network models section of chapter 5, so we're operating under the assumption that there *is* some low-dimensional space common to all of our networks that we can embed into in the first place.

Let's go back to our group of human and alien brains and try using MASE to embed them rather than averaging. Then, we'll dive deeper into what's going on under the hood. First, we'll instantiate a MASE classifier and embed down to two dimensions. Then we'll create a combined list of the human and alien brains, and use MASE to find the estimated latent positions. You can see the results of using MASE on the humans and aliens in fig 4-10.

```python
from graspologic.embed import MultipleASE as MASE

# Use MASE to embed everything
mase = MASE(n_components=2)
latents_mase = mase.fit_transform(humans + aliens)
```

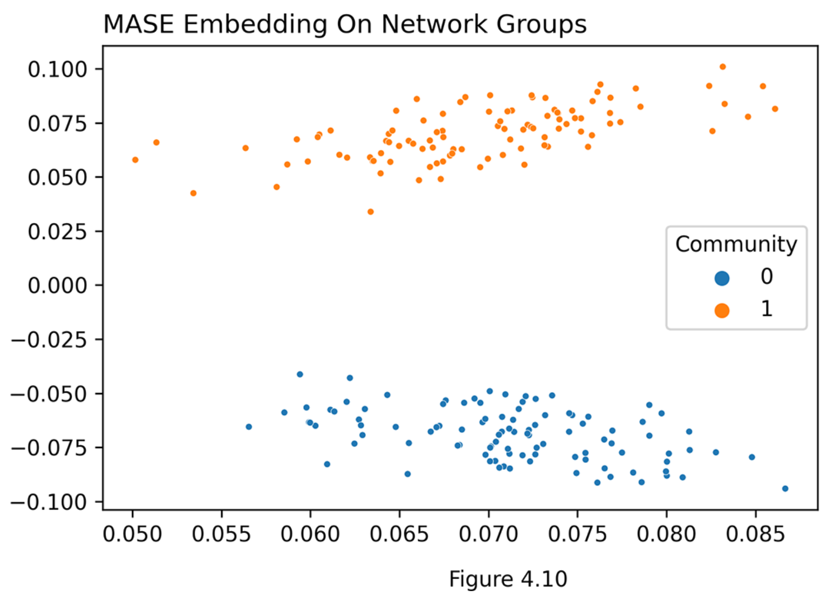


Figure 4-10. MASE Embedding on Network Groups

Unlike the disastrous results from simply averaging all of our networks together, MASE manages to keep the community structure that we found when we averaged our networks separately. Let's see what's under the hood.

Overview of MASE

Below, you can see how MASE works. We start with networks, drawn as nodes in space connected to each other. We turn them into adjacency matrices, and then we embed the adjacency matrices of a bunch of networks separately, using our standard Adjacency Spectral Embedding. Then, we take all of those embeddings, concatenate horizontally into a single matrix, and embed the entire concatenated matrix. The colors are the true communities each node belongs to: there's a red and an orange community. MASE is an unsupervised learning technique and so it doesn't need any information about the true communities to embed, but they're useful to see.

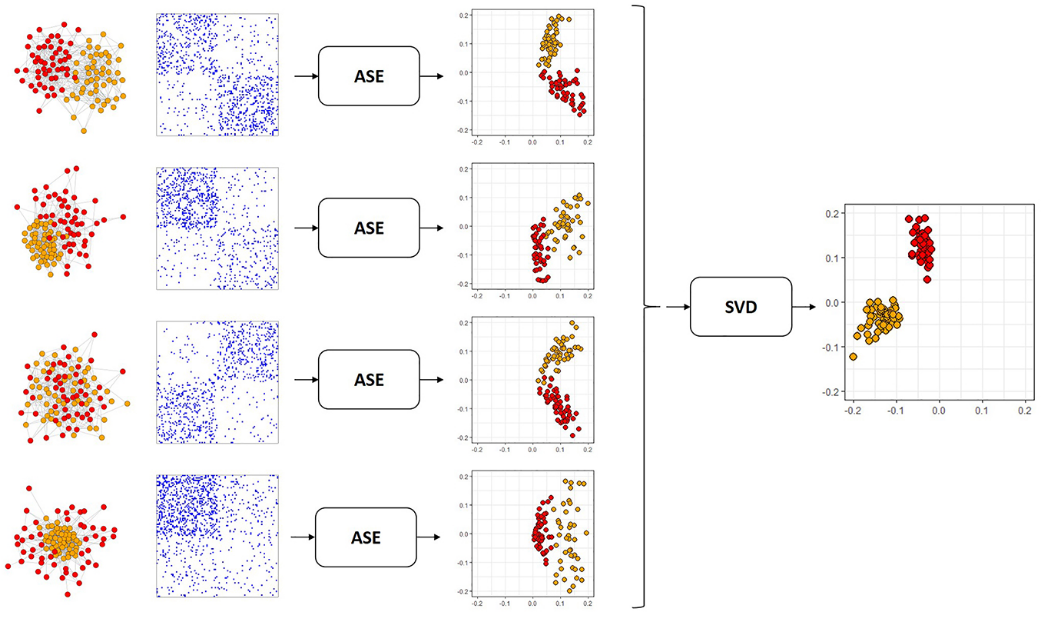


Figure 4-11. The MASE Algorithm

A Collection of Networks

We'll illustrate what's happening in the MASE algorithm by running through all of its steps ourselves, with a set of example networks.

Suppose we have a set of networks generated from Stochastic Block Models with two communities in each network. The networks have aligned nodes – meaning that the i_{th} row of all of their adjacency matrices represent the edges for the same node i. The nodes also all belong to the same communities. However, edge probabilities might change depending on the network. In the first network, you might have nodes in the same community having a high chance of connecting to each other, whereas in the second network, nodes are much more likely to be connected to other nodes in different communities. You want to end up with a classification that distinctly groups the nodes into their respective communities, using the information from all of the networks. Because MASE takes approach of combining the embeddings, we start by embedding each network separately with an Adjacency Spectral Embedding.

Below is Python code which generates four networks with Stochastic Block Models. Each of the networks is drawn from a different distribution (the block probability matrices are different), but the labels are the same across the networks (which means that nodes have a consistent community no matter which network you're looking at). If you're interested in the particular parameters used to generate these SBMs, you can see them in the code below.

```python
import numpy as np
from graspologic.simulations import sbm

n = 100
p1, p2, p3 = .12, .06, .03
A1, labels = make_sbm(p1, p3, p3, p1,
                      return_labels=True)
A2 = make_sbm(p1, p3, p3, p2)
A3 = make_sbm(p3, p2, p2, p3)
A4 = make_sbm(p1, p3, p3, p3)

networks = [A1, A2, A3, A4]
```

4.0.0.1 Embedding our Networks

Next, we embed each of the four networks separately using Adjacency Spectral Embedding. This step is pretty straightforward, so we won't dive into it too much: remember, we're combining the embeddings, not the networks, so we're not doing anything fancy. The python code below just groups the four networks into a list, and then loops through the list, embedding each network into two dimensions and saving the resulting embeddings into a variable.

```python
from graspologic.embed import AdjacencySpectralEmbed as ASE

networks = [A1, A2, A3, A4]
latents_mase = []
for network in networks:
    ase = ASE(n_components=2)
    latent = ase.fit_transform(network)
    latents_mase.append(latent)
```

It's important to keep in mind that these embeddings don't live in the same latent

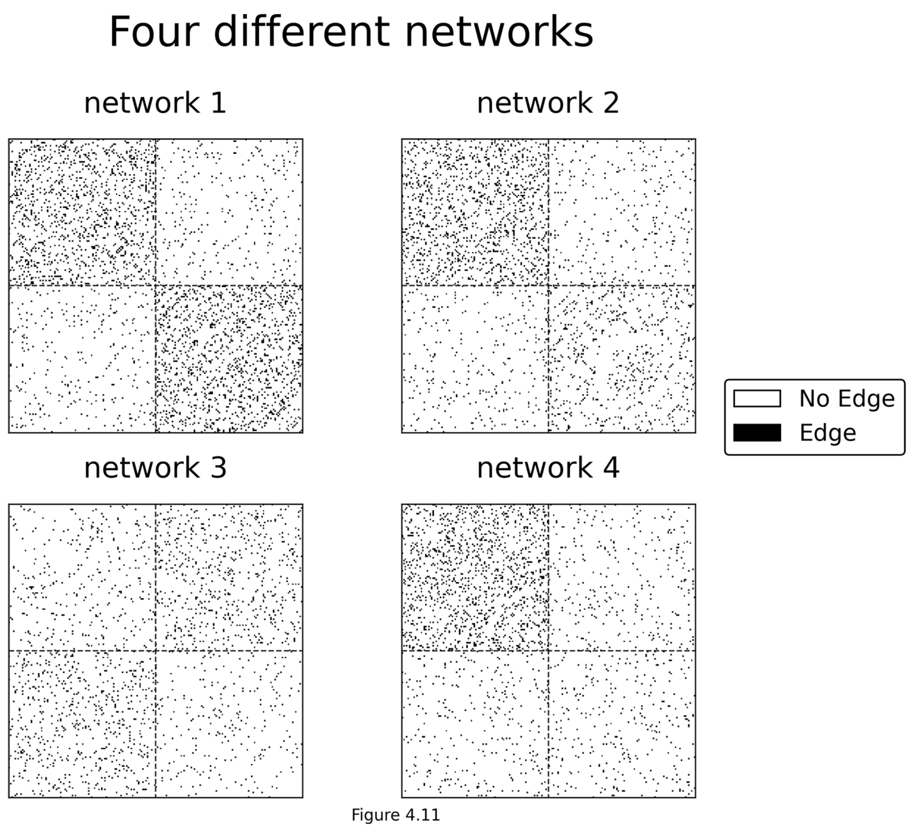


Figure 4-12. Four different networks

space. What this means is that averaging these networks together would result in essentially meaningless noise. This is because of the rotational invariance of latent position estimates: you can only estimate the latent positions of any network up to a rotation.

Combining our embeddings

Now comes the interesting part. Our goal is to find some way to take each of these individual embeddings and combine them. We want to find a reasonable way of doing this.

We can visualize each of our four embeddings a different way. Instead of the using the two latent position dimensions as the x-axis and the y-axis of our plot, we can just visualize our estimates latent position matrices directly. Each estimates latent

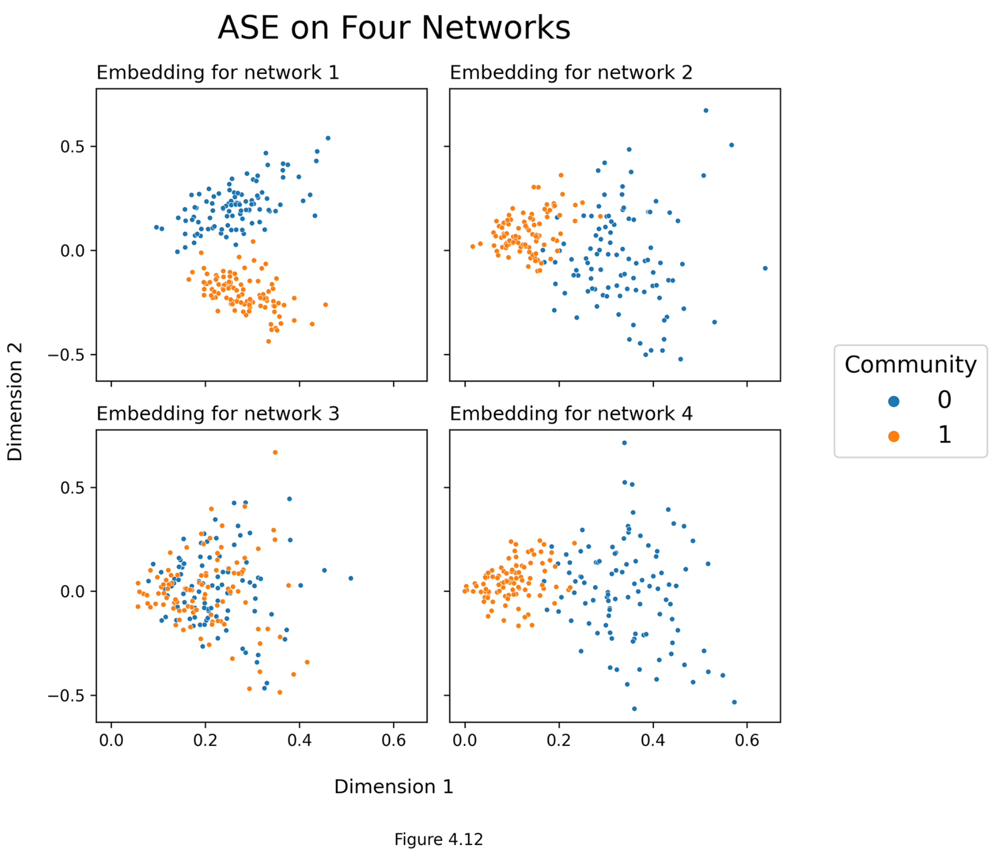


Figure 4-13. ASE on Four Networks

position now corresponds to rows in one of these matrices. The two columns are the two latent position dimensions, and the two colors in each row corresponds to the estimated latent position value. We're essentially substituting location for color.

Because the rows of these matrices are all aligned - meaning, row 0 corresponds to node 0 for all four matrices - we can actually think of each node as having (in this case) eight latent position dimensions: two for each of our four networks. Eight is a somewhat arbitrary number here: each network contributes two dimensions simply because we originally chose to embed all of our networks down to two dimensions with ASE, and the number of networks is of course even more arbitrary. You'll usually have more than four.

In the more general sense, we can think of each node as having $m \times d$ latent position dimensions, where m is the number of networks, and d is the number of

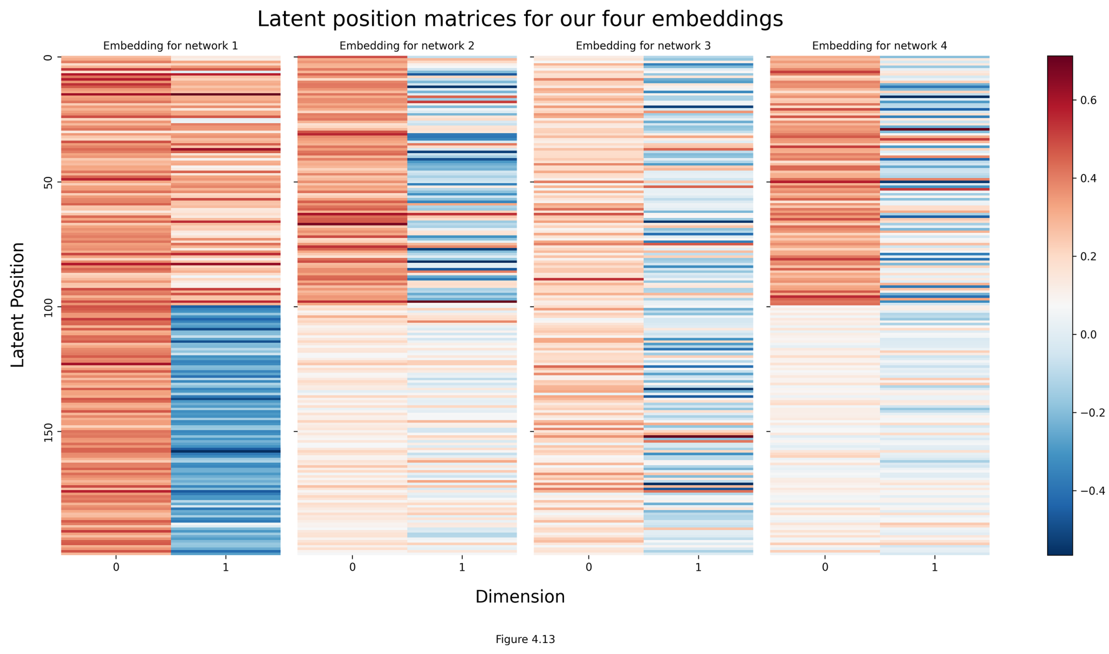


Figure 4-14. Estimated Latent Positions for our Four Embeddings

dimensions we embed each network into. We don't actually need separate matrices to express this idea: the natural thing to do would be to just concatenate all of the matrices horizontally into a single $m \times d$ matrix.

```python
# Concatenate our four matrices horizontally into a single m by d matrix
concatenated = np.hstack(latents_mase)
```

Embedding our Combination To Create a Joint Embedding

So now we have a combined representation for our separate embeddings, but we have a new problem: our estimated latent positions suddenly have way too many dimensions. In this example they have eight (the number of columns in our combined matrix), but remember that in general we'd have $m \times d$. This somewhat defeats the purpose of an embedding: we took a bunch of high-dimensional objects and turned them all into a single high-dimensional object. Big whoop. We can't see what our combined embedding look like in euclidean space, unless we can somehow visualize $m \times d$ dimensional space (hint: we can't). We'd like to just have ‘d‘ dimensions - that was the whole point of using ‘d‘ components for each of our Adjacency Spectral

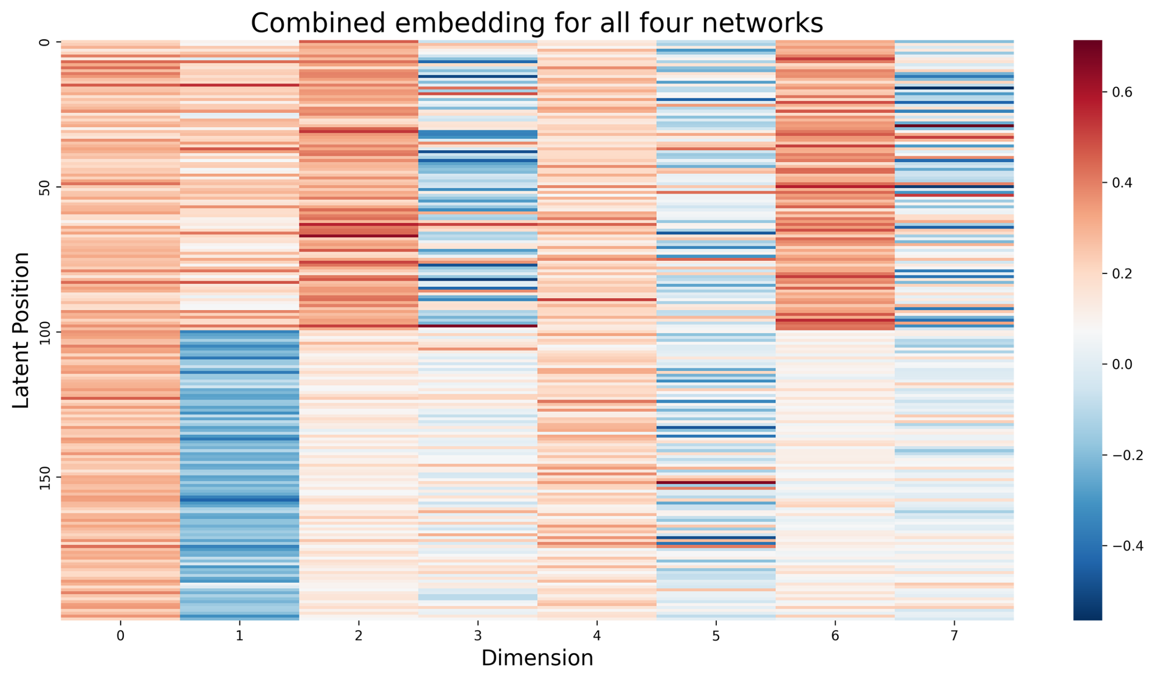


Figure 4.14

Figure 4-15. Combined embedding for all four networks

Embeddings in the first place!

There's an obvious solution here: why don't we just embed *again*? Nothing stops us from doing a Singular Value Decomposition on a nonsquare matrix, and so we can just create a joint embedding of our combined matrix and go back down to a healthy d columns.

```
from graspologic.embed import select_svd
joint_embedding, *_ = select_svd(concatenated, n_components=2)
```

Looks like this idea worked well - Our nodes are clearly grouped into two distinct communities, and all of our networks were drawn from the same distribution! To reiterate, what we did was:

1. Embed each of our four networks separately into two-dimensional space
2. Think of all of the resulting estimated latent positions for a particular node as a single vector

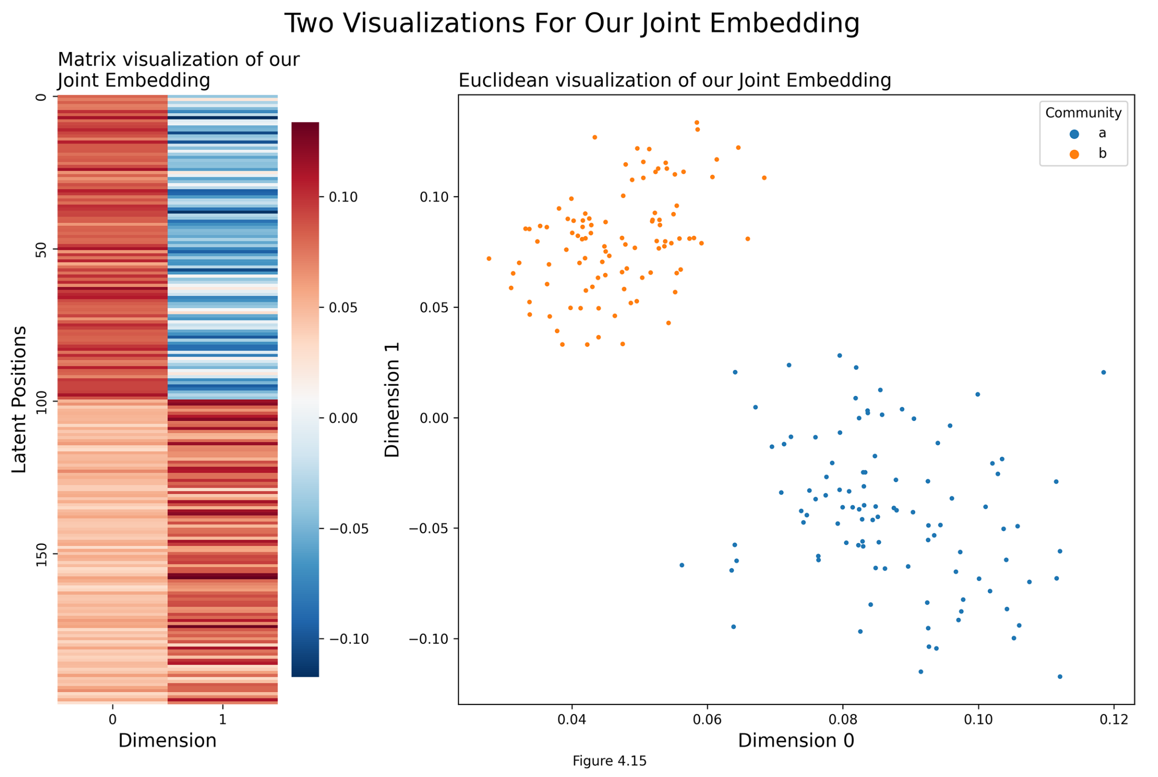


Figure 4-16. Two Visualizations for our Joint Embedding

3. With the intuition from 2, horizontally concatenate our four estimated latent position matrices into a single matrix

4. embed that new matrix down to 2 dimensions

Using Graspologic

In practice, you don't actually have to implement any of this stuff yourself. Graspologic's MultipleASE class implements it all for you under the hood. You can see the embedding below - you give MultipleASE a list of networks, and it spits out a set of joint latent position estimates. Graspologic's implementation of MASE is doing pretty much exactly what we just did: it embeds all of the networks you pass in, concatenates them horizontally, and then re-embeds the concatenated matrix. You can see this in the figure – MASE's embedding looks just like the one we made above.

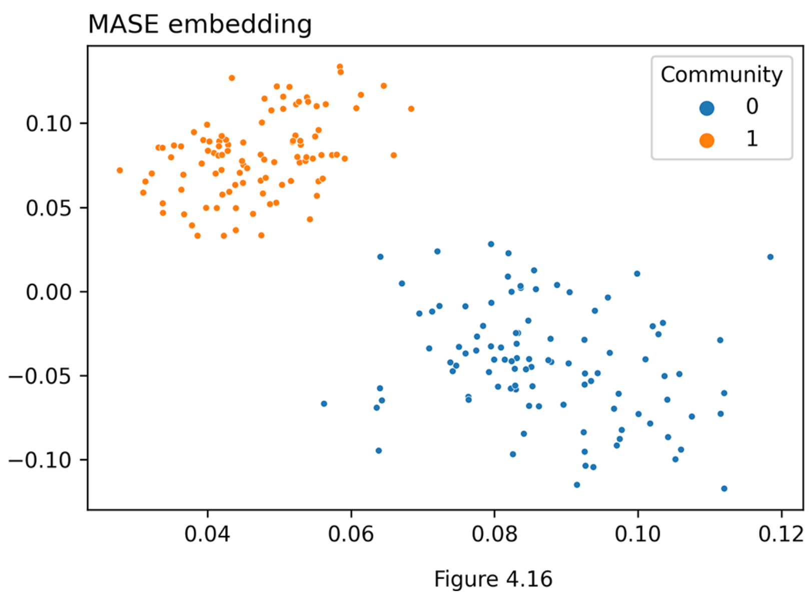


Figure 4-17. MASE Embedding

Score Matrices

Exactly how is the joint embedding we created related to all of separate, original networks? Well, to understand this, we need to introduce the concept of *score matrices*.

In MASE, each network is associated with its own score matrix. Just like the joint embedding describes how the networks are similar, the score matrices describe how each network is different.

Suppose we have a set of networks with adjacency matrices $A^{(1)}, ..., A^{(m)}$, with each network being unweighted. In the joint embedding we made before, for instance, we had $m = 4$.

Now, we run MASE using the method described above, and we get a joint embedding V. Then each adjacency matrix, $A^{(i)}$, can be decomposed into $VR^{(i)}V^\top$, where $R^{(i)}$ is the score matrix corresponding to the i_{th} network:

$$A^{(i)} = VR^{(i)}V^\top$$

This is how the score matrix of a particular network $R^{(i)}$ and the single joint embedding V is related to the original network $A^{(i)}$.

4.0.0.2 Finding Score Matrices

Any particular score matrix, $R^{(i)}$, is square and $d \times d$. The dimension, d, corresponds to the number of embedding dimensions – so if we wanted to embed down to two dimensions, each $R^{(i)}$ would be a 2×2 matrix.

Now, here's the interesting part: how do we find our score matrices? Well, there's a theorem in linear algebra about matrices which are *orthogonal*, meaning that the columns all perpendicular to each other. This theorem says that the inverse of an orthogonal matrix is its transpose. So, for an orthogonal matrix O,

$$O^\top = O^{-1}$$

Interestingly, the column-vectors of our joint embedding matrix (let's call it V) are all perpendicular. Since definitionally, what it means for two vectors to be perpendicular is that they have a dot product of 0, we can check this below:

```
V = joint_embedding.copy()

# Take the dot product of the columns of our estimated joint latent position
    matrix
np.round(V[:, 0] @ V[:, 1])

>> 0.0
```

What this all means is that $V^\top V$ is just the identity matrix I.

And so, finally, we can use the above two facts to find the score matrix for a particular network. We just take our original formula $A^{(i)} = VR^{(i)}V^\top$, left-multiply by $V^\top$, and right-multiply by V.

$$\begin{aligned}A^{(i)} &= VR^{(i)}V^\top \\ V^\top A^{(i)}V &= (V^\top V)R^{(i)}(V^\top V) \\ V^\top A^{(i)}V &= R^{(i)}\end{aligned}$$

Below, we turn the list of four networks we already embedded into a 3-D numpy array, and then do the above multiplication to get a new 3D numpy array of scores matrices. Because we embedded into two dimensions, each score matrix is 2×2, and the four score matrices are "slices" along the 0th axis of the numpy array.

```
networks_array = np.asarray(networks)
scores = V.T @ networks_array @ V
```

The scores have shape (4, 2, 2). Now, here's something interesting: it turns out that we can estimate the edge probability matrix which generated any graph with $P^{(i)} = VR^{(i)}V^\top$.

```
P_0 = V @ scores[0] @ V.T
```

Below and to the left, you can see the original adjacency matrix for the first matrix. In the center, you can see the heatmap for the first network's score matrix. Next to it, you can see the recreation of the first network. Remember that we only used the score matrix to recreate it. The first network has a block probability matrix of

$$\begin{bmatrix} .12 & .03 \\ .03 & .06 \end{bmatrix} \tag{4.1}$$

and so we should expect the edges in the top-left block of our adjacency matrix to be more connected, the edges in the two off-diagonal blocks to not be very connected, and the edges in the bottom-right block to be kind of connected.

So we've learned that MASE is useful when you want a joint embedding that combines all of your networks together, and when you want to estimate edge probabilities

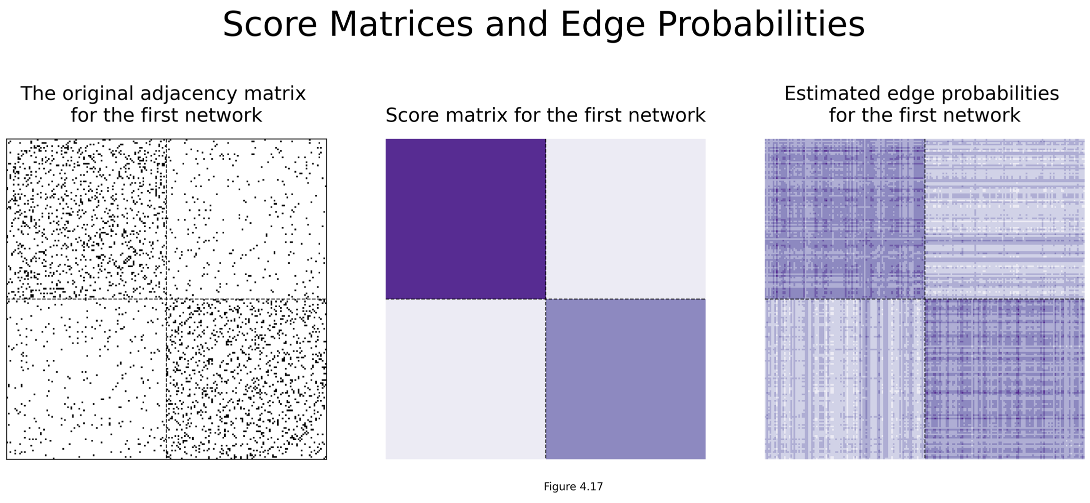


Figure 4-18. Score Matrices and Edge Probabilities

for one of your networks. What if we wanted to keep our separate embeddings, but put them all in the same space? That's what the Omnibus Embedding gives, and what we'll explore now.

Omnibus Embedding

The Omnibus Embedding combines networks separately to put them all into the same latent space. What this means is that the embeddings for each network after the omnibus embedding are *directly comparable*: none of the embeddings are rotations of each other, and distances between nodes across embeddings actually means something. You can use the omnibus embedding to answer a variety of questions about the interacting properties of a collection of networks. For example, you could figure out which nodes or subgraphs are responsible for similarities or differences across your networks, or you could determine whether subcommunities in your networks are statistically similar or different. You could try to figure out which underlying parameters of your network are the same, and which are different.

In the next section, we'll explore how the Omnibus Embedding works. Sections in future chapters will explore some the things you can do with your separate embeddings to learn about your networks.

OMNI on our four networks

We'll begin with an example. Let's go back to the four networks we created in the MASE section and look at their embeddings. Notice that the way the blue cluster of points and the red cluster of points is rotated is somewhat arbitrary across the embeddings for our different networks - this is because of the nonidentifiability problem in spectral embeddings.

The nonidentifiability problem is as follows: Let's take a network observed from an RDPG with n nodes. Each of these n nodes is associated with an estimated latent position vector, corresponding to that node's row in the network's embedding. What it means for a node to have a latent position vector is that the probability for an edge to exist between two nodes i and j is the dot product of their latent position vectors.

More specifically, if $\mathbf{P}$ is a matrix of edge probabilities, and $\mathbf{X}$ is our latent position matrix, then $\mathbf{P} = \mathbf{X}\mathbf{X}^\top$.

The nonidentifiability problem is as follows: take any orthogonal matrix (a matrix which only rotates or flips other matrices). Call it $\mathbf{W}$. By definition, the transpose of any orthogonal matrix is its inverse: $\mathbf{W}\mathbf{W}^\top = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. So,

$$\begin{aligned} P &= \mathbf{X}\mathbf{X}^\top && (4.2) \\ &= \mathbf{X}\mathbf{I}\mathbf{X}^\top && (4.3) \\ &= (\mathbf{X}\mathbf{W})(\mathbf{W}^\top\mathbf{X}^\top) && (4.4) \\ &= (\mathbf{X}\mathbf{W})(\mathbf{X}\mathbf{W})^\top && (4.5) \\ & && (4.6) \end{aligned}$$

What this means is that you can take any latent position matrix and rotate it, and the rotated version will still generate the same matrix of edge probabilities. So, when you try to estimate the latent positions, separate estimations can produce rotated versions of each other.

You need to be aware of this in situations where you're trying to directly compare more than one embedding. You wouldn't be able to figure out the average position of a node, for instance, when you have multiple embeddings of that node.

You can see the nonidentifiability problem in action below. The embeddings for network 1 and for network 2 are particularly illustrative; community 0 is generally top in network 1, but on the right in network two. There isn't a way to compare any two nodes directly. Another way to say this is that, right now, all of our embeddings live in different *latent spaces*: direct comparison between embeddings for nodes in network 1 and nodes in network 2 isn't possible. You can also see the estimated latent position corresponding to the first node as a big red circle in each network so that you can track a single point - you can see that the red points are likely to be rotated or flipped across networks.

OMNI on our four heterogeneous networks

Let's see what happens when, instead of embedding our networks separately as above, we estimate their latent positions with an Omnibus Embedding. Again, we'll plot a particular node with a circle so that we can track it across embeddings.

Unlike when we embedded the four networks separately, the clusters created by the Omnibus Embedding *live in the same space*: you don't have to rotate or flip your points to line them up across embeddings. The cluster of blue points is always in the top left, and the cluster of red points is always in the bottom right. This means that we can compare points directly; the relative location of the node in your network corresponding to the red circle, for instance, now means something across the four networks, and we can do stuff like measure the distance of the red circle in network 1 to the red circle in network two to gain information.

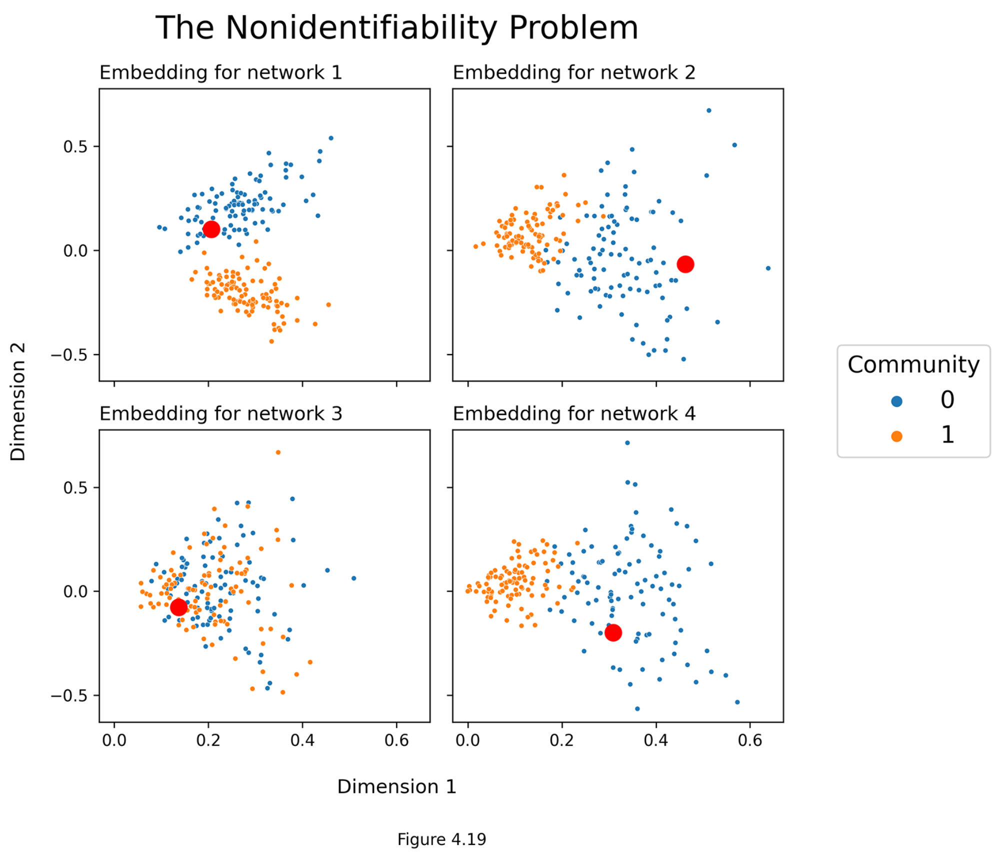


Figure 4-19. The Nonidentifiability Problem

Overview of OMNI

At a high level, the omnibus embedding is fairly simple. It: 1. Combines the adjacency matrices for all of our networks into a single, giant matrix (the Omnibus Matrix) 2. Embeds that matrix using a standard Adjacency or Laplacian Spectral Embedding.

The omnibus matrix itself just has every original adjacency or laplacian matrix along its diagonal, and the elementwise average of every pair of original matrices on the off-diagonals. This means that the Omnibus Matrix is *huge*: if you have m networks, each of which has n nodes, the Omnibus Matrix will be a $mn \times mn$ matrix.

For example, say we only have two networks. Let's name their adjacency matrices $A^{(1)}$ and $A^{(2)}$. Then, the omnibus embedding looks like this:

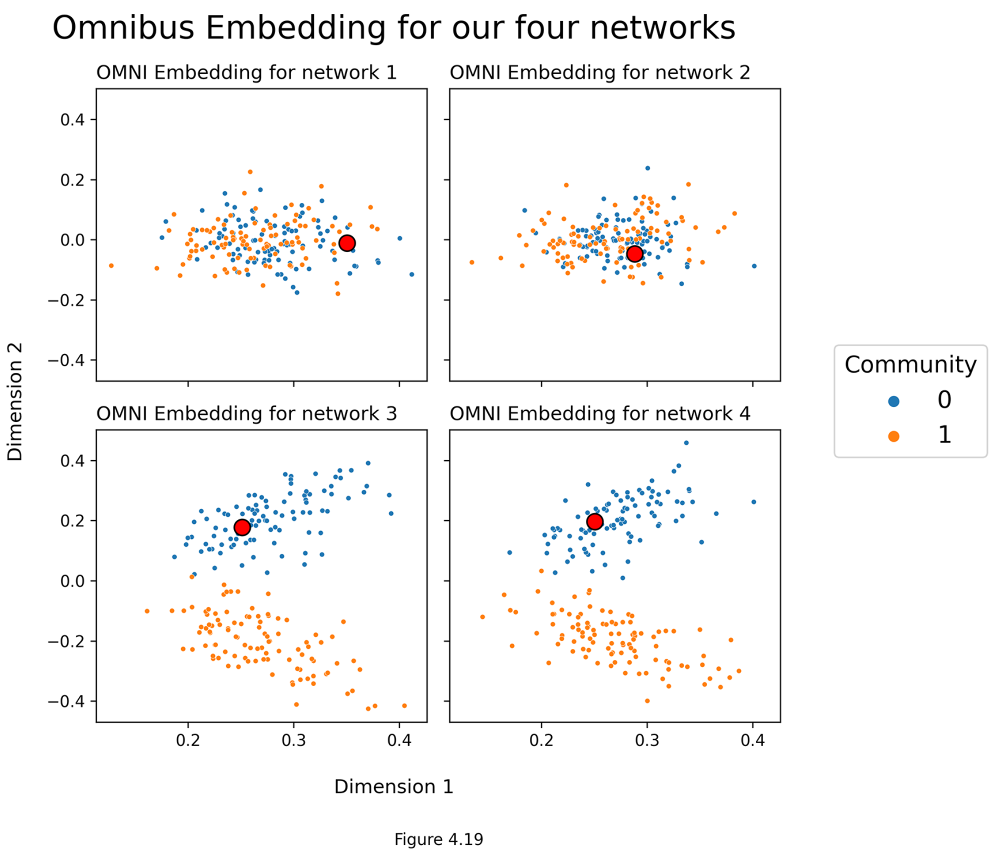


Figure 4-20. Omnibus Embedding for our Four Networks

$$\begin{bmatrix} A^{(1)} & \frac{A^{(1)}+A^{(2)}}{2} \\ \frac{A^{(2)}+A^{(1)}}{2} & A^{(2)} \end{bmatrix} \tag{4.7}$$

where each entry on the diagonal is itself a matrix. In general, when we have m networks, the i_{th} diagonal entry is $A^{(i)}$ and the $(i, j)_{th}$ entry is $\frac{A^{(i)}+A^{(j)}}{2}$. What this means is that you just stick each of your adjacency matrices on the diagonal of a large matrix, and you fill in the off-diagonals with the averages of each pair of two adjacency matrices.

You can see this in code below. Below, we just use numpy's block function to generate our simple Omnibus Matrix from two networks.

```
a0, a1 = networks[0], networks[1]
omni = np.block([[a0, (a0+a1)/2],
                 [(a1+a0)/2, a1]])
```

Below you can see the resulting Omnibus Matrix. The first and second networks are shown as heatmaps on the left, and their Omnibus Matrix is shown on the right.

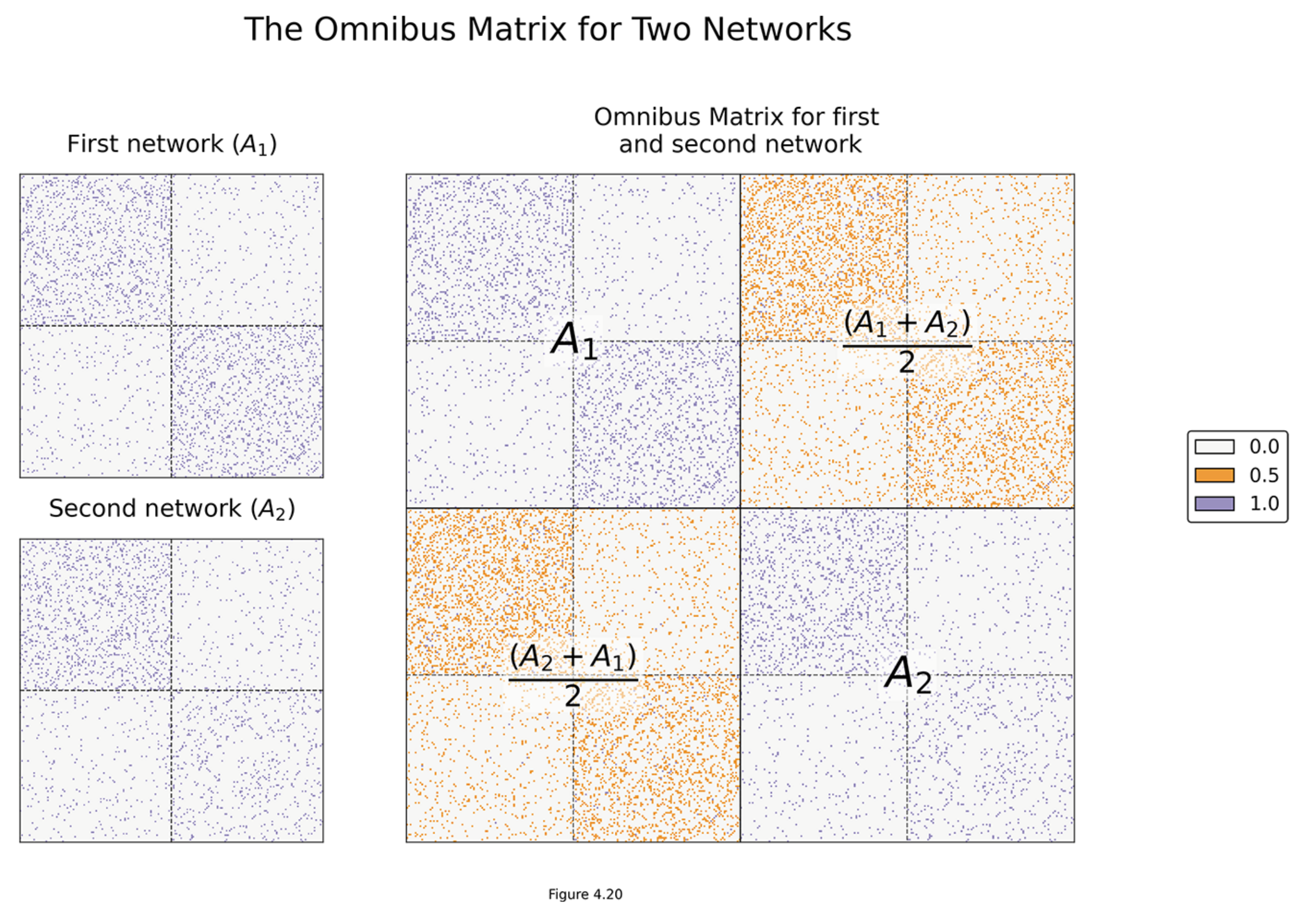


Figure 4-21. The Omnibus Matrix for Two Networks

Creating the Omnibus Matrix For All Four Networks

Here's the Omnibus Matrix for all four of our networks. You can see adjacency matrices for the original four networks on the diagonal blocks, highlighted in blue, and all possible pairs of averages of adjacency matrices on the off-diagonal blocks, highlighted in orange.

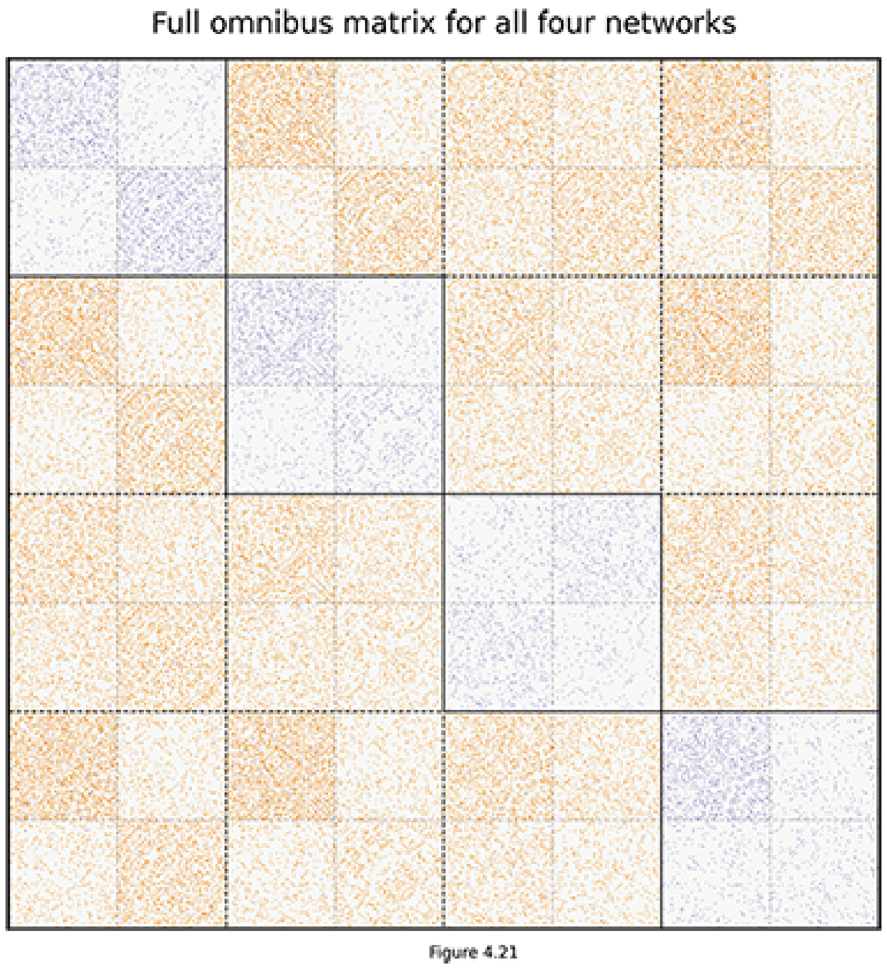


Figure 4-22. Full Omnibus Matrix for all four networks

Embedding the Omnibus Matrix

You should understand the next step fairly well by now. We embed the Omnibus Matrix normally, using ASE, as if it were just a normal adjacency matrix. This will estimate an $nm \times d$ sized latent position matrix (where, remember, n is the number of nodes in each network, m is the number of networks, and d is the number of embedding dimensions). Here, since each of our four networks has 200 nodes, mn is 800, and we chose to embed down to two dimensions.

```
from graspologic.embed import select_svd

U, D, V = select_svd(omni, n_components=2)
joint_embedding = U @ np.diag(np.sqrt(D))
```

The resulting matrix is 800×2.

Estimating Separate Latent Positions In The Same Latent Space

Now, the only question we have remaining is how to actually pull the separate estimated latent positions for each network from this matrix. It turns out that the individual latent position estimates for each network are actually stacked on top of each other: the first n rows of the joint matrix we just made correspond to the nodes of the first network, the second n rows correspond to the nodes of the second network, and so on.

If we want, we can pull out the separate latent position estimates for each network explicitly. Below, we reshape our 2-dimensional $mn \times d$ numpy array for the omnbus embedding into a $m \times n \times d$ array: the embeddings for each network are now simply stacked on top of each other on the third dimension (and the first axis of our numpy array).

```
m = len(networks)
n = len(networks[0])
latent_networks = joint_embedding.reshape(m, n, -1)

latent_networks.shape
```

Below, you can see the embeddings we just created. On the left is the full $mn \times d$ omnibus matrix, and on the right are the slices of the $m \times n \times d$ 3-D array we created above. If you look carefully, you can see that the top two blocks of colors (row-wise) in the larger embedding correspond to the estimated latent positions for network 1, the second two blocks correspond to the estimated latent positions for network 2, and so on. They're a bit squished, so that everything lines up nicely, but they're there.

And finally, below is the above embeddings, plotted in Euclidean space. Each point is a row of the embedding above, and the dots are colored according to their class label. The big matrix on the left (the joint OMNI embedding) just contains every estimated latent position we have, across all of our networks. This means that, on the

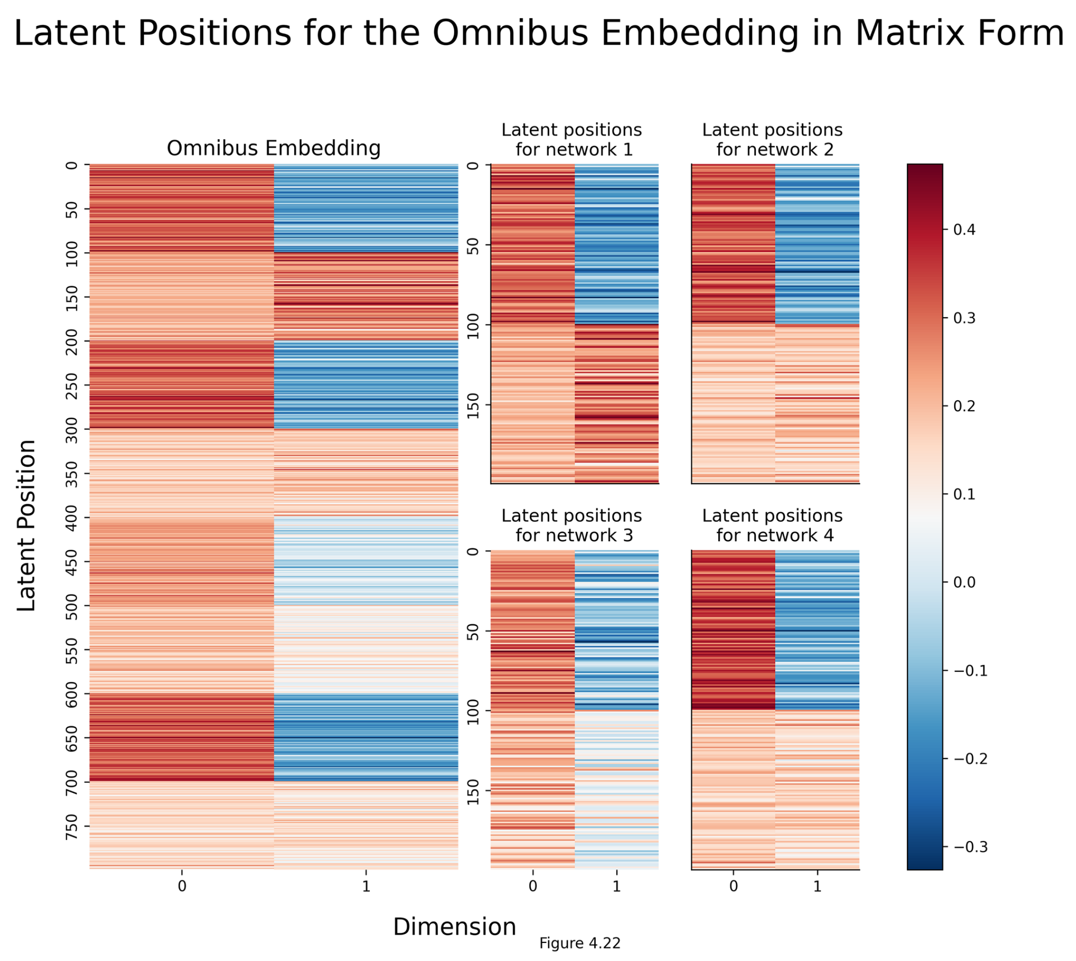


Figure 4-23. Latent Position Estimates for the Omnibus Embedding in Matrix Form

lefthand plot, there will be four points for every node (remember that we're operating under the assumption that we have the same set of nodes across all of our networks).

How Can You Use The Omnibus Embedding?

Fundamentally, the omnibus embedding is useful is because it lets you avoid the somewhat annoying and noise-generating process of figuring out a good way to rotate your separate embeddings to line them up. For instance, say you want to figure out if two networks are generated from the same distribution (This means that the matrix that contains edge probabilities, $\mathbf{P}$, is the same for both networks). Then, it's reasonable to assume that, since their true latent positions are the same, their

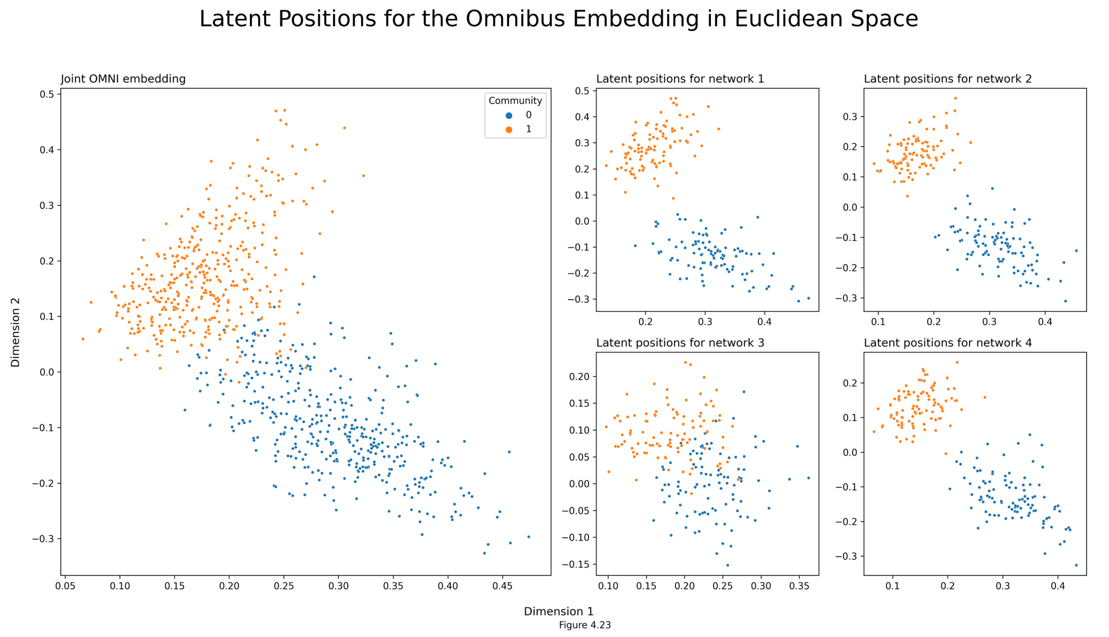


Figure 4-24. Latent Position Estimates for the Omnibus Embedding in Euclidean Space

estimated latent positions will be pretty close to each other. Look at the equation below:

$\min_W ||\hat{N}_1 - \hat{N}_2 W||_F$

Here: - W is a matrix that just rotates or flips (called an isometry, or an orthonormal matrix) - $\hat{N}_1$ and $\hat{N}_2$ are the estimated latent positions for networks one and two, respectively

the $||X||_F$ syntax means that we're taking the frobenius norm of X. Taking the frobenius norm of a matrix is the same as unwrapping the matrix into a giant vector and measuring that vector's length. So, this equation is saying that the latent position estimates for a given node in network one should be close to the latent position estimates in network two.

But, there's that W there, the rotation matrix. We actually wish we didn't have to find it. We have to because of the same problem we keep running into: you can rotate latent positions and they'll still have the same dot product relative to each other, and so you can only embed a network up to a rotation. In practice, you can find this matrix and rotate latent positions for separate networks using it to compare them directly, but again, it's annoying, adds compute power that you probably don't want to use, and it'll add noise to any kind of inference you want to do later.

The Omnibus Embedding is fundamentally a solution to this problem. Because the embeddings for all of your networks live in the same space, you don't have to rotate them manually – and you cut out the noise that gets created when you have to *infer* a good rotation matrix. We'll explore all the downstream use cases in future chapters, but below is a sneak peak.

The figure below (adapted from Gopalakrishnan et al. 2021), is the omnibus embedding for 32 networks created from a bunch of mouse brains, some of which have been genetically modified. The nodes of these networks represent the regions of a mouse brain and the edges represent how well-connected the neurons in a given pair of regions are. The figure below actually only shows two nodes: the node representing one region in the left hemisphere, and the node representing its corresponding region in the right hemisphere.

So what we're actually *plotting* in this embedding is a bit different than normal, because rather than being nodes, the points we plot are *networks*: one for each of our thirty-two mice. The only reason we're able to get away with doing this is the omnibus embedding: each network lives in the same space!

You can clearly see a difference between the genetically modified mice and the normal mice. The genetically modified mice are off in their own cluster; if you're familiar with classical statistics, you could do a MANOVA here and find that the genetically modified mice are significantly different from the rest - if we wanted to, we could figure out which mice are genetically modified, even without having that information in advance!

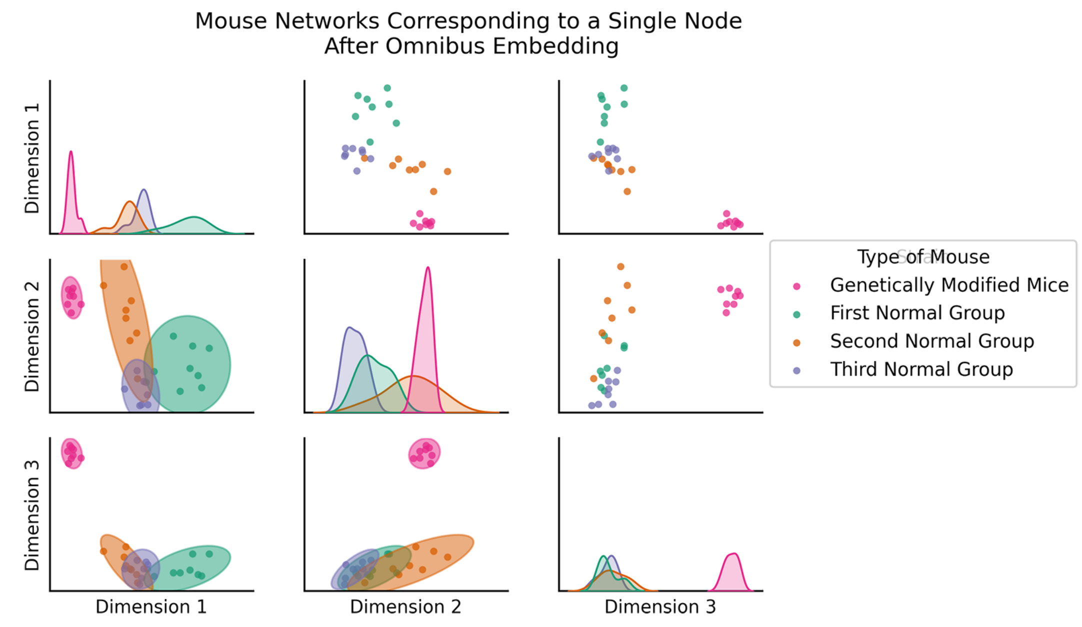


Figure 4-25. Mouse Networks Corresponding to a Single Node after Omnibus Embedding

Chapter 5

Joint Representation Learning

In many problems, our network might be more than just the information contained in its adjacency matrix (which, in network space, is called its topology, or the collection of nodes and edges). If we were investigating a social network, we might have access to extra information about each person – their gender, for instance, or their age. If we were investigating a brain network, we might have information about the physical location of neurons, or the volume of a brain region. When we we embed a network, it seems like we should be able to use these extra bits of information - called the "features" or "covariates" of a network - to somehow improve our analysis. The techniques and tools that we'll explore in this section use both the covariates and the topology of a network to create and learn from new representations of the network. Because they jointly use both the topology of the network and its extra covariate information, these techniques and tools are called joint representation learning.

There are two primary reasons that we might want to explore using node covariates in addition to topological structure. First, they might improve our standard embedding algorithms, like Laplacian and Adjacency Spectral Embedding. For example, if the latent structure of the covariates of a network lines up with the latent structure of its topology, then we might be able to reduce noise when we embed, even if the communities in our network don't overlap perfectly with the communities in our covariates. Second, figuring out what the clusters of an embedding actually mean can sometimes be difficult and covariates create a natural structure in our network that we can explore. Covariate information in brain networks telling us where in the brain each node is, for instance, might let us better understand the types of characteristics that distinguish between different brain regions.

In this section, we'll explore different ways to learn from our data when we have access to the covariates of a network in addition to its topological structure. We'll explore *Covariate-Assisted Spectral Embedding* (CASE), a variation on Spectral Embedding. In CASE, instead of embedding just the adjacency matrix or its regularized Laplacian, we'll combine the Laplacian and our covariates into a new matrix and embed that.

A good way to illustrate how using covariates might help us is to use a model in which some of our community information is in the covariates and some is in our topology. Using the Stochastic Block Model, we'll create a simulation using three communities: the first and second community will be indistinguishable in the topological structure of a network, and the second and third community will be indistinguishable in its covariates. By combining the topology and the covariates, we'll get a nice embedding that lets us find three distinct community clusters.

Topology

Suppose we have a network observed from an SBM with three communities. In our adjacency matrix, which contains only our network's topological information, we'd like to create a situation where the first two communities are completely indistinguishable: Any random node in the first community will have exactly the same chance of being connected to another node in the first community or to a node in the second community. We'd like the third community to be distinct, with only a small probability that nodes in it will connect to nodes in either of the other two communities.

The Python code below generates a matrix that looks like this. There are 1500 nodes, with 500 nodes per community. Because the 3×3 block probability matrix that generated this SBM has the same probability values (.3) in its upper-left 2×2 square, a node in community 1 has a 30% chance of being connected to either another node in community 1 or a node in community 2. As a result, in our adjacency matrix, we'll see the nodes in communities one and two as a single giant block. On the other hand, nodes in community three only have a 15% chance to connect to nodes in the first community. So, the end result is that we've created a situation where we have three communities that we'd like to separate into distinct clusters, but the topological structure in the adjacency matrix can't separate the three groups by itself.

```
import numpy as np
from graspologic.simulations import sbm

# Start with some simple parameters
N = 1500 # Total number of nodes
n = N // 3 # Nodes per community
p, q = .3, .15
B = np.array([[.3, .3, .15],
             [.3, .3, .15],
             [.15, .15, .3]]) # Our block probability matrix

# Make our Stochastic Block Model
A, labels = sbm([n, n, n], B, return_labels = True)
```

Here you can see what our adjacency matrix looks like. Notice the giant block in the top-left: this block contains both nodes in both of the first two communities, and they're indistinguishable from each other.

If we wanted to embed this graph using our Laplacian or Adjacency Spectral Embedding methods, we'd find the first and second communities layered on top of each other (though we wouldn't be able to figure that out from our embedding if we didn't cheat by knowing in advance which community each node is supposed to belong to). The python code below embeds the Laplacian of our network down to two dimensions. Below it, you can see a plot of the estimated latent positions, with each node color-coded by its true community.

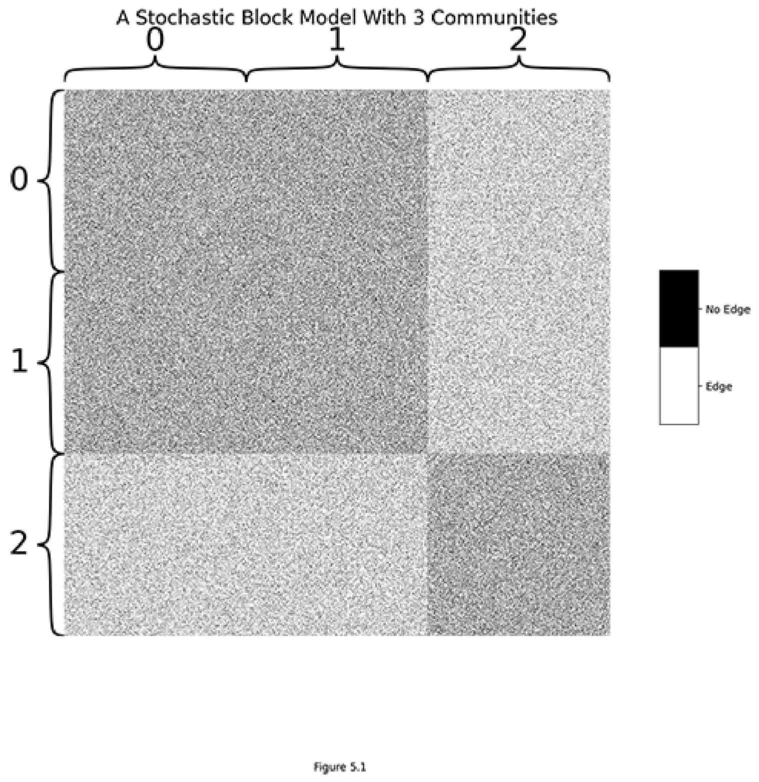


Figure 5-1. An SBM With the first two communities indistinguishable

```
from graspologic.embed import LaplacianSpectralEmbed as LSE
from graspologic.utils import to_laplacian

L = to_laplacian(A, form="R-DAD")
L_latents = LSE(n_components=2).fit_transform(L)
```

As you can see, we'd have a tough time clustering this - the first and second community are completely indistinguishable. It would be nice if we could use extra information to more clearly distinguish between them. We don't have this information in our adjacency matrix: it needs to come from somewhere else.

Covariates

But we're in luck - we have a set of covariates for each node! These covariates contain the extra information we need that allows us to separate our first and second community. However, with only these extra covariate features, we can no longer

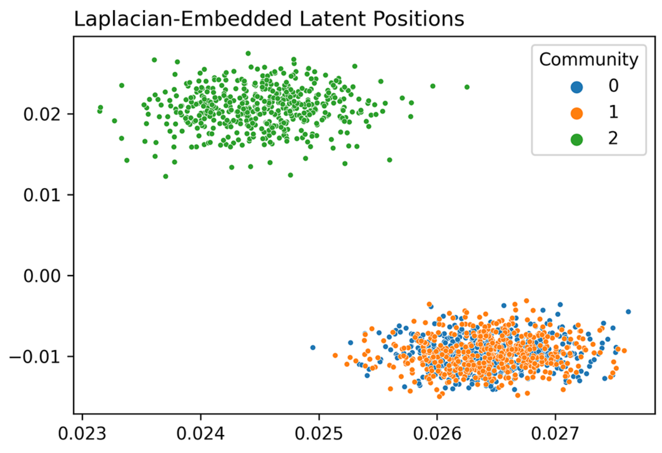


Figure 5.2

Figure 5-2. Laplacian-Embedded Estimated Latent Positions

distinguish between the last two communities - they contain the same information.

Below is Python code that generates these covariates. Each node is associated with its own group of 30 covariates (thirty being chosen primarily to visualize what's happening more clearly). We'll organize this information into a matrix, where the i_{th} row contains the covariates associated with node i. Remember that we have 1500 nodes in our network, so there will be 1500 rows. We'll draw all the covariates for each node from a Beta distribution (with values ranging from 0 to 1). If you're not familiar with Beta distributions, don't worry about it - in our case, we're essentially just using it to pick values between 0 and 1. The nodes in the first community will be drawn from a different Beta distribution than the nodes in the last two.

```python
from scipy.stats import beta

def make_community(a, b, n=500):
    return beta.rvs(a, b, size=(n, 30))

def gen_covariates():
    c1 = make_community(2, 5)
    c2 = make_community(2, 2)
    c3 = make_community(2, 2)

    covariates = np.vstack((c1, c2, c3))
    return covariates
```

```
# Generate a covariate matrix
Y = gen_covariates()
```

Fig 5-3 contains a visualization of the covariates we just created. The first community is represented by the lighter-colored rows, and the last two are represented by the darker-colored rows.

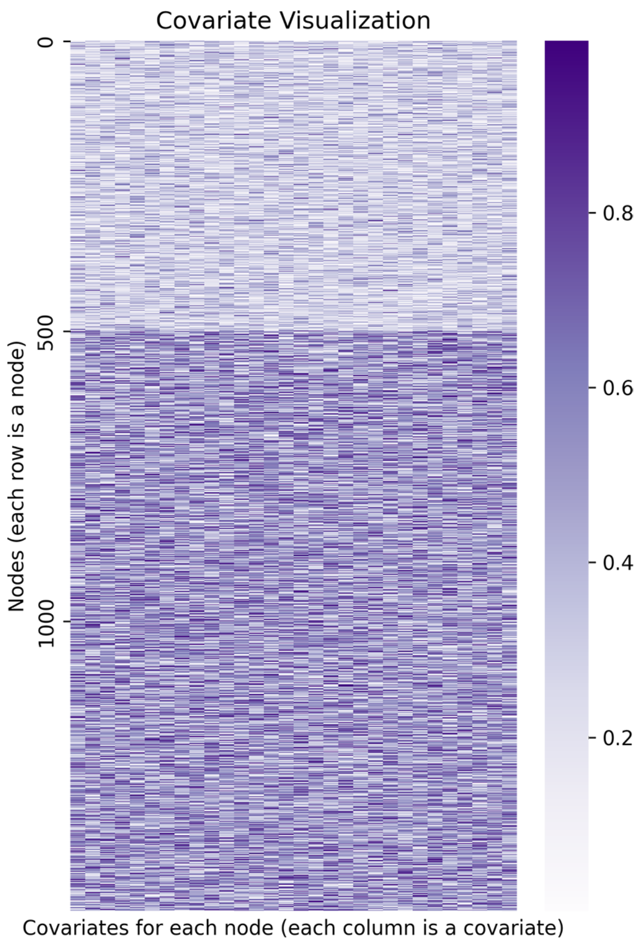


Figure 5.3

Figure 5-3. Covariate Visualization

We can play almost the same game here as we did with the Laplacian. If we embed the information contained in this matrix of covariates into lower dimensions, we can see the reverse situation as before - the first community is separate, but the last two are overlayed on top of each other.

Below is Python code that embeds the covariates. We'll use custom embedding code rather than graspologic's LSE class, because it isn't technically right to act as if we're embedding a Laplacian (even though we're doing essentially the same thing under the hood). Underneath it is a plot of the resulting embedding.

```
from sklearn.utils.extmath import randomized_svd

def embed(matrix, *, dimension):
    latents, _, _ = randomized_svd(matrix, n_components=dimension)
    return latents

Y_latents = embed(Y, dimension=2)
```

As you can see, we're in a similar situation as we were in with the adjacency matrix, but with different communities: instead of the first and second communities being indistinguishable, now the second and third are. We'd like to see full separation between all three communities, so we need some kind of representation of our network that allows us to use both the information in the topology and the information in the covariates. This is where CASE comes in.

Covariate-Assisted Spectral Embedding

Covariate-Assisted Spectral Embedding, or CASE, is a simple way of combining our network and our covariates into a single model. In the most straightforward version of CASE, we combine the network's regularized Laplacian matrix L and a function of our covariate matrix $YY^\top$. Here, Y is just our covariate matrix, in which row i contains the covariates associated with node i. Notice the word "regularized" - This means (from the Laplacian section earlier) that our Laplacian looks like $L = L_\tau = D_\tau^{-1/2} A D_\tau^{-1/2}$ (remember, D is a diagonal matrix with D_{ii} telling us the degree of node i).

A note on understanding what $YY^\top$ is all about: Suppose that Y only contains 0's and 1's. To interpret YY^T, notice we're effectively taking the the dot product of each row of Y with each other row, because the transpose operation turns rows into columns. Now, look at what happens below when we take the dot product of two example vectors with only 0's and 1's in them:

$$\begin{bmatrix}1 & 1 & 1\end{bmatrix} \cdot \begin{bmatrix}0\\1\\1\end{bmatrix} = 1 \times 0 + 1 \times 1 + 1 \times 1 = 2 \tag{5.1}$$

If there are two overlapping 1's in the same position of the left vector and the right vector, then there will be an additional 1 added to their weighted sum. So, in the case of a binary YY^T, when we matrix-multiply a row of Y by a column of Y^T, the resulting value, $(YY^T)_{i,j}$, will be equal to the number of shared locations in which vectors i and j both have ones.

So a particular value in $YY^\top$, $(YY^\top)_{i,j}$, can be interpreted as measuring the "agreement" or "similarity" between row i and row j of our covariate matrix. Since each row represents the covariates for a particular node, the higher the value of $(YY^\top)_{i,j}$, the more similar the covariates of the i_{th} and j_{th} nodes are. The overall result is a matrix that looks fairly similar to our Laplacian!

The following Python code generates our covariate similarity matrix $YY^\top$. We'll also normalize the rows of our covariate matrix to have unit length using scikit-learn - this is because we want the scale for our covariate matrix to be roughly the same as the scale for our adjacency matrix. Later, we'll weight $YY^\top$ to help with this as well.

```python
from sklearn.preprocessing import normalize

Y = normalize(Y, axis=0)
YYt = Y@Y.T
```

Below, you can see the Laplacian we generated earlier next to $YY^\top$. Remember, each matrix contains information about our communities that the other doesn't have - and our goal is to combine them in a way that lets us distinguish between all three communities.

The way we'll combine the two matrices will simply be a weighted sum of these two matrices - this is what CASE is doing under the hood. The weight (here called α)

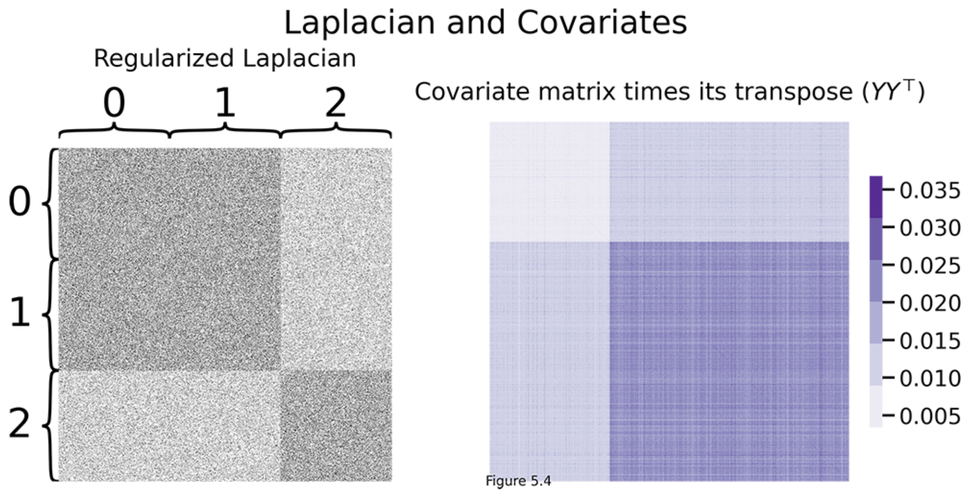


Figure 5-4. Laplacian and Covariates

is multiplied by $YY^\top$ - that way, both matrices contribute an equal amount of useful information to the embedding.

$$L + \alpha YY^\top$$

Weight Exploration

An obvious question here is how to weight the covariates. If we simply summed the two matrices by themselves, we'd unfortunately be in a situation where whichever matrix contained larger numbers would dominate over the other. In our current setup, without a weight on $YY^\top$, the covariates of our network would dominate over its topology.

What do different potential weights look like? Let's do a comparison. Below you can see the embeddings for 9 possible weights on $YY^\top$, ranging between 10^{-5} and 100.

It looks like we'd probably want a weight somewhere between 0.01 and 0.5 - then, we'll have three communities which are fairly distinct from each other, implying that we're pulling good information from both our network's topology and its covariates. We could just pick our weight manually, but it would be nice to have some kind of

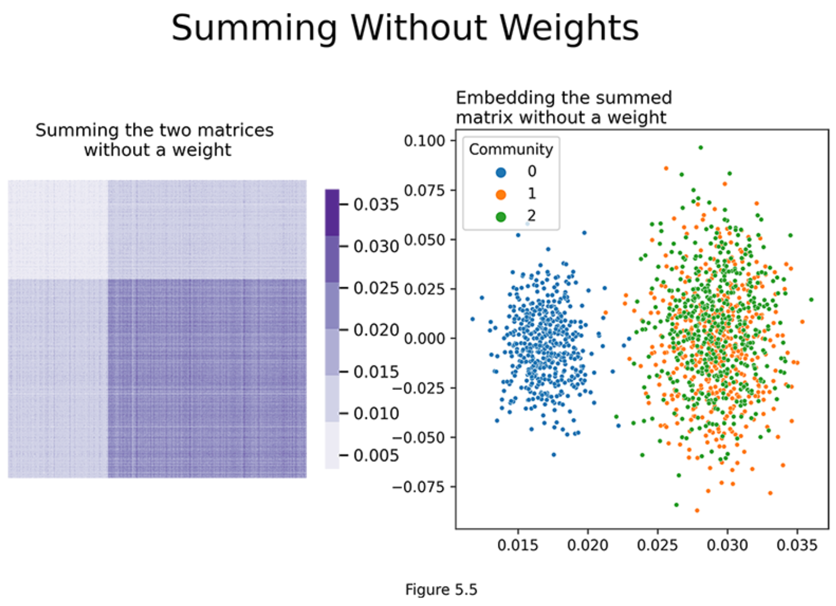


Figure 5-5. Summing Without Weights

algorithm or equation which lets us pick a reasonable weight automatically.

Weight Estimation

In general, we'd like to embed in a way that lets us distinguish between communities. This means that if we knew which community each node belonged to, we'd like to be able to correctly retrieve the correct commmunities for each node as possible with a clustering algorithm after embedding. This also implies that the communities will be as distinct as possible.

We already found a range of possible weights, embedded our combined matrix for every value in this range, and then looked at which values produced the best clustering. But, how do we find a weight which lets us consistently use useful information from both the topology and the covariates?

When we embed symmetric matrices, keep in mind that the actual points we're plotting are the components of the eigenvectors with the biggest eigenvalues. When we embed into two-dimensional space, for instance, the X-axis values of our points are the components of the eigenvector with the biggest eigenvalue, and the Y-axis values

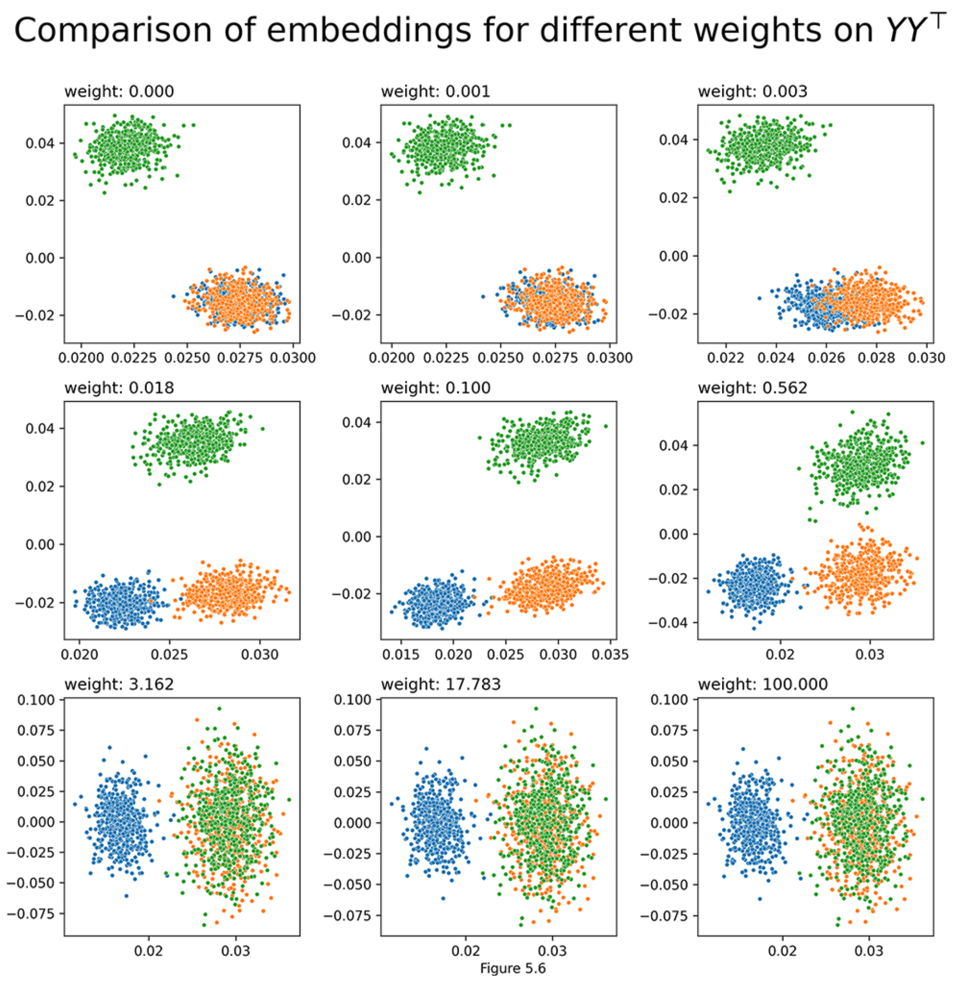


Figure 5-6. Comparison of Embeddings for different weights on $YY^\top$

are the components of the eigenvector with the second-biggest eigenvalue. This means that we should probably be thinking about how much information the Laplacian and $YY^\top$ contributes to the biggest eigenvalue/eigenvector pairs.

Thinking about this more, if we have a small weight, $YY^\top$ will contribute only a small amount to the biggest eigenvalue/vector pair. If we have a large weight, $YY^\top$ will contribute a large amount to the biggest eigenvalue/vector pair. The weight that causes the Laplacian and $YY^\top$ to contribute the same amount of information, then, is just the ratio of the biggest eigenvalue of L and the biggest eigenvalue of $YY^\top$:

$$weight = \frac{\lambda_1(L)}{\lambda_1(YY^\top)}$$

Let's check what happens when we combine our Laplacian and covariates matrix using the weight described in the equation above. Our embedding works the same as it does in Laplacian Spectral Embedding: we decompose our combined matrix using Singular Value Decomposition, truncating the columns, and then we visualize the rows of the result. Remember, we'll be embedding $L + \alpha YY^\top$, where α is our weight. We'll embed all the way down to two dimensions, just to make visualization simpler.

```
from scipy.sparse.linalg import eigsh

# Find the biggest eigenvalues in both of our matrices
leading_eigval_L, = eigsh(L, return_eigenvectors=False, k=1)
leading_eigval_YYt, = eigsh(YYt, return_eigenvectors=False, k=1)

# Per our equation above, we get the weight using
# the ratio of the two biggest eigenvalues.
weight = leading_eigval_L / leading_eigval_YYt

# Do our weighted sum, then embed
L_ = L + weight*YYt
latents_ = embed(L_, dimension=2)
```

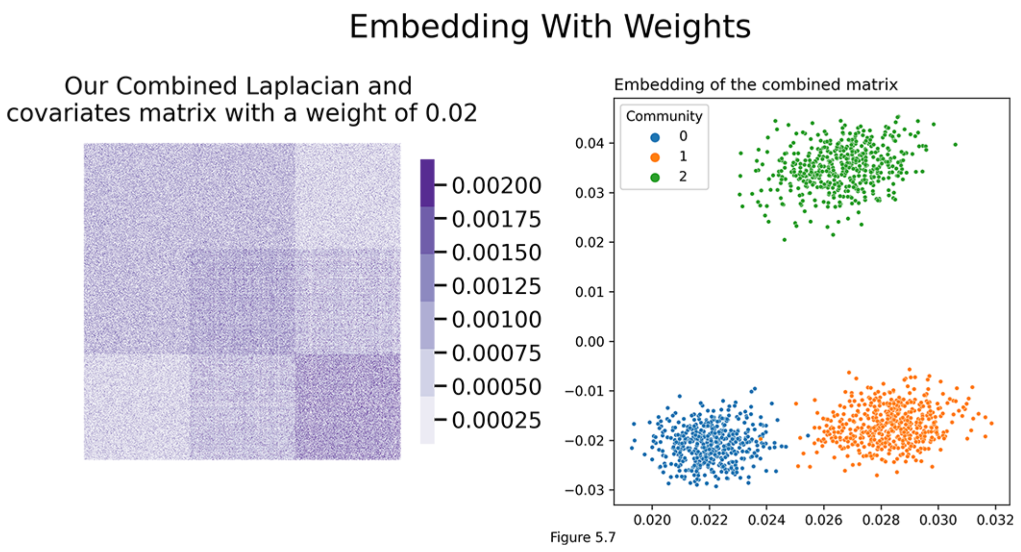


Figure 5-7. Embedding With Weights

Success! We've managed to achieve separation between all three communities. Below we can see (from left to right) a comparison of our network's estimated latent positions when we only use its topological information, when we only use the information contained in its covariates, and finally using the weight we found.

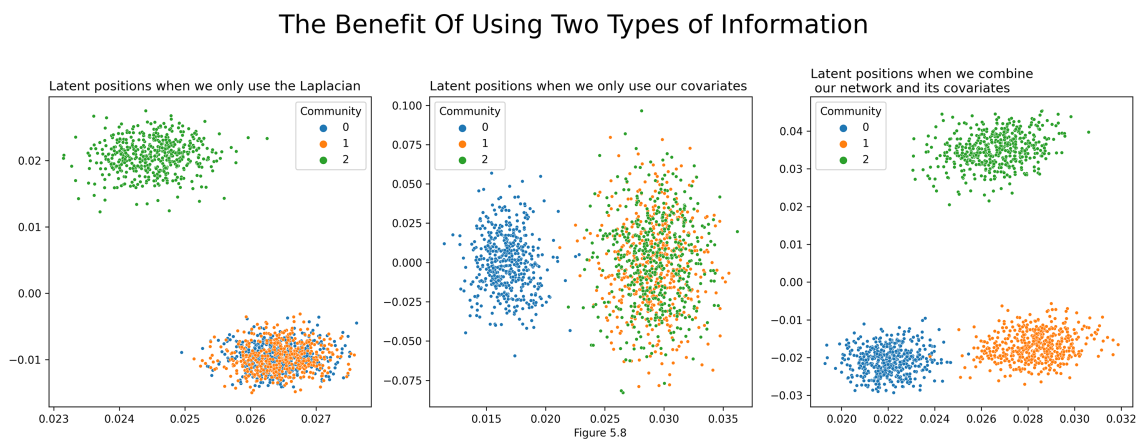


Figure 5-8. The Benefit Of Using Both Topology and Covariates

Using Graspologic

Graspologic's CovariateAssistedSpectralEmbedding class implements CASE directly. The following code applies CASE to reduce the dimensionality of $L + aYY^T$ down to two dimensions, and then plots the estimated latent positions to show the clustering.

Note that we don't always necessarily want to embed $L + \alpha YY^\top$. Using the regularized Laplacian by itself, for instance, isn't always best. If your network is *non-assortative* - meaning, the between-block probabilities are greater than the within-block probabilities - it's better to square our Laplacian. This is because the adjacency matrices of non-assortative networks have a lot of negative eigenvalues; squaring the Laplacian gets rid of a lot of annoying negative eigenvalues, and we end up with a better embedding. In the non-assortative case, we end up embedding $LL + aYY^\top$. The ‘embedding_alg‘ parameter controls this: you can write ‘embedding_alg="non-assortative"‘ if you're in the non-assortative situation.

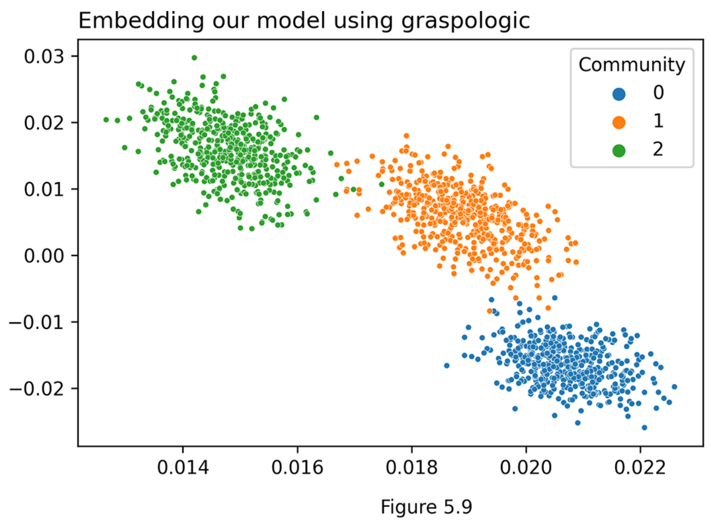


Figure 5-9. Embedding our model with CASE using graspologic

Omnibus Joint Embedding

If you've read the Multiple-Network Representation Learning section, you've seen the Omnibus Embedding (if you haven't read that section, you should go read it before reading this one!). To recap, the way the omnibus embedding works is:

1. Have a bunch of networks
2. Combine the adjacency matrices from all of those networks into a single, huge network
3. Embed that huge network

Remember that the Omnibus Embedding is a way to bring all of your networks into the same space (meaning, you don't run into any rotational nonidentifiability issues when you embed). Once you embed the Omnibus Matrix, it'll estimate a huge latent position matrix, which you can break apart along the rows to recover estimated latent positions for your individual networks.

You might be able to predict where this is going. What if we created an Omnibus embedding not with a set of networks, but with a network and covariates?

```
from graspologic.embed import OmnibusEmbed

# embed with Omni
omni = OmnibusEmbed()

# Normalize the covariates first
YYt = Y@Y.T
YYt /= np.max(YYt)

# Create a joint embedding
joint_embedding = omni.fit_transform([A, YYt])
```

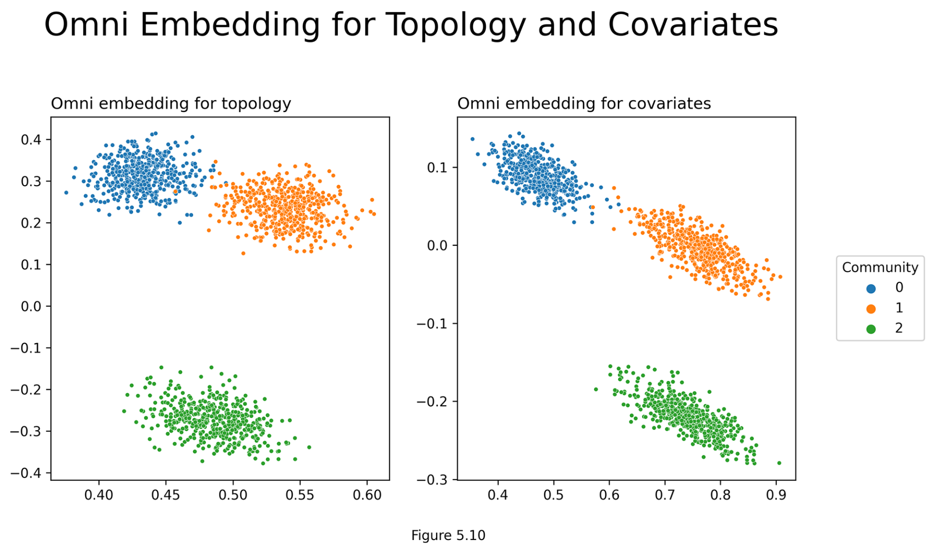


Figure 5-10. Omni Embedding for Topology and Covariates

There's a few things going on here. First, we had to normalize the covariates by dividing $YY^\top$ by its maximum. This is because if we didn't, the covariates and the adjacency matrix would contribute vastly different amounts of information to the omnibus matrix. You can see that by looking at the average value of $YY^\top$ compared to the average value of A:

```
print(r"Mean␣of␣Y@Y.T:", f"{np.mean(Y@Y.T):.2f}")
print(r"Mean␣of␣A:", f"{np.mean(A):.2f}")

>> Mean of Y@Y.T: 0.02
>> Mean of A: 0.23
```

They're completely different, which means that the two sets of values operate on completely different scales. Now, let's look at what happens when we normalize the covariates:

```
print(r"Mean␣of␣normalized␣Y@Y.T:", f"{np.mean(YYt):.2f}")
print(r"Mean␣of␣A:", f"{np.mean(A):.2f}")
```

```
>> Mean of normalized Y@Y.T: 0.42
>> Mean of A: 0.23
```

Remember the way this data is set up: you can separate the first community from the last two with the topology, you can separate the last community from the first two with the covariates, but you need both data sources to separate all three.

Here, you can see that both embeddings are able to separate all three communities. This is because the Omnibus Embedding induces dependence on the estimated latent positions it outputs. Remember that the off-diagonals of the Omnibus Matrix contain the averages of pairs of networks fed into it. These off-diagonal elements are responsible for some "information leakage": so the topology embedding contains information from the covariates, and the covariate embedding contains information from the topology.

MASE Joint Embedding

Just like you can use the OMNI to do a joint embedding, you can also use MASE to do a joint embedding. This fundamentally comes down to the fact that both embeddings fundamentally just eat matrices as their input - whether those matrices are the adjacency matrix or $YY^\top$ doesn't really matter.

Just like OMNI, we'll quickly recap how MASE works here:

1. Have a bunch of networks
2. Embed them all separately with ASE or LSE
3. Concate those embeddings into a single estimated latent position matrix with a lot more dimensions
4. Embed that new matrix

The difference here is the same as with Omni – we have the adjacency matrix (topology) and its covariates for a single network. So instead of embedding a bunch of adjacency matrices or Laplacians, we embed the adjacency matrix (or Laplacian) and the similarity matrix for the covariates $YY^\top$ separately, concatenate, and then embed again.

```python
from graspologic.embed import MultipleASE as MASE

# Remmeber that YY^T is still normalized!
mase = MASE(n_components=2)
joint_embedding = mase.fit_transform([A, YYt])
```

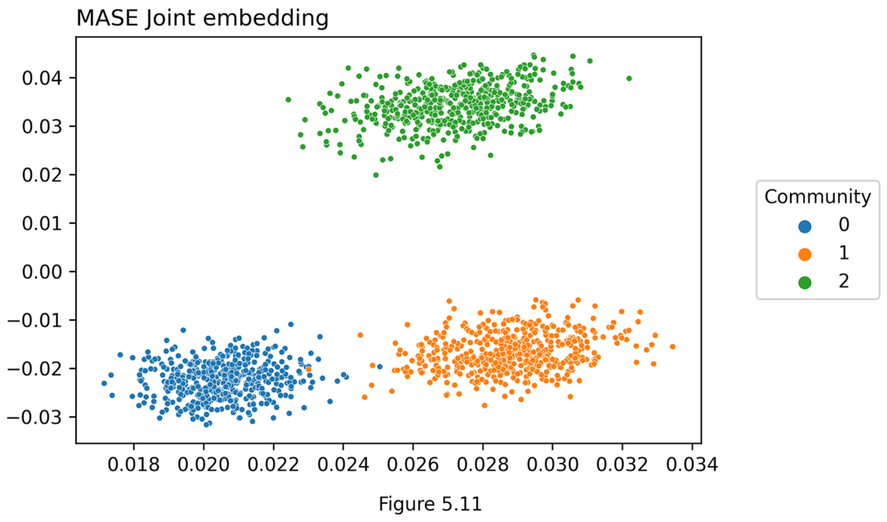


Figure 5-11. MASE Joint Embedding

As you can see, MASE lets us get fairly clear separation between communities. The covariates are still normalized, as with OMNI, so that they can contribute the same amount to the embedding as the adjacency matrix.

Chapter 6

Single-Network Vertex Nomination

Say you're a criminal investigator trying to uncover a human trafficking ring. You build a network of potential suspects: some of the nodes in the network represent human traffickers, and some represent innocent people. The edges of the network represent a working relationship between a given pair of individuals.

Your team has limited resources, and so it's difficult to scrutinize everybody in the network directly to see if they are human traffickers. Ideally, you'd like to use your network to nominate potential suspects, so that you can prioritize your investigative efforts. You've already done some work: you have a list of a few nodes of the network who are known to be traffickers, and you have a list of a few who you know are not. Your goal, then, is to build an ordered list of nodes in the network that are most similar to the nodes you already know belong to human traffickers. Ideally, the first nodes in the list would be more likely to be traffickers, and the nodes would get less and less likely the further down in the list you go.

This is the idea behind *single-network vertex nomination.* You have a group of "seed nodes" that you know have the right community membership, and then you take the rest of the nodes in your network and order them by their relationship to the seed nodes in terms of that community membership. The nomination task here isn't just classification: it's prioritization. You're prioritizing how important the rest of your nodes are with respect to the seed nodes, with the most important nodes at the top.

Spectral Vertex Nomination

There are a few approaches to vertex nomination. You can take a likelihood-maximization approach, or a bayes-optimal approach - but what we'll focus on is *Spectral Vertex Nomination*. The general idea is that you embed your network, and then you just order the estimated latent positions by how close they are to the seed node estimated latent positions. There are a few ways of doing this: you could create a *separate* set of nominees for each node, for instance. This would correspond to finding the people closest to *each* human trafficker, rather than finding a single list of nominees. You could also just get a single list of nominees: you could first take the centroid of the estimated latent positions of your seed nodes, and then find the closest nodes to that *centroid*. There are also a few different ways of defining what it means to be "close" to seed node estimated latent positions. The obvious way is euclidean distance, which is what you'd traditionally think of as the distance between two points, but you could also use something like the Mahalanobis distance, which is essentially Euclidean distance but with a coordinate system and a rescaling defined by the covariance in your data.

In any case, all forms of Spectral Vertex Nomination involve finding embeddings and then taking distances. In contrast to the other approaches, it scales well with very large networks (since you're essentially just doing an embedding followed by a simple calculation) and doesn't require any prior knowledge of community membership.

Let's see what spectral vertex nomination looks like. Below, we see the estimated latent positions for a network with three communities, where two of the communities are more closely linked than the third community. We do a standard adjacency spectral embedding, and we end up with a set of estimated latent positions. Our seed nodes - the ones whose community membership we know - are marked.

```
import numpy as np
from graspologic.simulations import sbm
from graspologic.embed import AdjacencySpectralEmbed as ASE

# construct network
n = 100
B = np.array([[0.5, 0.35, 0.2],
             [0.35, 0.6, 0.3],
             [0.2, 0.3, 0.65]])

# Create a network from and SBM, then embed
A, labels = sbm([n, n, n], p=B, return_labels=True)
ase = ASE()
X = ase.fit_transform(A)

# Let's say we know that the first five nodes belong to the first community.
# We'll say that those are our seed nodes.
seeds = np.ones(5)

# grab a set of seed nodes
memberships = labels.copy() + 1
mask = np.zeros(memberships.shape)
seed_idx = np.arange(len(seeds))
mask[seed_idx] = 1
memberships[~mask.astype(bool)] = 0

# estimate the latent positions for the seed nodes
seed_latents = X[memberships.astype(bool)]
```

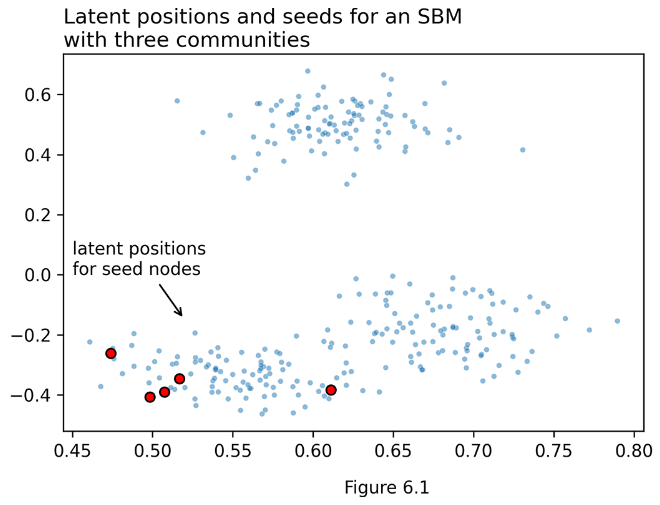


Figure 6-1. Estimated Latent Positions and Seeds for an SBM with three communities

Now, we'd like to order the rest of the vertices in this network by their degree of similarity to the seed nodes. Remember that we talked about two ways of doing this:

we could find a separate set of nominations for each seed node, or we could find a single set of nominations for all of the seed nodes. Let's start by finding a single set, using the centroid.

Finding a single set of nominations

Computing the centroid is as easy as just taking the mean value for the seed estimated latent positions along each coordinate axis. Since our example is in 2 dimensions, we can just take our $m \times 2$ matrix of seed estimated latent positions and take the mean along the first axis to create a 1×2 vector. That vector will be the centroid, and its location in Euclidean space will be right in the middle of the seeds. You can see the centroid (red star) along with the seed estimated latent positions (red circles) below.

```
centroid = seed_latents.mean(axis=0)
```

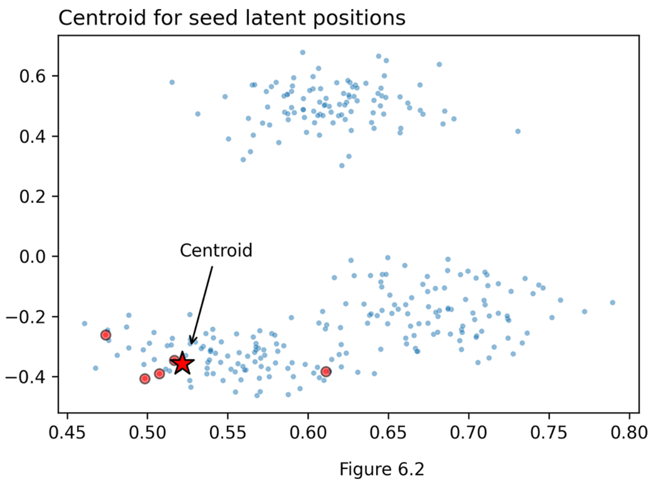


Figure 6-2. Centroid for seed estimated latent positions

Now, all we do is order the rest of the estimated latent positions (the blue dots in the figure above) by their distance to the centroid. The nodes corresponding to the closer estimated latent positions will be higher up in our nomination list. Scikit-learn has a 'NearestNeighbors' classifier, so we'll just use that. Below, we fit the classifier to our estimated latent positions matrix, then get our nominations using the 'kneighbors' function. The estimated latent positions closer to the centroid are more visible, and they get progressively less visible the further from the centroid they are.

```python
from sklearn.neighbors import NearestNeighbors

# Find the nearest neighbors to the seeds, excluding other seeds
neighbors = NearestNeighbors(n_neighbors=len(X))
neighbors.fit(X)
distances, nominations = neighbors.kneighbors(centroid[np.newaxis, :])
```

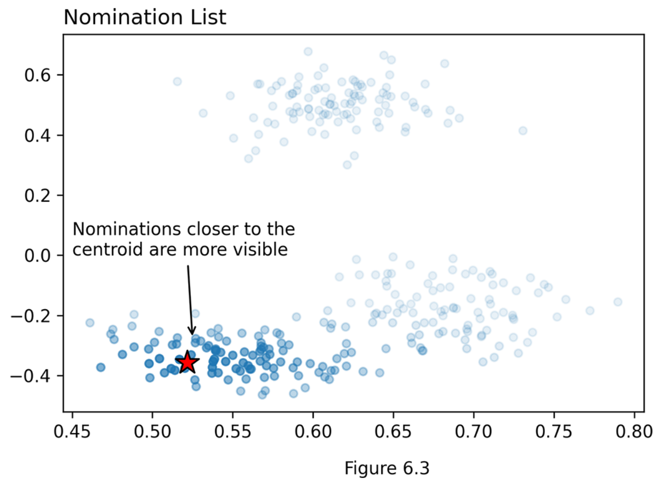


Figure 6-3. Nomination List

Let's look at the layout plot (also called a network plot) for the network directly, and see where our nominations tend to be. Below is a network colored by nomination rank: nodes that are higher up in the nomination list are more purple, and nodes that are lower in the nomination list are more white. You can see that the higher up in the nomination list you get (more purple), the more well-connected nodes tend to be to the seed nodes.

Finding Nominations for Each Node

Another approach, if we don't want to combine the information from all of our seed nodes, is to create a different nomination list for each node. This would correspond to finding multiple sets of people close to *each* human trafficker, rather than finding a single set of people for the *group* of human traffickers. Graspologic does this

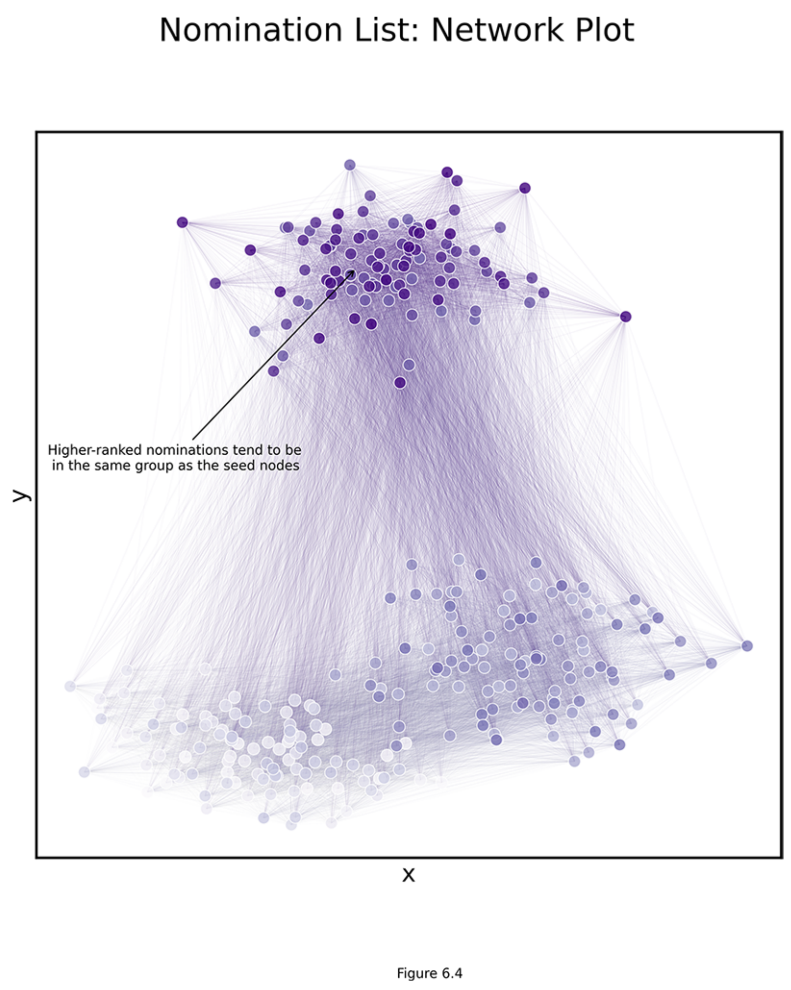


Figure 6-4. Nomination List: Network Plot

natively; the only real difference between the two approaches is that we take the nearest neighbors of the centroid for the first method rather than for each individual. Because of this, we'll just use graspologic directly, rather than showcasing the algorithm.

```python
from graspologic.nominate import SpectralVertexNomination

# Choose the number of nominations we want for each seed node
svn = SpectralVertexNomination(n_neighbors=5)
svn.fit(A)

# get nominations and distances for each seed index
nominations, distances = svn.predict(seed_idx)
```

Below you can see the nominations for each node. The first row containes the indices for each seed node, and each subsequent row contains the nearest neighbors for those seed nodes.

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 97 & 65 & 62 & 51 & 57 \\ 2 & 84 & 0 & 52 & 58 \\ 51 & 83 & 8 & 0 & 45 \\ 98 & 29 & 94 & 49 & 32 \end{bmatrix}$$

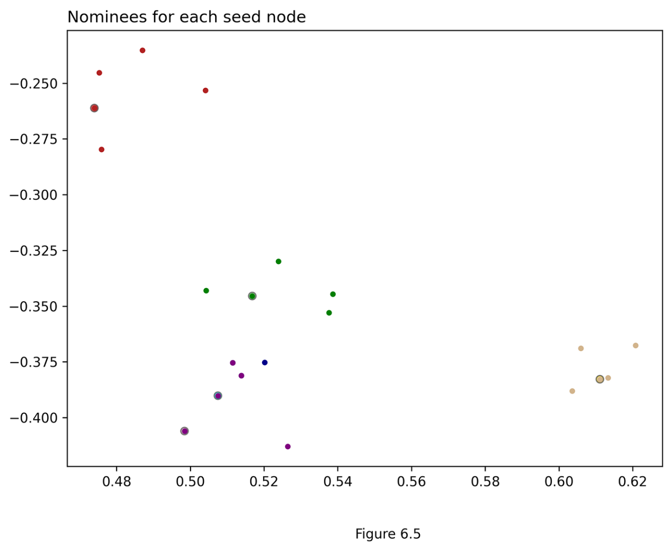


Figure 6-5. Nominations for each seed node

Chapter 7

Out-of-Sample Embedding

Imagine you have a citation network of scholars publishing papers. The nodes are the scholars, and an edge exists in a given pair of scholars if they're published a paper together.

You've already found a representation using ASE or LSE and you've estimated a set of latent positions, which you then clustered to figure out who came from which university. It took a long time for you to get this representation - there are a lot of people doing research out there!

Now, suppose a new graduate student publishes a paper. Your network gets bigger by a single node, and you'd like to estimate this person's latent position (thus adding them to the clustering system). To do that, however, you'd have to get an entirely new representation for the network: your estimated latent position matrix is $n \times d$, and it would need to become $(n+1) \times d$. Re-embedding the entire network with the new node added seems like it should be unecessary - after all, you already know the estimated latent positions for every other node.

This section is all about this problem: how to find the representation for new nodes without the computationally expensive task of re-embedding an entire network. As it turns out, there has been some work done, and there is a solution that can get you pretty close the estimated latent position for the new node that you would have had. For more details and formality, see the 2013 paper "Out-of-sample extension for latent position graphs", by Tang et al (although, as with most science, the theory in this paper was built on top of other work from related fields).

Let's observe a network from an SBM, as well as an additional node that should belong to the first community. Then, we'll embed the network and explore how to estimate the latent position for the additional node.

Data Generation

```
import numpy as np
from graspologic.simulations import sbm
from graspologic.utils import remove_vertices

# Generate parameters
B = np.array([[0.8, 0.2],
             [0.2, 0.8]])

# Generate both an original network along with community memberships,
# and an out-of-sample node with the same SBM call
network, labels = sbm(n=[101, 100], p=B, return_labels=True)
labels = list(labels)

# Grab out-of-sample node
oos_idx = 0
oos_label = labels.pop(oos_idx)

# create our original network
A, a_1 = remove_vertices(network, indices=oos_idx, return_removed=True)
```

What we have now is a network and an additional node. You can see the adjacency matrix for the network below, along with the adjacency vector for the additional node (Here, an "adjacency vector" is a vector with a 1 in every position that the out-of-sample node has an edge with an in-sample node). The heatmap on the left is a network with two communities, with 100 nodes in each community. The vector on the right is purple on row i if the i^{th} in-sample node is connected to the out-of-sample node.

After embedding with ASE, we have an embedding for the original network. The rows of this embedding contain the estimated latent position for each original node.

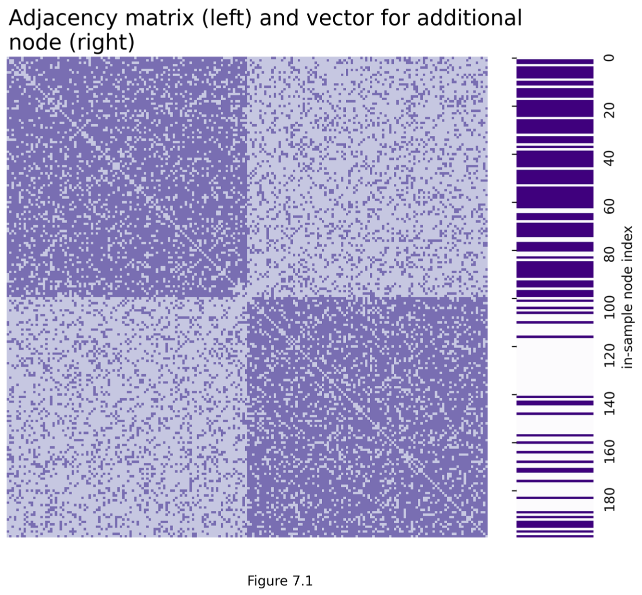


Figure 7-1. Adjacency Matrix (left) and vector for additional node (right)

We'll call the embedding X.

```
from graspologic.embed import AdjacencySpectralEmbed as ASE

ase = ASE(n_components=2)
ase.fit(A)
X = ase.transform(A)
```

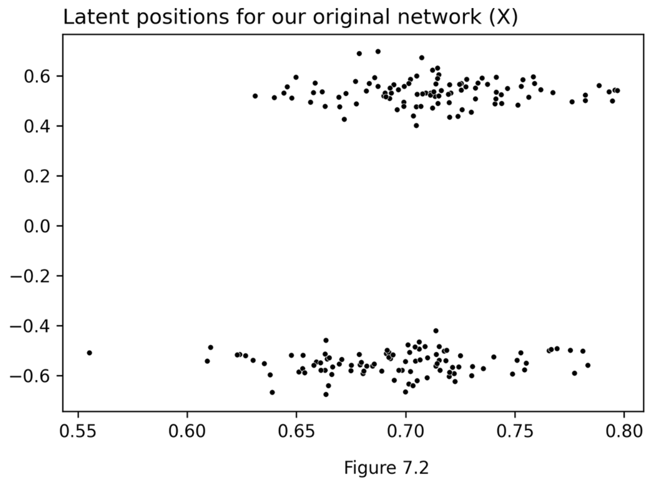


Figure 7-2. Adjacency Matrix (left) and vector for additional node (right)

Probability Vector Estimation

Everything up until now has just been pretty standard stuff. We still haven't done anything with our new node - all we have is a big vector that tells us which other nodes it's connected to, and our standard matrix of estimated latent positions. However, it's time for a bit more exploration into the nature of the latent position matrix X, and what happens when you view it as a linear transformation. This will get us closer to understanding the out-of-sample embedding.

Remember from the section on latent positions that X can be used to estimate the block probability matrix. When you use ASE on a single network to estimate X, by finding $\hat{X}$, $\hat{X}\hat{X}^\top$ estimates P: meaning, $(\hat{X}\hat{X}^\top)_{ij}$, the element on the $i^{(th)}$ row and $j^{(th)}$ column of $\hat{X}\hat{X}^\top$, will estimate the probability that node i has an edge with node j.

Let's take a single estimated latent position vector - call it v_i (this will be the i_{th} row of the estimated latent position matrix). What will $\hat{X}v_i$ look like? Well, it'll look the same as grabbing the i_{th} column of $\hat{X}\hat{X}^\top$. Meaning, $\hat{X}v_i$ will be a single vector whose $j^{(th)}$ element estimates the probability that node i will connect to node j.

You can see this in action below. We took the latent position corresponding to the first node out of the estimated latent position matrix (and called it v_1), and then multiplied it by the estimated latent position matrix itself. What emerged is what you see below: a vector that estimates the probability that node 0 has an edge with each other node in the network. The true probabilities for the first half of nodes (the ones in the same community) should be .8, and the true probabilities for the second half of nodes in the other community should be .2. The average estimations were .775 and .149 - so, pretty close!

```
v_1 = X[0, :]
v_est_proba = X @ v_1
```

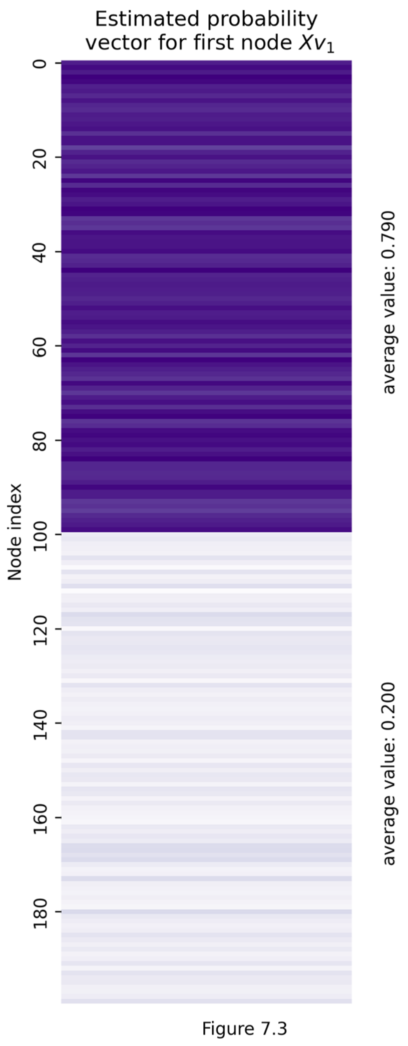


Figure 7-3. Estimated probability vector for first node X_{v_1}

Inversion of Probability Vector Estimation

Remember that our goal is to take the adjacency vector for a new node and use it to estimate that node's latent position without re-embedding the whole network. So far, we've essentially figured out how to use the node's estimated latent position to estimate a probability vector.

Let's think about the term "estimated probability vector" for a second. This should be a vector associated to node i with n elements, where the $j^{(th)}$ element of the vector contains the probability that node i will connect to node j. The thing we're starting with for the out-of-sample node, however, is an adjacency vector full of 0's and 1's - 0 if there isn't an edge, 1 if there is an edge.

If you think about it, however, you can think of this adjacency vector as kind of an estimate for edge probabilities. Say you sample a node's adjacency vector from an RDPG, then you sample again, and again. Averaging all of your samples will get you closer and closer to the actual edge connection probabilities. So you can think of a single adjacency vector as an estimate for the edge probability vector!

The point here is that if you can start with a latent position estimate and then estimate the edge probabilities, it's somewhat equivalent (albeit going in the other direction) to start with an out-of-sample adjacency vector and then estimate a node's the latent position.

Let's call the estimated probability vector $\hat{a_i}$. We know that $\hat{a_i} = \hat{X}\hat{v_i}$: you multiply the estimated latent position matrix by the i_{th} estimated latent position to estimate the probability vector (remember that the $\hat{}$hats above letters means we're getting an estimate for something, rather than getting the thing itself). How do we isolate the estimated latent position $\hat{v_i}$?

Well, if X were invertible, we could do $\hat{X}^{-1}\hat{a_i} = \hat{v_i}$: just invert both sides of the equation to get v_i by itself. Unfortunately, in practice, $\hat{X}$ will almost never be invertible. We'll have to do the next-best thing, which is to use the *pseudoinverse.*

We'll take a brief break in the coming section to talk about the pseudoinverse for a bit, then we'll come back and use it to estimate the out-of-sample latent position.

The Moore-Penrose Pseudoinverse

The Moore-Penrose Pseudoinverse is useful to know in general, since it pops up a lot in a lot of different places. Say you have a matrix which isn't invertible. Call it T.

The pseudoinverse T^+ is the closest approximation you can get to the inverse T^{-1}.

This is best understood visually. Let's take T to be a matrix which projects points on the x-y coordinate axis down to the x-axis, then flips them to their negative on the number line. The matrix would look like this:

$$T = \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix}$$

Some information is inherently lost here. Because the second column is all zeroes, any information in the y-axis can't be recovered. For instance, say we have some vectors with different x-axis and y-axis coordinates:

$$v_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}^\top$$
$$v_2 = \begin{bmatrix} 2 & 2 \end{bmatrix}^\top$$

When we use T as a linear transformation to act on v_1 and v_2, the y-axis coordinates both collapse to the same thing (0, in this case). Information in the x-axis, however, is preserved.

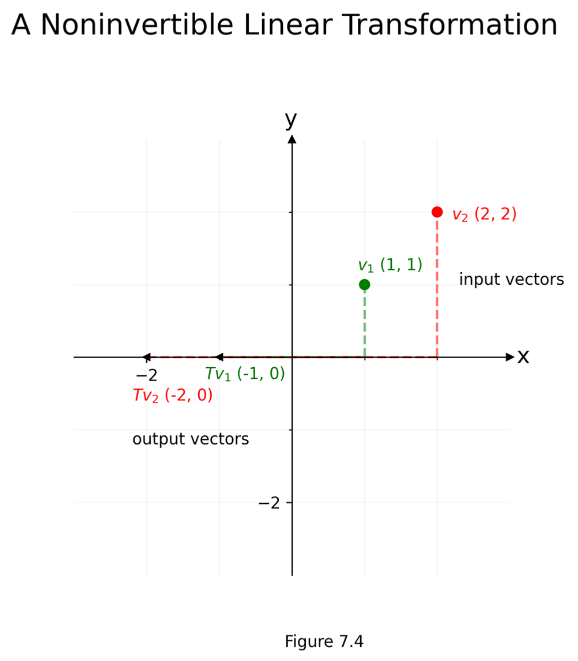


Figure 7-4. A Noninvertible Linear Transformation

Our goal is to reverse T and bring Tv_1 and Tv_2 back to v_1 and v_2. Unfortunately, since both v_1 and v_2 get squished onto zero in the y-axis position after getting passed through T, we've lost all information about what was happening on the y-axis – that's a lost cause. So it's impossible to get perfectly back to v_1 or v_2.

If you restrict your attention to the x-axis, however, you'll see that Tv_1 and Tv_2 landed in different places (v_1 went to -1, and v_2 went to -2). You can use this information about the x-axis location of Tv_1 and Tv_2 to re-orient the x-axis values back to where they were prior to the vectors getting passed through X, even if it's impossible to figure out where the y-values were.

That's what the pseudoinverse does: it reverses what it can, and accepts that some information has vanished.

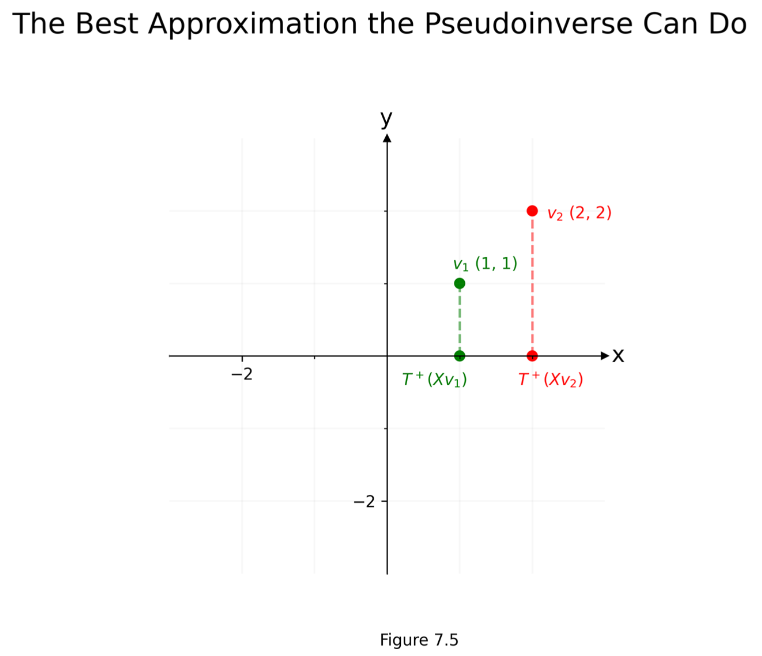


Figure 7-5. The Best Approximation the Pseudoinverse Can Do

Using the Pseudoinverse for Out-of-Sample Estimation

Let's get back to estimating our out-of-sample latent position.

Remember that we had a nonsquare latent position matrix estimate $\hat{X}$. Like we learned before, we can estimate the probability vector a_i (the vector with its probability of connecting with node j in the j_{th} position) for a node by passing its estimated latent position ($\hat{v}_i$) through the estimated latent position matrix.

$$\hat{a}_i = \hat{X}\hat{v}_i$$

We can think of $\hat{X}$ as a matrix the same way we thought of T: it's a linear transformation that eats a vector, and doesn't necessarily preserve all the information about that vector when it outputs something (In this case, since $\hat{X}$ brings lower-dimensional latent position estimates to higher-dimensional probability vector estimates, what's happening is more of a restriction on which high-dimensional vectors you can access than a loss of information, but that's not particularly important).

The pseudoinverse, $\hat{X}^+$, is the best we can do to bring a higher-dimensional adjacency vector to a lower-dimensional latent position estimate. Since the adjacency vector just estimates the probability vector, we can call it $\hat{a_i}$. In practice, the best we can do generally turns out to be a pretty good guess, and so we can get a decent estimation of the true latent position v_i.

$$X^+\hat{a_i} \approx X^+(Xv_i) \approx v_i$$

Let's see it in action. Remember that we already grabbed our out-of-sample latent position and called it 'a_1'. We use numpy's pseudoinverse function to generate the pseudoinverse of the estimated latent position matrix. Finally, we use it to get ‘a_1‘’s estimated latent position, and call it ‘v_1‘. You can see the location of this estimate in Euclidean space below: it falls squarely into the first community, which is where it should be.

```python
from numpy.linalg import pinv

# Make the pseudoinverse of the latent position matrix
X_pinverse = pinv(X)

# Get its estimated latent position
v_1 = X_pinverse @ a_1
```

‘v_1‘ is the array ‘[0.731, 0.504]‘. You can see the estimated location for the out-of-sample latent position in fig 7-6 below.

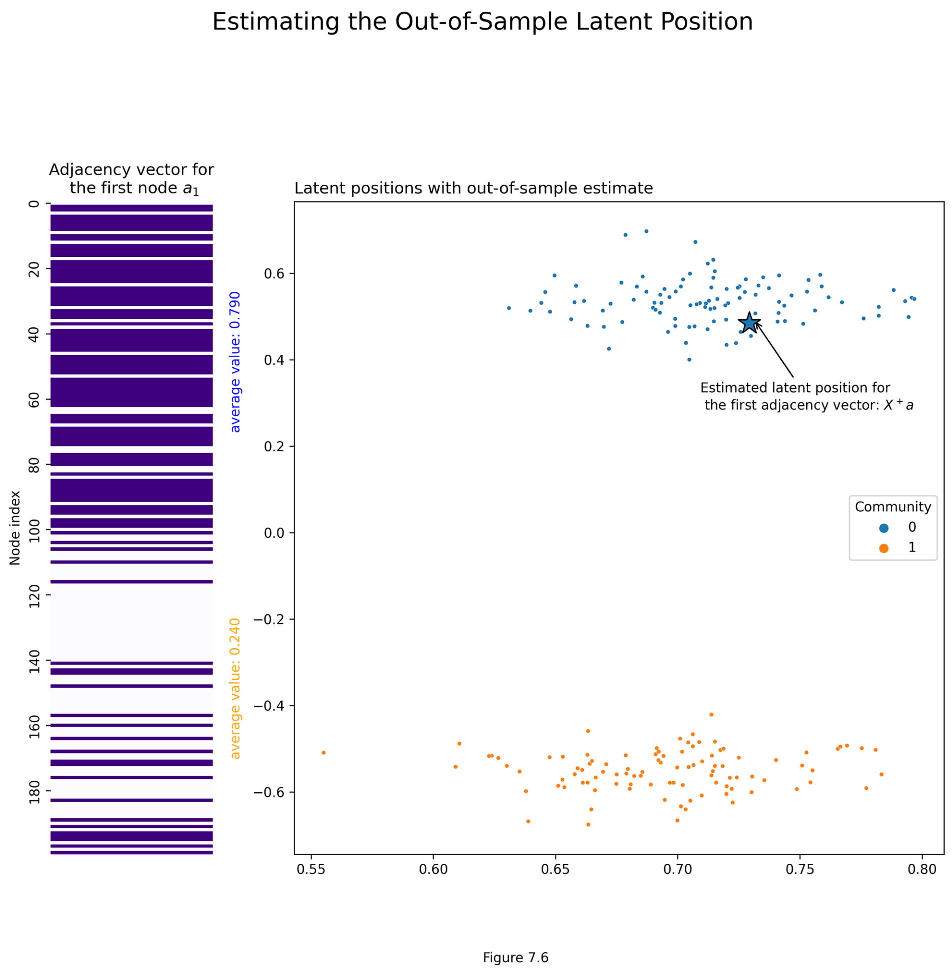


Figure 7-6. Estimated Latent Positions with out-of-sample estimate

Using Graspologic

Of course, you don't have to do all of this manually. Below we observe an adjacency matrix A from an SBM, as well as the adjacency vector for an out-of-sample node a_1. Once we fit an instance of the ASE class, the latent position for any new out-of-sample nodes can be estimated by simply calling 'ase.transform' on the new adjacency vectors.

You can do the same thing with multiple out-of-sample nodes if you want by stacking their adjacency vectors on top of each other in a numpy array, then transforming the whole stack.

```
from graspologic.embed import AdjacencySpectralEmbed as ASE

# Generate parameters
B = np.array([[0.8, 0.2],
              [0.2, 0.8]])

# Generate a network along with community memberships
network, labels = sbm(n=[101, 100], p=B, return_labels=True)
labels = list(labels)

# Grab out-of-sample vertex
oos_idx = 0
oos_label = labels.pop(oos_idx)
A, a_1 = remove_vertices(network, indices=oos_idx, return_removed=True)

# Make an ASE model
ase = ASE(n_components=2)
X = ase.fit_transform(A)

# Estimate out-of-sample latent position(s) by transforming
v_1 = ase.transform(a_1)
```

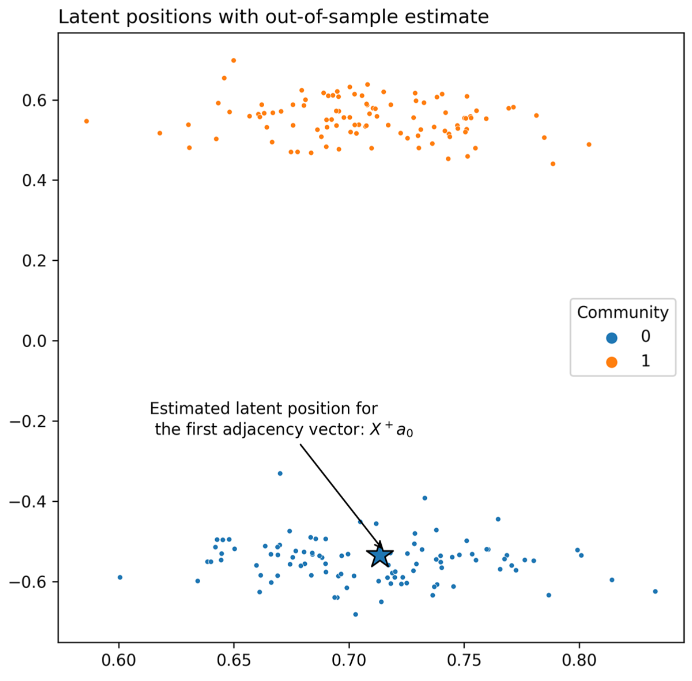


Figure 7.7

Figure 7-7. Estimated Latent positions with out-of-sample estimate

Chapter 8

Anomaly Detection for Timeseries of Networks

There is a particular type of sea slug which has gills on the outside of its body. When you squirt water at these gills, they withdraw into the slug. The interesting thing about this type of slug is that the brain network involved in this gill withdrawal reflex is entirely mapped out, from the neurons which detect and transmit information about the water into the slug's brain, to the neurons that leave the brain and fire at its muscles. (For those interested, this is a real thing - look up Eric Kandel's research on Aplysia!)

Say you're a researcher studying these sea slugs, and you have a bunch of brain networks of the same slug. We can define each node as a single neuron, and edges denote connections between neurons. Each of the brain networks that you have were taken at different time points: some before water started getting squirted at the slug's gills, and some as the water was getting squirted. Your goal is to reconstruct when water started to get squirted, using only the networks themselves. You hypothesize that there should be some signal change in your networks which can tell you the particular time at which water started getting squirted. Given the network data you have, how do you figure out which timepoints these are?

The broader class of problems this question addresses is called *anomaly detection*.

The idea, in general, is that you have a bunch of snapshots of the same network over time. Although the nodes are the same, the edges are changing at each time point. Your goal is to figure out which time points correspond to the most change, either in the entire network or in particular groups of nodes. You can think of a network as "anomalous" with respect to time if some potentially small group of nodes within the network concurrently changes behavior at some point in time compared to the recent past, while the remaining nodes continue with whatever noisy, normal behavior they had.

In particular, what we would really like to do is separate the signal from the noise. All of the nodes in the network are likely changing a bit over time, since there is some variability intrinsic in the system. Random noise might just dictate that some edges get randomly deleted and some get randomly created at each step. We want to figure out if there are timepoints where the change isn't just random noise: we're trying to figure out a point in time where the probability distribution that the network *itself* is generated from changes.

Let's simulate some network timeseries data so that we can explore anomaly detection more thoroughly.

Simulating Network Timeseries Data

For this data generation, we're going to assemble a set of 12 time-points for a network directly from its true latent positions (we'll assume that each time-point for the network is drawn from an RDPG). Ten of these time points will just have natural variability, and two will have a subset of nodes whose latent positions were perturbed a bit. These two will be the anomalies.

We'll say that the latent positions for the network are one-dimensional, and that it has 100 nodes. There will be the same number of adjacency matrices as there are time points, since our network will be changing over time.

To make the ten non-anomalous time points, we'll:

1. Generate 100 latent positions. Each latent position will be a (uniformly) random number between 0.2 and 0.8.
2. Use graspologic's rdpg function to sample an adjacency matrix using these latent positions. Do this ten times.

And to make the two perturbed time points, we'll do the following twice:

1. Add a small amount of noise to the first 20 latent positions that we generated above.
2. Generate an adjacency matrix from this perturbed set of latent positions.

Once we have this simulated data, we'll move into some discussion about how we'll approach detecting the anomalous time points.

Below is code for generating the data. We define a function to generate a particular time-point, with an argument which toggles whether we'll perturb latent positions for that time point. Then, we just loop through our time-points to sample an adjacency matrix for each one.

```python
import numpy as np
from graspologic.simulations import rdpg


def gen_timepoint(X, perturbed=False, n_perturbed=20):
    if perturbed:
        X = np.squeeze(X)
        baseline = np.array([1, -1, 0])
        delta = np.repeat(baseline, (n_perturbed//2,
                                     n_perturbed//2,
                                     nodes-n_perturbed))
        X += (delta * .15)
    if X.ndim == 1:
        X = X[:, np.newaxis]
    A = rdpg(X)
    return A


time_points = 12
nodes = 100
X = np.random.uniform(.2, .8, size=nodes)
networks = []

for time in range(time_points - 2):
    A = gen_timepoint(X)
    networks.append(A)

for perturbed_time in range(5, 7):
    A = gen_timepoint(X, perturbed=True)
    networks.insert(perturbed_time, A)

networks = np.array(networks)
```

You can see the adjacency matrices we generated below. Note that you can't really distinguish a difference between the ten normal time points and the two perturbed time points with the naked eye, even though the difference is there, so it would be pretty difficult to manually mark the time points - and if you have many time points, rather than just a few, you'd want to be able to automate the process.

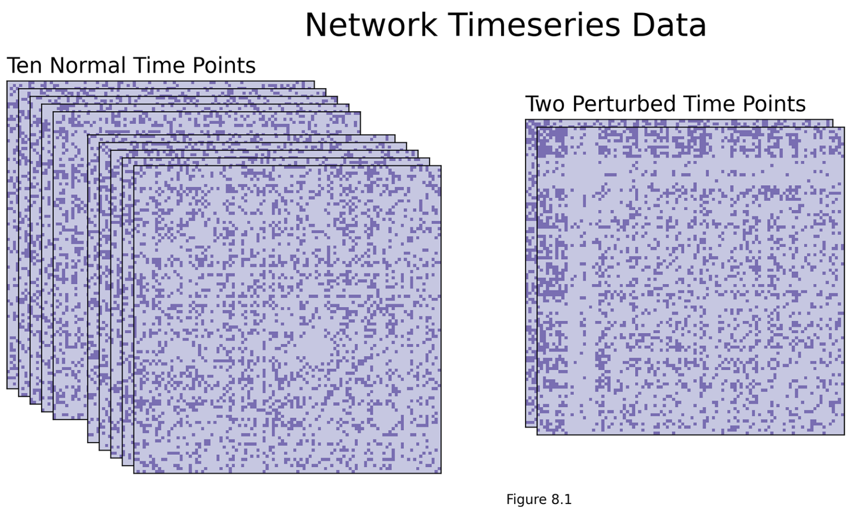


Figure 8-1. Network Timeseries Data

Approaches for Anomaly Detection

It's time to start thinking about how we'd approach figuring out which of the time points are anomalies.

One of the simplest approaches to this problem might just be to figure out which node has the highest count of edge changes across your timeseries. For each node across the timeseries, you'd count the number of new edges that appeared (compared to the previous point in time), and the number of existing edges that were deleted. Whichever count is highest could be your anomalous node.

This might give you a rough estimate – and you could even potentially find perturbed time points with this approach – but it's not necessarily the best solution. Counting edges doesn't account for other important pieces of information: for instance, you might be interested in which other nodes new edges were formed with. It seems like deleting or creating edges with more important nodes, for instance, should be weighted higher than deleting or creating edges with unimportant nodes.

So let's try another method. You might actually be able to guess it! The idea will be to simply estimate each network's latent positions, followed by a hypothesis testing approach. Here's the idea.

Let's call the latent positions for our network $X^{(t)}$ for the snapshot of the network at time t. You're trying to find specific time points, $X^{(i)}$, which are different from their previous time point $X^{(i-1)}$ by a large margin. You can define "different" as "difference in matrix norm". Remember that the matrix norm is just a number that generalizes the concept of vector magnitude to matrices. In other words, We're trying to find a time point where the difference in norm between the latent positions at time t and the latent positions at time $t-1$ is greater than some constant: $||X^{(t)} - X^{(t-1)}|| > c$. The idea is that non-anomalous time points will probably be a bit different, but that the difference will be within some reasonable range of variability.

There's an alternate problem where you restrict your view to *nodes* rather than entire adjacency matrices. The idea is that you'd find time-points which are anomalous for particular nodes or groups of nodes, rather than the entire network. The general idea is the same: you find latent positions, then test for how big the difference is between time point t and time point $t-1$. This time, however, your test is for particular nodes. You want to figure out if $||X_i^{(t)} - X_i^{(t-1)}|| > c$, where you're looking at a particular latent position X_i rather than all of them at once. We'll be focusing on the problem for whole networks, but you can take a look at the original paper if you're curious about how to apply it to nodes. We will, of course, be using *estimated* latent positions rather than the true latent positions, since in practice we would never have access to the true latent positions.

Detecting if the First Time Point is an Anomaly

We'll start with the first time point, which (because we generated the data!) we know in advance is not an anomaly.

If we were to just estimate the latent positions for each timepoint separately with ASE or LSE, we'd run into the nonidentifiability problem that we've seen a few times over the course of this book: The latent position estimates would be rotated versions of each other, and we'd have to use something like Procrustes (which adds variance, since it's just an estimate) to rotate them back into the same space.

However, since we have multiple time points, each of which is associated to an adjacency matrix, it's natural to use models from the Multiple-Network Representation Learning section (You can go back and read chapter 6.7 if you're fuzzy on the details here). In that section, we introduced the Omnibus Embedding as a way to estimate latent positions for multiple *networks* simultaneously, but all we really need for it is multiple *adjacency matrices*. These exist in our network in the form of its multiple time points; So, we'll just embed multiple time points at once with the Omnibus Embedding, and then they'll live in the same space.

We only really *need* to embed two time points at a time, since all we really care about is being able to directly compare a time point $X^{(t)}$ and the point prior to it $X^{(t-1)} = Y$ - but because of the way Omni works, we'll get smaller-variance estimates if we embed all the time points at once. Embedding them all at once also to be more robust to embedding dimension in practice. If you wanted to save computational power - for instance, if you had a lot of time points - you could instead choose to embed subsets of them, or just the two you'll actually be using.

So, here's what's going on in the code below:

1. We embed the time points using OMNI and then get our estimates for the first two sets of latent positions $\hat{X} = \hat{X}^{(t)}$ and $\hat{Y} = \hat{X}^{(t-1)}$.
2. Then, we get the norm of their difference $||\hat{X} - \hat{Y}||$ with numpy.

An important point to clarify is that there are a lot of different types of matrix norms: Frobenius norm, spectral norm, and so on. In our case, we'll be using the l_2 operator norm, which is simply the largest singular value of the matrix. The ‘ord‘ parameter argument in numpy determines which norm we use, and ‘ord=2‘ is the operator norm.

Again, this norm, intuitively, will tell us how different two matrices are. If the norm of the difference between the true latent positions $X - Y$ is small, then X and Y are very similar matrices; whereas if the norm of $X - Y$ is large, then X and Y are very different. The norm should be large for anomalies, and small for everything else.

```python
from graspologic.embed import OmnibusEmbed as OMNI

def get_statistic(adjacencies, return_latents=False):
    """
    Get the operator norm of the difference of two matrices.
    """
    omni = OMNI(n_components=1)
    latents_est = omni.fit_transform(adjacencies)
    Xhat, Yhat = latents_est[0], latents_est[1]
    yhat = np.linalg.norm(Xhat - Yhat, ord=2)
    if return_latents:
        return yhat, Xhat
    else:
        return yhat

yhat, Xhat = get_statistic(networks, return_latents=True)
```

Hypothesis Testing With our Test Statistic

We have our norm y, which will be our test statistic. It should be a small value if the first two adjacency matrices in the timeseries are distributed the same, and large if they're distributed differently. Remember that we're fundamentally trying to figure out whether $X = X^{(t)}$, our latent positions at time t, is the same as $Y = X^{(t-1)}$, our latent positions at time $t-1$. This is also known as a *hypothesis test*: we're testing the the null hypothesis that $X = Y$ against the alternative hypothesis that $X \neq Y$.

The value of our test statistic is 0.633. The problem is that we don't know how big this is, relatively. Is 0.633 relatively large? small? how should we determine whether it's small enough to say that X and Y probably come from the same distribution, and aren't anomaly time points?

Well, what if we could use our estimated latent positions $\hat{X}$ at time t to generate a bunch of networks, then make test statistics from those new networks? We'd know for a fact that any pair of those networks are drawn from the same set of true latent positions, and we could get a sense for what our test statistic should look like if the latent positions actually were the same. This technique is called *bootstrapping*, since you're using estimated parameters to "pull yourself up by your own bootstraps" and generate a bunch of artificial data. Bootstrapping pops up all over the place in machine learning and statistics contexts.

Bootstrapped Distribution Estimation

We don't have the true latent positions for a given time point, but we do have the estimated latent positions (we just used OMNI embedding to find them!)

So what we can do is the following:

1. Using a set of the estimated latent positions we just found, generate two new adjacency matrices.
2. Get the test statistic for these two adjacency matrices.
3. Repeat 1) and 2) a bunch of times, getting new test statistics each time
4. Look at the distribution of these test statistics, and determine whether y is an outlier or not with respect to this distribution.

So we're artificially generating data that we *know for a fact* is distributed in exactly the same way, and then looking at how our test statistic is distributed under those assumptions. This artificial data will necessarily be a bit biased, since the latent positions you're using to generate it are themselves only estimates, but it should be close enough to the real thing to be useful.

Below is some code. We generate 1000 pairs of adjacency matrices from our estimated latent positions for the first time point $\hat{X}^{(t)}$, and get the test statistic for each pair. Underneath this looping code, you can see the distribution of these bootstrapped test statistics in the form of a histogram. They look roughly normally distributed, and hover around 0.60. The red line shows where our actual test statistic lies, where we compare $\hat{X}^{(t)}$ to $\hat{X}^{(t-1)}$.

If the red line is super far away from the bulk of the mass in the test statistic distribution, then it would be fairly unlikely to be drawn from the same set of latent positions as the bootstrapped test statistics, and we'd reject the hypothesis that it is.

If it's well within the range of values we'd reasonably expect, then we wouldn't reject this possibility.

```python
# null hypothesis that X = Y. Bootstrap X.
N = 1000
bootstraps = []
for est in range(N):
    A_est = rdpg(Xhat)
    B_est = rdpg(Xhat)
    bootstrapped_y = get_statistic([A_est, B_est])
    bootstraps.append(bootstrapped_y)
bootstraps = np.array(bootstraps)
```

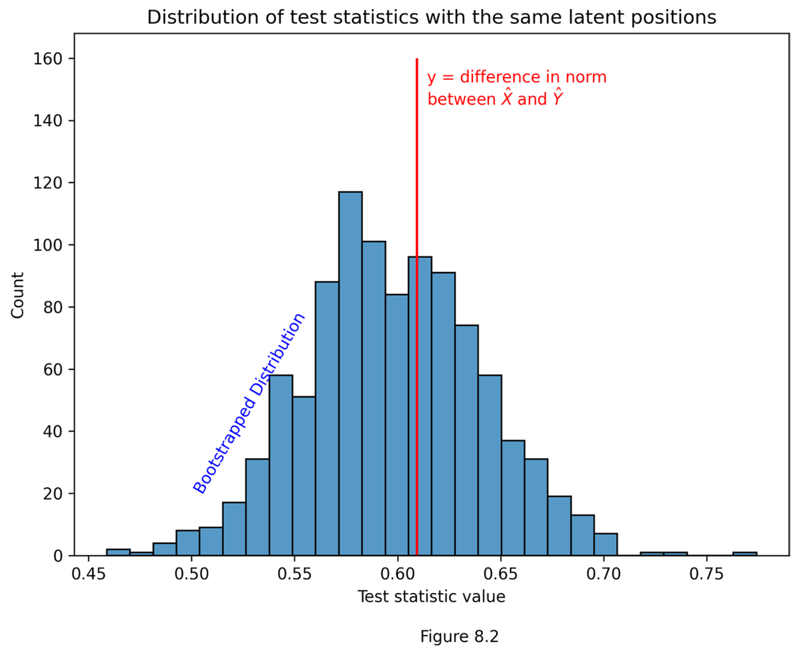


Figure 8-2. Distribution of Test Statistics with the same true latent positions

Fortunately, 0.633 is well within a reasonable range under the assumption that the time-points share true latent positions. However, we can't always eyeball stuff, and we need a way to formalize what it means for a test statistic to be "within a reasonable range". Our test statistic is $y = ||\hat{X}^{(t)} - \hat{X}^{(t-1)}||$, we're trying to figure out if $X^{(t)} = X^{(t-1)}$, and we have a bunch of bootstrapped test statistics that we know were drawn from the same distribution (and are thus examples of the case where the null hypothesis is true).

P-Value Estimation

Since we have a range of examples of y values in which the null hypothesis is true, we have an estimate for the distribution of the null hypothesis. So, to find the probability that any new value drawn from this bootstrapped distribution is greater than a particular value c, we can just find the proportion of our bootstrapped values that are greater than c.

$$p = \frac{\text{number of bootstrapped values greater than } c}{\text{total number of bootstrapped values}}$$

When we let c be equal to our test statistic, y, we find the probability that any new bootstrapped value will be greater than y (assuming that y is drawn from the null distribution). Here we have our formalization.

Below is some simple numpy code that performs this estimation. We just count the number of bootstrapped statistics that are greater than our y value, and then divide by the number of bootstrapped test statistics. If the resulting p-value is less than some pre-determined probability (say, for instance, 0.05), then we reject the null hypothesis and say that y probably comes from a different distribution than the bootstrapped statistics. This, in turn, implies that $X^{(t)} \neq X^{(t-1)}$, and we've found an anomaly time point.

```
p = (bootstraps > y).sum() / N
```

Our p value is 0.327, which is much larger than 0.05. Therefore, we don't reject the null hypothesis, and we can conclude that we haven't found an anomaly time. Since this is all synthetic data, we know how the data generation process worked, so we actually know for a fact that this is the right result – the adjacency matrix at time t actually *was* drawn from the same distribution as the adjacency matrix at time $t-1$.

Testing the Remaining Time Points

Now that we've gone through this for one time point, we can do it for the rest. The process is exactly the same, except that you're comparing different pairs of timepoints and you're generating the bootstrapped test statistics with different estimated latent positions.

Below we get our test statistic for every pair of time points. Our two anomaly time points are drawn from the same distribution, by design, so we shouldn't catch an anomaly when we test them against each other; however, we should catch anomalies when we test them against other, non-anomaly time points, and that's exactly what we see.

```
ys_true = {}
for i, adjacency in enumerate(networks[1:]):
    y = get_statistic([adjacency, networks[i-1]])
    ys_true[f"{i}:{i+1}"] = float(f"{y:.3f}")
```

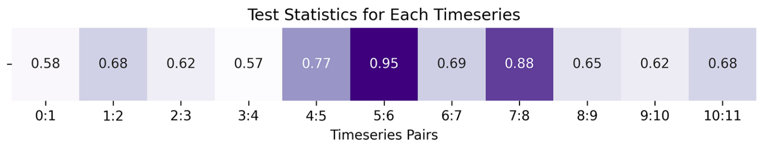


Figure 8.3

Figure 8-3. Test Statistics for Each Timeseries

If we were to plot a distribution of bootstrapped test statistics with each of our estimated y-values, it would look like the histogram below. Notice that two test statistics are clearly anomalous: the one comparing times five and six, and the one comparing times seven and eight. We know by design that networks six and seven actually are anomalous, and so we can see that our test managed to correctly determine the anomaly times.

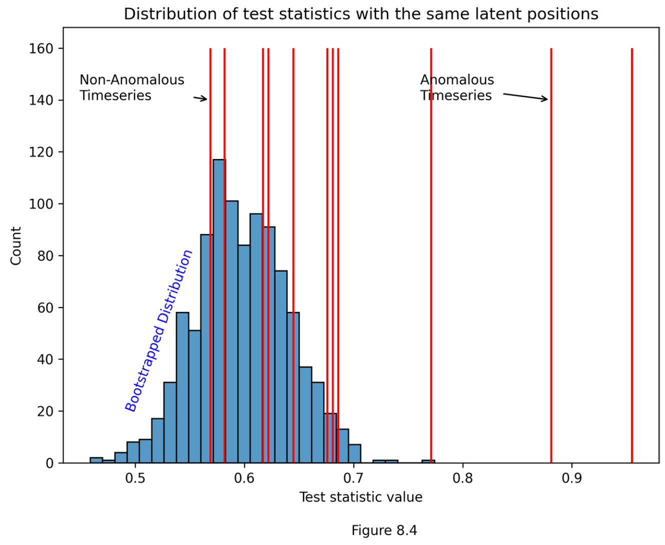


Figure 8-4. Distribution of Test Statistics with the same latent positions

The Distribution of the Bootstrapped Test Statistic

One issue that could pop up is that the bootstrapped test statistic is slightly biased. Since we're generating it from an estimate $\hat{X}$ of the true latent positions X, we'll have a bias of $|\hat{X} - X|$. It's worth comparing the two distributions to determine if that bias is a big deal in practice.

Below you can see the true distribution of the test statistic for the real, unperturbed set of latent positions X we generated the data from (that's the blue distribution). You can also see a distribution of test statistics bootstrapped from a $\hat{X}$. You can see that in this case, they're fairly close.

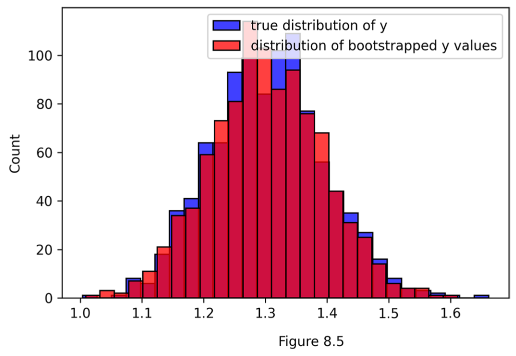


Figure 8-5. Distribution Comparison For the Bootstrapped and True Test Statistic