Title: Random Graphs: Block Partitions and Embeddings

Author: Adam Levav, Chenhongshu Yu

Affiliation: Department of Mathematics & Department of Computer Science

University of Maryland

Date: May 9, 2023

Venue: MATH 420 Final Project Presentation

Introduction

This project involves applying techniques for random graphs model selection and community detection on a specific dataset of 40 cities represented by coordinates and weights. The project involves tasks such as ordering edges by weight, computing the number of cliques, estimating parameters for random graph models (Erdos-Renyi and SSBM), and plotting and analyzing the results. The ultimate goal is to gain insights into the structure and properties of the graph and its communities.

The dataset that we used was a subset of the SGB128 dataset from the Florida State University CITIES city distance data sets [1]. The data contains the following information:

- Coordinates of a set of 40 points (cities) taken from SGB128 dataset;
- Contains 780 weighted edges;
- A symmetric matrix of weights defined by

  $$V(i,j) = \exp(-6 * d(i,j) / \max_{i,j} d(i,j))$$

  for all $i \neq j$ where $d(i,j)$ is the Euclidean distance between city i and city j;
- A weight matrix W obtained by thresholding V to 20 percent of its maximum entry;
- And the adjacency matrix A associated with W: $A(i,j) = 1$ iff $W(i,j) > 0$.

The graph is shown in Figure 1.

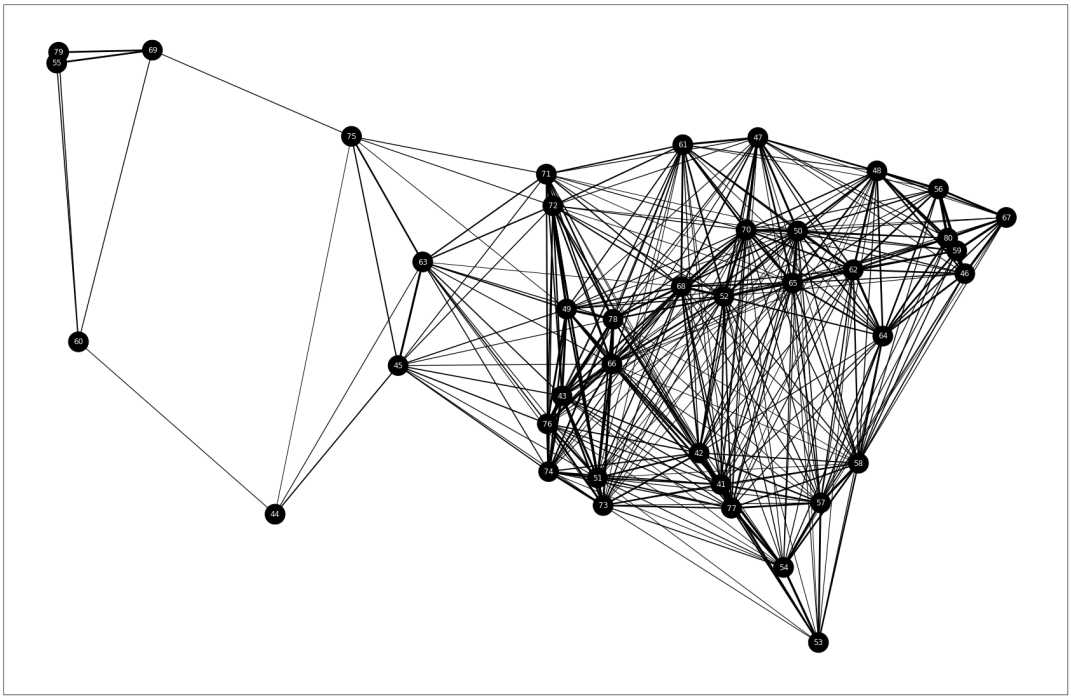


Figure 1: The graph of all 40 cities in the subset, with the weight of the edges indicated by the thickness of the line.

Background / Methods

I. Random Graph Model Testing

i) Goal: To estimate the number of k-cliques under two different models of random graphs.

- Sort edges descending by weights
- For each set of k largest edges, create a graph and compute:
    - Actual number of 3- and 4-cliques
    - Estimated number of 3- and 4-cliques under:
        - Erdos-Renyi (ER) random graph model
        - Symmetric Stochastic Block Model (SSBM)
- Find exponential fit for each model, and then compare.

ii) Calculate the actual number of 3- and 4-cliques:

```
def count_3_4_cliques(G, num_3_4_cliques, new_edge):
    a, b, *data = new_edge
    # 3 cliques
    adj_a = neighbors(G,a)
    adj_b = neighbors(G,b)
    all_c = adj_a & adj_b # intersection of neighbors
    num_3_4_cliques[3] += len(all_c)
    # 4 cliques
    all_d = [v for c in all_c for v in neighbors(G,c)
             if v in all_c]
    num_3_4_cliques[4] += len(all_d)//2

    # return
    G.add_edge(a, b)
    return G, num_3_4_cliques
```

- Appends the new edge to the previous edges and updates the prev_cliques structure accordingly.
- For a new edge (a,b), check for all nodes c such that (a,b) and (a,c) are already in the graph, and append those to Counter.
- Also check for all nodes c and d such that (a,c), (c,d), and (d,b) are in the graph, and append those to the new Counter.

iii) Erdos-Renyi:

The Erdos-Renyi class $\mathcal{G}_{n,p}$ of random graphs is defined as follows:

- Let V denote the set of n vertices, $V = \{1,\ 2,\ \cdots,\ n\}$, and let $\mathcal{G}$ denote the set of all graphs with vertices V. There are exactly $2^{\binom{n}{2}}$ such graphs. The probability mass function on $\boldsymbol{\mathcal{G}}$, $P: \boldsymbol{\mathcal{G}} \rightarrow [0,\ 1]$, is obtained by assuming that, as random variables, edges are independent from one another, and each edge occurs with probability $p \in [0,\ 1]$. Thus a graph $G \in \boldsymbol{\mathcal{G}}$ with m edges will have probability P(G) given by

$$P(G) = p^m(1-p)^{\binom{n}{2}-m}.$$

- Then, given a random graph with n vertices and m edges, the MLE estimator of p is:

$$P_{MLE} = \frac{m}{\binom{n}{2}} = \frac{2m}{n(n-1)}$$

- Based on the calculated MLE, we are able to calculate the estimated number of 3- and 4-cliques on ER-model:

Since $X_q = \sum_{(i_1,\cdots,i_q)\in S_q} 1_{i_1,\cdots,i_q}$ and

$Prob((i_1,\cdots,i_q)\ is\ a\ clique) = p^{\binom{q}{2}}$ we obtain: $E[q-cliques] = \binom{n}{2} p^{\frac{q(q-1)}{2}}$

iv) Symmetric Stochastic Block Model (SSBM):

- The pair $(Z,\ G)$ is drawn under $SSBM(n,k,a,b)$ if Z is an n-dimensional random vector with i.i.d. components uniformly distributed over $[k] = \{1,\ 2,\ \cdots,\ k\}$, and G is an n-vertex graph where distinct vertices i and j are connected with probability a if $Z_i = Z_j$ and probability b if $Z_i \neq Z_j$, independently of other pairs of vertices.
- The agreement between two community vectors $x, y \in [k]^n$ is obtained by maximizing the number of common components of these two vectors over all possible relabelling (i.e., permutations): $Agr(x,y) = \max_{\pi\in S_k} \sum_{i=1}^{n} 1(x_i = \pi(y_i))$ where S_k denotes the group of permutations.
- Use the Maximum Likelihood Estimator (MLE): $(a_{MLE}, b_{MLE}) = argmax_{a,b} Prob(G|Z,a,b)$
- Take the logarithm and obtain: $a_{MLE} = \frac{2(m_{11}+m_{22})}{n_1(n_1-1)+n_2(n_2-1)}$, m_{12}
- Then, we can calculate $a_{MM} = b_{MM} = \frac{1}{2}(c_1 + \sqrt[3]{2c_2 - c_1^3})$ a_{MM} *and* b_{MM}:
- **Algorithm (1)**:
  - Input: n, m ,t
  - Step 1: Compute: $p = \frac{2m}{n(n-1)}, \delta = \frac{6t}{n(n-1)(n-2)} - p^3$
  - Step 2: Test and Constrained Matching compute the Moment estimates: $A_1 = max(0, 2p-1), A_2 = min(1, 2p)$ $P(A_1) = (A_1 - p)^3 - \delta, P(A_2) = (A_2 - p)^3 - \delta$
    - If $P(A_1) \leq 0 \leq P(A_2)$ then $a_{CMM} = p + \delta^{1/3}$, $b_{CMM} = p + \delta^{1/3}$
    - If $P(A_2) < 0$ then $a_{CMM} = A_2$, $b_{CMM} = 2p - A_2$
    - If $P(A_1) > 0$ then $a_{CMM} = A_1$, $b_{CMM} = 2p - A_1$
  - Output: a_{CMM} *and* b_{CMM}

- **Algorithm (2)**:
  - Input: n, m ,t
  - Step 1: Compute:

$$p = \frac{2m}{n(n-1)}, \delta = \frac{6t}{n(n-1)(n-2)} - p^3$$

$$A_1 = max(0, 2p-1), A_2 = min(1, 2p)$$

$$P(A_1) = (A_1 - p)^3 - \delta, P(A_2) = (A_2 - p)^3 - \delta$$

  - Step 2: Test and compute the Constrained Moment Matching estimates:
    - If $P(A_1) \leq 0 \leq P(A_2)$ then $a_{CMM} = p + \delta^{1/3}$, $b_{CMM} = p + \delta^{1/3}$
    - If $P(A_2) < 0$ then $a_{CMM} = A_2$, $b_{CMM} = 2p - A_2$
    - If $P(A_1) > 0$ then $a_{CMM} = A_1$, $b_{CMM} = 2p - A_1$
  - Step 3: Adjust to produce the Modified Constrained Moment Matching estimates:
    - If $0 < a_{CMM}, b_{CMM}$ then $a_{MCMM} = a_{CMM}$, $b_{MCMM} = b_{CMM}$
    - If $b_{MCMM} = 0$ then $a_{MCMM} = 0.99a_{CMM}$, $b_{MCMM} = 0.01b_{CMM}$
    - If $a_{MCMM} = 0$ then $a_{MCMM} = 0.01a_{CMM}$, $b_{MCMM} = 0.99b_{CMM}$
  - Output: a_{MCMM} *and* b_{MCMM}
- $E[X_3]$:

$$\mathbb{E}[X_3] = \frac{n(n-2)}{6}(n-1+\frac{3}{4}(n+1)-\frac{3}{2}n)a^3 + \frac{n(n-2)(\frac{n}{2}-\frac{n+1}{4})}{2}ab^2$$

$$= \frac{n(n-1)(n-2)}{24}a^3 + \frac{n(n-1)(n-2)}{8}ab^2 = \frac{n(n-1)(n-2)}{24}(a^3+3ab^2)$$

- $E[X_4]$:

$$\mathbb{E}[X_4] = \frac{a^0}{24}\left(2\mathbb{E}[n_1^4] - 4n\mathbb{E}[n_1^3] + (6n^2-18n+22)\mathbb{E}[n_1^2]\right.$$

$$\left.+(-4n^3+18n^2-22n)\mathbb{E}[n_1] + n^4 - 6n^3 + 11n^2 - 6n\right) +$$

$$+\frac{a^3b^3}{6}\left(-2\mathbb{E}[n_1^4] + 4n\mathbb{E}[n_1^3] + (-3n^2+3n-4)\mathbb{E}[n_1^2] + (n^3-3n^2+4n)\mathbb{E}[n_1]\right)$$

$$+\frac{a^2b^4}{4}\left(\mathbb{E}[n_1^4] - 2n\mathbb{E}[n_1^3] + (n^2+n-1)\mathbb{E}[n_1^2] + (-n^2+n)\mathbb{E}[n_1]\right)$$

II. Community Detection

i) Goal: partition nodes into two communities using multiple methods

- Apply spectral methods using:
  - Weight matrix (W)
  - Laplacian matrix (Δ)
  - Normalized Laplacian matrix ($\hat{\Delta}$)
- Compute agreement between communities from these methods
  - Actual number of 3- and 4-cliques
  - Estimated number of 3- and 4-cliques under:
    - Erdos-Renyi (ER) random graph model

        - Symmetric Stochastic Block Model (SSBM)
    - Visualize, for each of the three algorithms, and map the two communities using two colors, say red and blue, using the coordinates (X, Y) from the coordinate file assigned to the project. Draw edges according to:
        - Adjacency matrix A
        - Weight matrix W

ii) Spectral Algorithms:

- **Spectral Algorithm using the Weight Matrix (W)**
    - Input: Weight matrix $W \in R^{nxn}$. If the graph is not connected then produce a disjoint partition $(\Omega_1, \Omega_2, \ldots, \Omega_d)$ into connected components. Else:
        - Step 1: Compute the second largest eigenpair of W : (f_2, μ_2), with $Wf_2 = \mu_2 f_2$.
        - Step 2: Define the partition $\Omega_1 = \{k: f_2(k) > 0\}$, $\Omega_2 = \{k: f_2(k) \leq 0\}$. Set d = 2.
    - Output: Disjoint partition $(\Omega_1, \Omega_2, \ldots, \Omega_d)$ of nodes $[k] = \{1, 2, \cdots, k\}$.

- **Spectral Algorithm using the Weighted Graph Laplacian (Δ)**
    - Input: Weight matrix $W \in R^{nxn}$. If the graph is not connected then produce a disjoint partition $(\Omega_1, \Omega_2, \ldots, \Omega_d)$ into connected components. Else:
        - Step 1: Compute the weighted graph Laplacian Δ = D − W , with $D = Diag(W \cdot 1)$.
        - Step 2: Compute the second smallest eigenpair: (e_1, λ_1), with $\Delta e_1 = \lambda_1 e_1$ and $\lambda_1 > 0 = \lambda_0$.
        - Step 3: Define the partition $\Omega_1 = \{k: e_1(k) > 0\}$, $\Omega_2 = \{k: e_1(k) \leq 0\}$. Set d = 2.
    - Output: The disjoint partition $(\Omega_1, \Omega_2, \ldots, \Omega_d)$ of the set of nodes $[n] = \{1, 2, \cdots, n\}$.

- **Spectral Algorithm using the symmetric normalized Weighted Graph Laplacian ($\hat{\Delta}$)**
    - Input: Weight matrix $W \in R^{nxn}$. If the graph is not connected then produce a disjoint partition $(\Omega_1, \Omega_2, \ldots, \Omega_d)$ into connected components. Else:

- Step 1: Compute the symmetric normalized weighted graph Laplacian $\hat{\Delta} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$, with $D = Diag(W \cdot 1)$.
- Step 2: Compute the second smallest eigenpair: (e_1, λ_1), with $\Delta e_1 = \lambda_1 e_1$ and $\lambda_1 > 0 = \lambda_0$.
- Step 3: Define the partition $\Omega_1 = \{k: e_1(k) > 0\}$, $\Omega_2 = \{k: e_1(k) \leq 0\}$. Set d = 2.

○ Output: Disjoint partition $(\Omega_1, \Omega_2, \ldots, \Omega_d)$ of nodes $[n] = \{1, 2, \cdots, n\}$.

iii) Agreement with Partitions:

The agreement between two community vectors $x, y \in [k]^n$ is obtained by maximizing the number of common components of these two vectors over all possible relabelling (i.e., permutations):

$Agr(x, y) = \frac{1}{n} \max_{\pi \in S_k} \sum_{i=1}^{n} 1(x_i = \pi(y_i))$ where S_k denotes the group of permutations.

```
# 2. Compute the agreement matrix between these partitions: The output
# should be a 3 × 3 matrix Agr so that Agr(k, l) represents the partition
# agreement between method k and method l, 1 ≤ k, l ≤ 3, the 3 methods
# above.
from itertools import product

partition_agreement = lambda p1, p2: len(p1 & p2)
agreement = lambda two_partitions: max(partition_agreement(p1,p2) for p1, p2 in product(*two_partitions))

partitions = [comp_weight, comp_lap, comp_norm_lap]
#First_Agr = np.array([[(partition_agreement(p1[0], p2[0])+partition_agreement(p1[1], p2[1]))/40 for p1 in partitions] for p2 in partitions])
#display(First_Agr)
#Agr = np.array([[agreement((p1, p2)) for p1 in partitions] for p2 in partitions])
Agr = np.array([[
            max((partition_agreement(p1[i], p2[j])+partition_agreement(p1[1-i], p2[1-j]))/G.number_of_nodes() for i in [0,1] for j in [0,1])
                    for p1 in partitions] for p2 in partitions])
display(Agr)
```

III. Data Embedding

i) Goal: Find a lower-dimensional representation of the graph using multiple methods

- Apply two embedding algorithms:
  - Laplacian Eigenmap data embedding, target dimension 2
  - LLE dimension reduction after Laplacian Eigenmap data embedding:
    - Laplacian Eigenmap data embedding algorithm, N = 10

- LLE algorithm with non-negativity constraints on previous step (d = 2; K = 4)

ii) Laplacian Eigenmap:

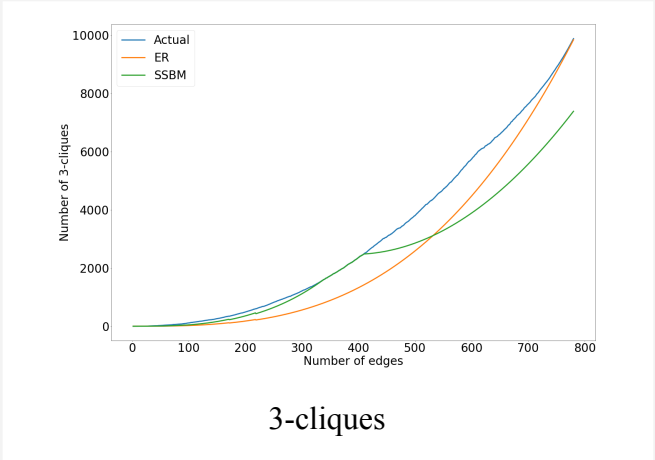
- Step 1: Construct the diagonal matrix $D = Diag(D_{ii})_{1 \le i \le n}$, where $D_{ii} = \sum_{k=1}^{n} W_{i,k}$.
- Step 2: Construct the normalized Laplacian $\hat{\Delta} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$.
- Step 3: Compute the bottom $d + 1$ eigenvectors $e_1, \cdots, e_{d+1}$, $\hat{\Delta} e_k = \lambda_k e_k$, $0 \le \lambda_1 \le \cdots \le \lambda_{d+1}$.
- Step 4: Construct the d × n matrix Y, $Y = [e_2^T \ldots e_{d+1}^T]^{transpose} D^{-\frac{1}{2}}$
- Step 5: The new geometric graph is obtained by converting the columns of Y into n d-dimensional vectors: $[y_1 \ldots y_n] = Y$

iii) Local Linear Embedding (LLE):

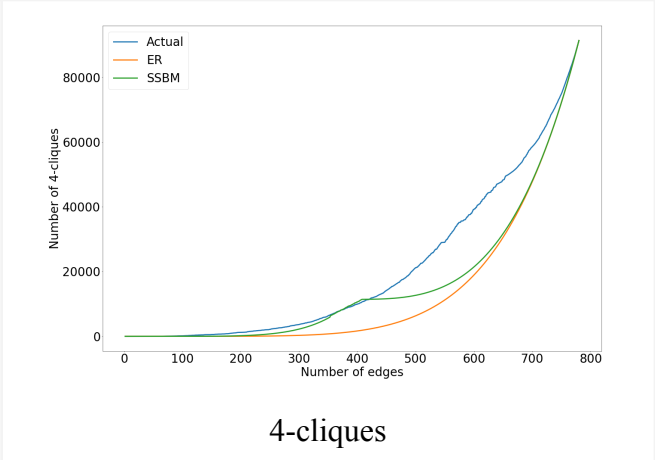
- Part (1): Finding the weight matrix B: For each point i do the following:
  - Find its closest K neighbors, say V_i ; Let r : $V_i \rightarrow \{1, 2, \cdots, K\}$ denotes indexing map.
  - Compute the K × K local covariance matrix C, $C_{r(j),r(k)} = < x_j - x_i, x_k - x_i >$.
  - Solve for u, minimize $u^T C u$, subject to u ≥ 0 , $u^T \cdot 1 = 1$ where 1 denotes the K-vector of 1's.
  - Set $B_{i,j} = u_{r(j)}$ for $j \in V_i$.
- Part (2): Solving the Eigen Problem:
  - Create the (typically sparse) matrix $L = (I - B)^T (I - B)$
  - Find the bottom $d + 1$ eigenvectors of L (the bottom eigenvector would be $[1, \cdots, 1]^T$ associated to eigenvalue 0) $\{e_1, e_2, \cdots, e_{d+1}\}$.
  - Discard the last vector (the constant eigenvector) and insert all other eigenvectors as rows into matrix Y, $Y = [e_2^T, \cdots, e_{d+1}^T]^T$

Results

I. Random Graph Model Testing



3-cliques

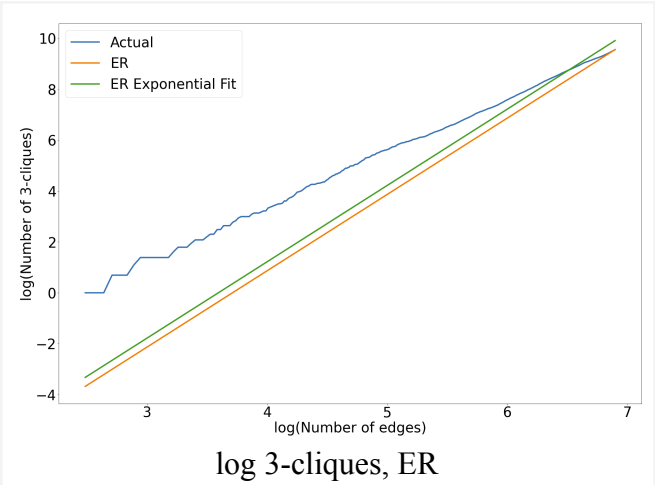


4-cliques

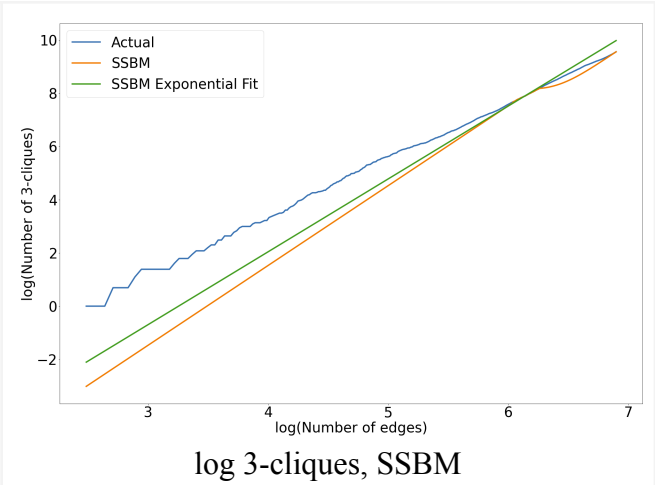
Figure 2: The actual and estimated number of 3-cliques and 4-cliques under the Erdos-Renyi (ER) and Symmetric Stochastic Block Matrix (SSBM) graph models.

# cliques	ER	SSBM
3	$\underline{6.14 \cdot 10^5}$	$7.88 \cdot 10^8$
4	$1.04 \cdot 10^8$	$\underline{6.49 \cdot 10^7}$

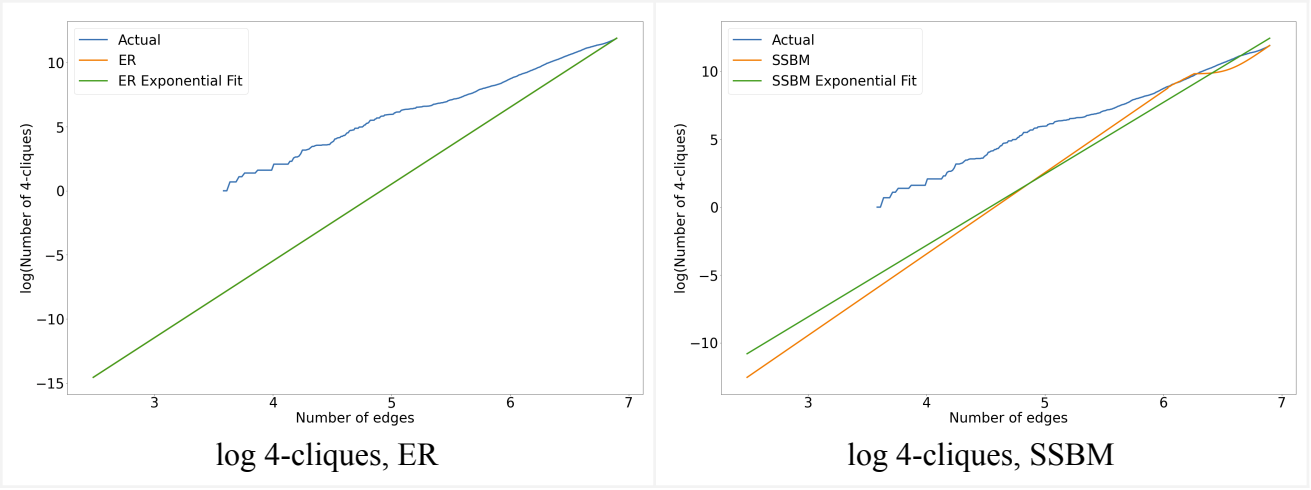
Table 1: Mean squared error of ER and SSBM *k*-clique estimates



log 3-cliques, ER



log 3-cliques, SSBM



log 4-cliques, ER

log 4-cliques, SSBM

Figure 3:

***k*-cliques**	**Approximation**	***C***	***r***
3-cliques	ER	0.000067	2.807609
	SSBM	0.000676	2.461062
4-cliques	ER	1.167037e-12	5.829317
	SSBM	2.405197e-08	4.356753

Table 2: Exponential fits for each curve (best fit to $y = Ce^r$)

II. Community Detection

Algorithm	**Partition 1**	**Partition 2**
Weight	{0, 1, 2, 3, 4, 5, 7, 8, 9, 11, 17, 18, 21, 26, 27, 28, 30, 31, 32, 34, 37, 38, 39}	{33, 35, 36, 6, 10, 12, 13, 14, 15, 16, 19, 20, 22, 23, 24, 25, 29}
Laplacian	{32, 33, 2, 3, 4, 34, 37, 39, 9, 10, 11, 12, 16, 17, 18, 24, 25, 28}	{0, 1, 5, 6, 7, 8, 13, 14, 15, 19, 20, 21, 22, 23, 26, 27, 29, 30, 31, 35, 36, 38}
Normalized Laplacian	{0, 2, 34, 36, 5, 37, 7, 38, 11, 14, 15, 16, 17, 19, 20, 21, 23, 30}	{1, 3, 4, 6, 8, 9, 10, 12, 13, 18, 22, 24, 25, 26, 27, 28, 29, 31, 32, 33, 35, 39}

Table 3: Calculated partitions from each spectral algorithm.

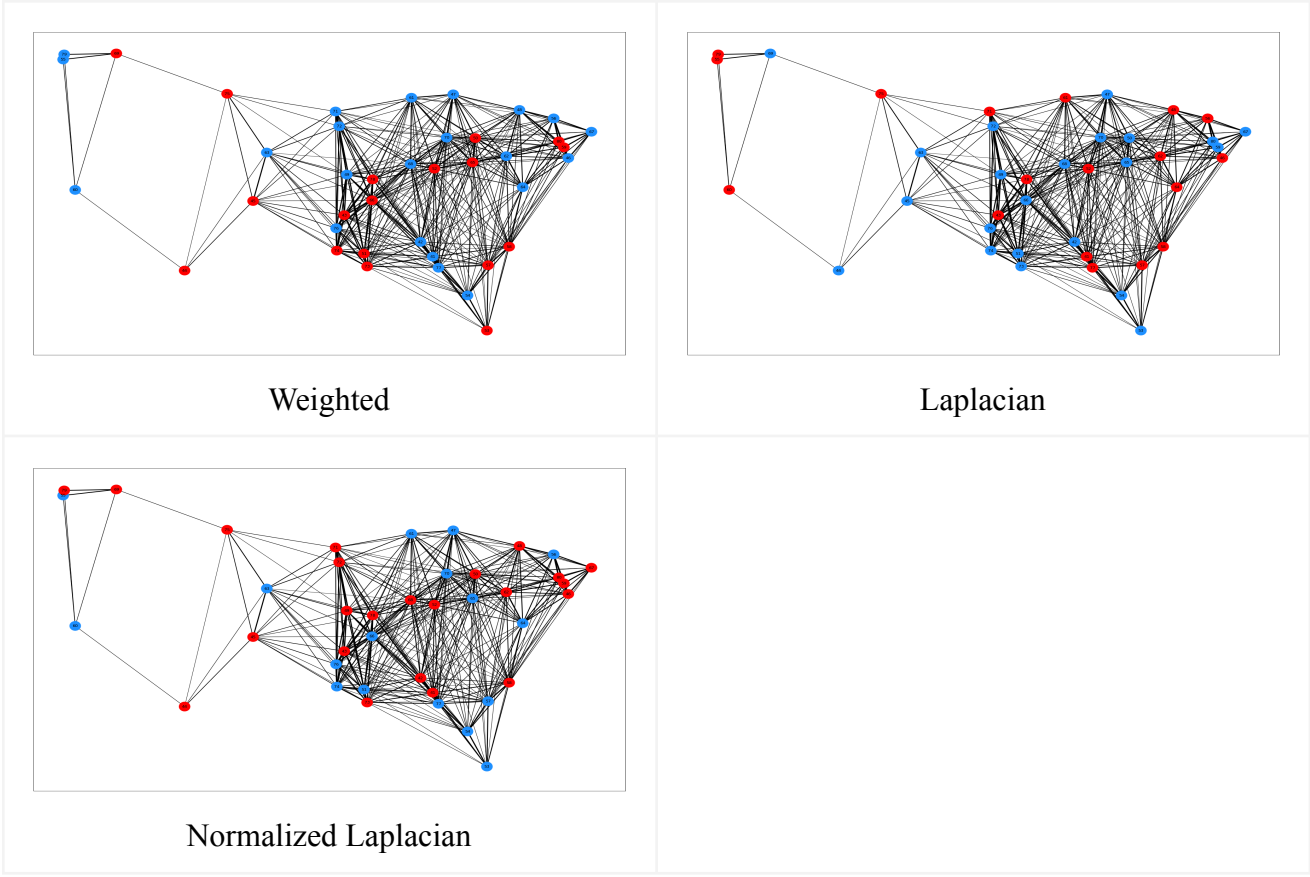


Figure 4: All partition graphs with weighted edges. (The versions of the graphs with the unweighted edges are excluded from this report to avoid redundancy.)

	Weight	**Laplacian**	**Norm. Laplacian**
Weight	100%	57.5%	52.5%
Laplacian	57.5%	100%	60%
Norm. Laplacian	52.5%	60%	100%

Table 4: Agreement Matrix

III. Data Embedding

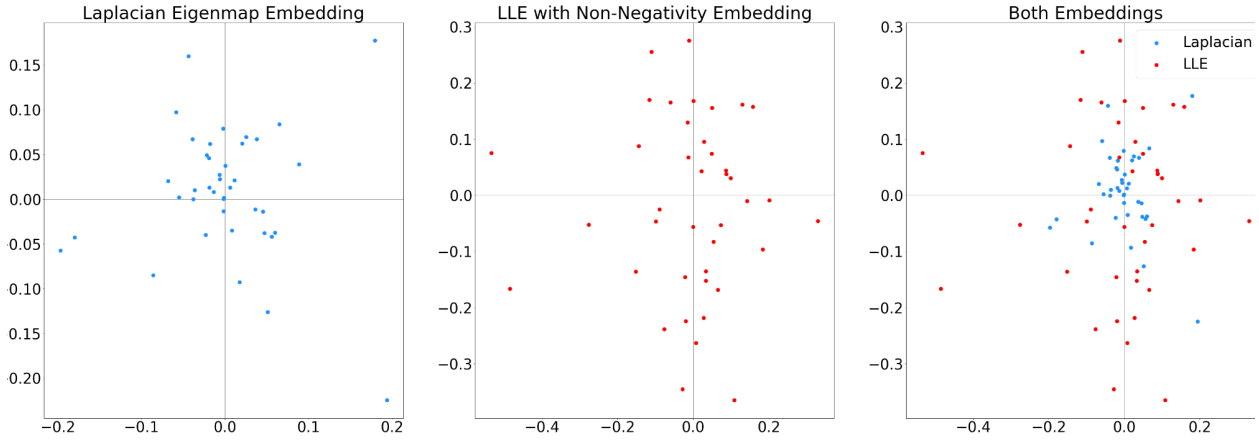


Figure 5: All three embeddings.

Discussion

I. Random Graph Model Testing

Our first challenge to overcome was the calculation of the 4-cliques; using the most naive brute force approach would have required the code to run for over one day to complete. To resolve this, we developed an iterative algorithm which, at each step, adds a new edge to the previous sub-graph and determines the number of new 3- and 4-cliques which are newly induced. This algorithm reduced our runtime dramatically, down to less than 3 seconds.

The primary challenge with this section was that there was an unexpected discrepancy between the number of 3 and 4 cliques as calculated by the SSBM model and the actual number of cliques. As can be seen in the graph below, there is only a small window (between approximately 328 and 410 edges) where neither *a* nor *b* are 'clamped' at 0 or 1. In this section, the SSBM model fits the 3- and 4-clique numbers very well; however, outside of this region the SSBM estimates dramatically diverge from the actual counts.

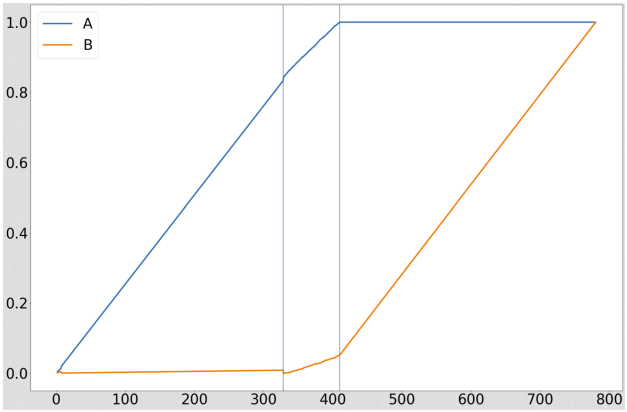


Figure 6: The plots of a_{CMM} and b_{CMM} (the estimated parameters for SSBM model) over the number of edges included.

We double checked the calculations and cross-referenced our results with those of Jiatong Liang, Hudson Hinshaw and James Boggs, and got consistently poor results throughout. It may be that the graph data is biased such that it only approximates an SSBM graph in specific conditions; however, since other groups claimed to get a far better fit with their given datasets, it is likely there is some error in the calculation.

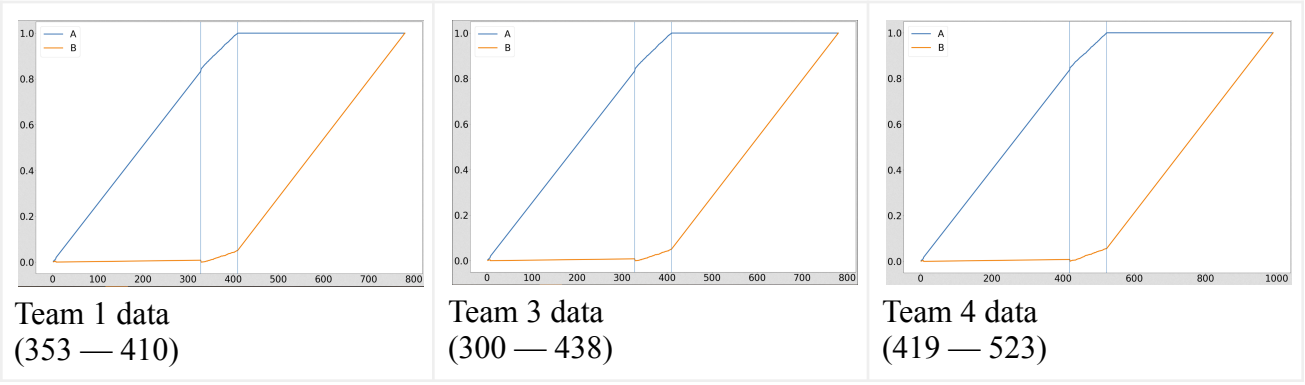


Figure 7: *a* and *b* estimates for each of the other team's data, along with the 'goldi-locks zone' where the estimates are closely aligned with the actual values.

From the graphs, it is clear that in a majority of situations both approximations underestimate the number of cliques, except in the case of SSBM's 4-cliques, where the estimate briefly over-estimates before dropping back down. It is unclear if this is indicative of some relevant information for this data set (e.g. that there are more connections than expected for this number of cities); more analysis is needed to investigate this further.[1]

If we take these results at face value, it is not clear which algorithm is better for this data set. The MSE of the 3-clique of ER is *lower* compared to SSBM, but the opposite is the case for the 4-cliques, where SSBM narrowly beats out ER. When visually inspecting the fits, the SSBM graph is the clear winner (especially in the so-called 'goldilocks zone'), so in this case it appears that SSBM is a more effective and accurate approximation for this data set.

One thing to note is that for the exponential fits, we first had to exclude the first *n* entries for each; for 3-cliques, *n* was 10, and for 4-cliques *n* was 100. This is most likely because among the first few sets of edges, there are unlikely to be any 3-cliques or 4-cliques. Therefore, the first entries will provide non-useful noise in our fits; especially considering the fact that this fit involves applying a logarithm and log(0) is undefined, it is clear that those values should not be included in our fits.[2]

[1] One other thing of interest to note is that the exponential approximation perfectly matches with the "actual" estimate of number of 3- and 4-cliques for ER, as the calculation for the ER clique counts is equivalent to an exponential (can be expressed in exponential form).

[2] There are other justifications for this practice, which will not be described here.

II. Community Detection

The community detention results are unfortunately unilluminating. The agreement between the partitions is quite low, barely above 50 percent (i.e. the amount that would be expected by random chance) between any of the partitions. The largest agreement (60 percent) was between the Laplacian and Normalized Laplacian spectral algorithms. There are also no clear patterns in the locations and connections of the different partitions when drawing the graphs.

III. Data Embedding

The embeddings reduced the dimension of the graphs dramatically, but do not appear to have much structure within the embeddings (e.g. there are no clear clusters or communities). This matches with the results from the previous section showing low visual coherence of the partitions. We can also see from the graph below that the embeddings have means very close to zero, which is what would be expected from these algorithms.

	Mean X	**Mean Y**
Laplacian Eigenmap	0.0004267	0.009480
LLE	-0.007152	-0.0208631

Table 6. Means of each of the embedding algorithms

In an attempt to verify the results, we used the community detection algorithms from Part II to color the points in the embeddings. We would expect that the colors would show distinct patterns of groups in the data; in particular, we expect that when mapped to the Laplacian Eigenmap embedding, the communities derived from the Laplacian-based spectral algorithms would show a clean divide across the y-axis. However, this is not what we observed; the given groups were spread throughout the embeddings without apparent rhyme or reason.

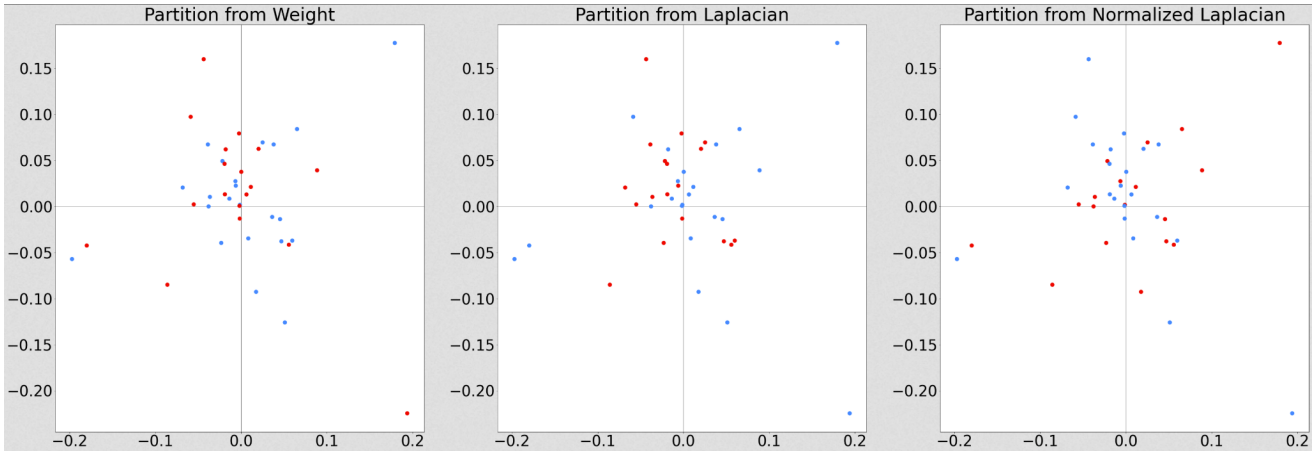


Figure 7: Using the three partitions from Part II to color the Laplacian embedding from Part III.

These results indicate that there is an issue with some components of our analysis pipeline; the most likely candidate is the embedding algorithms due to their high complexity and numerous opportunities for error. Future work must be done to isolate the specific cause of these problematic results and resolve any ambiguities or other programmatic errors.

References

- Balan, R. (n.d.). Lectures for MATH 420. https://www.math.umd.edu/~rvbalan/TEACHING/MATH420Spring2023/LECTURES/index.html
- *Cities city distance datasets*. CITIES - City Distance Datasets. (n.d.). https://people.sc.fsu.edu/~jburkardt/datasets/cities/cities.html

Appendix

1. Part One: Random Graph Model Testing

```python
from itertools import chain
def count_3_4_cliques(G, num_3_4_cliques, new_edge):
    """Appends the new edge to the previous edges and updates the prev_cliques structure accordingly.
    For a new edge (a,b), check for all nodes c such that (a,b) and (a,c) are already in the graph,
    and append those to the new Counter;
    Also check for all nodes c and d such that (a,c), (c,d), and (d,b) are in the graph,
    and append those to the new Counter."""
    a, b, *data = new_edge
    #print(a,b)
    # 3 cliques
    adj_a = neighbors(G,a)
    adj_b = neighbors(G,b)

    all_c = adj_a & adj_b # intersection of neighbors of a and b
    num_3_4_cliques[3] += len(all_c)
    # 4 cliques
    """
    possible_d = set().union(*[neighbors(G,c) for c in all_c])
    all_d = possible_d & all_c
    print(all_c, "&", possible_d, "->", all_d)
    """
    all_d = [v for c in all_c for v in neighbors(G,c) if v in all_c]
    #print(all_d)

    num_3_4_cliques[4] += len(all_d)//2
    # return
    G.add_edge(a, b)
    return G, num_3_4_cliques
```

Python

```python
erdos_reyni_p_MLE = lambda num_edges, num_nodes: \
    (2 * num_edges) / (num_nodes * (num_nodes - 1))
```

Python

```python
from math import comb
erdos_reyni_estimated_q_cliques = lambda q, num_nodes, p: \
    comb(num_nodes, q) * (p ** (q * (q - 1) / 2))
```

Python

```python
def ssbm_a_b_CMM(num_nodes, num_edges, num_3_cliques, output_P=False):
    """Return the expected values for a and b for the SSBM with 2-communities SSBM(n,2,a,b)
    by Modified Constrained Moment Matching Algorithm 2."""
    n, m, t = num_nodes, num_edges, num_3_cliques
    p = 2*m/(n*(n-1))
    delta = 6*t/(n*(n-1)*(n-2)) - p**3
    A1, A2 = max(0,2*p-1), min(1,2*p)
    P = lambda A:(A-p)**3 - delta
    PA1, PA2 = P(A1), P(A2)

    if PA1 <= 0 <= PA2: a_CMM, b_CMM = p + delta**(1/3), p - delta**(1/3)
    elif PA2 < 0:       a_CMM, b_CMM = A2, 2*p-A2
    else:               a_CMM, b_CMM = A1, 2*p-A1

    # Adjust to produce the Modified Constrained Moment Matching estimates
    if b_CMM == 0:   a_MCMM, b_MCMM = 0.99*a_CMM, 0.01*a_CMM
    elif a_CMM == 0: a_MCMM, b_MCMM = 0.01*b_CMM, 0.99*b_CMM
    else:            a_MCMM, b_MCMM = a_CMM, b_CMM

    if output_P:     return PA1, PA2, a_MCMM, b_MCMM
    else:            return a_MCMM, b_MCMM
```

Python

```python
def ssbm_estimated_q_cliques(q, num_nodes, a, b):
    n = num_nodes
    if q == 3:
        expected_3_cliques = (
            (n*(n-1)*(n-2)/24) * (a**3 + 3*a*b**2)
        )
        return expected_3_cliques
    elif q == 4:
        E_n1 = n / 2
        E_n1_square = (n ** 2 + n) / 4
        E_n1_cube = (n ** 2) * (n + 3) / 8
        E_n1_quad = n * (n + 1) * (n ** 2 + 5 * n - 2) / 16
        expected_4_cliques = (
            (a ** 6) / 24 *
            (
                2 * E_n1_quad -
                4 * n * E_n1_cube +
                (6 * (n ** 2) - 18 * n + 22) * E_n1_square +
                (-4 * (n ** 3) + 18 * (n ** 2) - 22 * n) * E_n1 +
                (n ** 4) - 6 * (n ** 3) + 11 * (n ** 2) - 6 * n
            ) +
            (a ** 3) * (b ** 3) / 6 *
            (
                -2 * E_n1_quad +
                4 * n * E_n1_cube +
                (-3 * (n ** 2) + 3 * n - 4) * E_n1_square +
                ((n ** 3) - 3 * (n ** 2) + 4 * n) * E_n1
            ) +
            (a ** 2) * (b ** 4) / 4 *
            (
                E_n1_quad -
                2 * n * E_n1_cube +
                ((n ** 2) + n - 1) * E_n1_square +
                (-(n ** 2) + n) * E_n1
            )
        )
        return expected_4_cliques
    else:
        raise Exception("q must be 3 or 4.")
```

Python

2. Community Detection

```python
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
```

Python

```python
def spectral_algorithm_weight(G):
    weight = nx.adjacency_matrix(G).todense()
    # (1) Compute the second largest eigenpair of A: (f2, μ2), with Af2 = μ2f1.
    eigenvalues, eigenvectors = np.linalg.eig(weight)
    sorted_eigenvectors = eigenvectors[eigenvalues.argsort()]
    f2 = sorted_eigenvectors[-2]
    # (2) Define the partition Ω1 = {k : f2(k) > 0}, Ω2 = {k : f2(k) ≤ 0}. Set
    # d = 2.
    omega1 = {k for k, g in zip(G.nodes,f2) if g > 0}
    omega2 = {k for k, g in zip(G.nodes,f2) if g <= 0}
    return omega1, omega2
```

Python

```python
def spectral_algorithm_laplacian(G):
    # (1) Compute the graph Laplacian Δ = D − A, with D = Diag(A · 1), the
    # degree matrix.
    laplacian = nx.laplacian_matrix(G).todense()
    # (2) Compute the second smallest eigenpair: (e1, λ1), with Δe1 = λ1e1
    # and λ1 > 0 = λ0.
    eigenvalues, eigenvectors = np.linalg.eig(laplacian)
    sorted_eigenvectors = eigenvectors[eigenvalues.argsort()]
    e1 = sorted_eigenvectors[1]
    # (3) Define the partition Ω1 = {k : e1(k) > 0}, Ω2 = {k : e1(k) ≤ 0}. Set
    # d = 2.
    omega1 = {k for k, g in zip(G.nodes,e1) if g > 0}
    omega2 = {k for k, g in zip(G.nodes,e1) if g <= 0}
    return omega1, omega2
```

Python

```python
def spectral_algorithm_normalized_laplacian(G):
    if not nx.is_connected(G):
        return [component.nodes for component in nx.connected_components(G)]
    else:
        # (1) Compute the symmetric normalized graph Laplacian
        # Δ = I − D−1/2AD−1/2, with D = Diag(A · 1) the degree matrix.
        normalized_laplacian = nx.normalized_laplacian_matrix(G).todense()
        # TODO: Check if there are
        # (2) Compute the second smallest eigenpair: (e1, λ1), with ~Δe1 = λ1e1
        # and λ1 > 0 = λ0.
        eigenvalues, eigenvectors = np.linalg.eig(normalized_laplacian)
        sorted_eigenvectors = eigenvectors[eigenvalues.argsort()]
        e1 = sorted_eigenvectors[1]
        # (3) Define the partition Ω1 = {k : e1(k) > 0}, Ω2 = {k : e1(k) ≤ 0}. Set
        # d = 2
        omega1 = {k for k, g in zip(G.nodes,e1) if g > 0}
        omega2 = {k for k, g in zip(G.nodes,e1) if g <= 0}
        return omega1, omega2
```

Python

3. Data Embedding

```python
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
```

Python

```python
from scipy.linalg import fractional_matrix_power

# Laplacian Eigenmap data embedding for target dimension d = 2
def laplacian_eigenmap(G, d):
    # Input: Weight matrix W , target dimension d
    weight = nx.adjacency_matrix(G).todense()
    # (1) Construct the diagonal matrix D = diag(Dii )1≤i≤n, where
    # Dii = ∑_{k=1}^N Wi,k
    diagonal = np.diag(np.sum(weight,axis=1))
    # (2) Construct the normalized Laplacian ~Δ = I − D−1/2WD−1/2.
    normalized_laplacian = nx.normalized_laplacian_matrix(G).todense()
    # (3) Compute the bottom d + 1 eigenvectors e1, · · · , ed+1, ~Δek = λk ek ,
    # 0 = λ1 ≤ · · · ≤ λd+1.
    eigenvalues, eigenvectors = np.linalg.eig(weight)
    sorted_eigenvectors = eigenvectors[eigenvalues.argsort()]
    bottom_eigenvectors = sorted_eigenvectors[1:d+1]
    # (4) Construct the d × n matrix Y:
    # Y = [e2^T, ..., ed+1^T]^T * D−1/2
    Y = bottom_eigenvectors @ fractional_matrix_power(diagonal,-1/2)
    # (5) The new geometric graph is obtained by converting the columns of Y
    # into n d−dimensional vectors:
    # [y1 | · · · | yn] = Y
    return Y
```

Python

```python
from sklearn.neighbors import NearestNeighbors
from scipy.optimize import minimize

def dimension_reduction_lle_non_negativity_constraints(X, K, d):
    # Input: A geometric graph {x1, x2, · · · , xn} ⊂ RN . Parameters:
    # neighborhood size K and dimension d.
    n = X.shape[0]

    # (1) Finding the weight matrix B:
    B = np.zeros((n, n))

    # (Precomputing nearest neighbors for each point)
    nn = NearestNeighbors(n_neighbors=K+1)
    nn.fit(X)
    _, indices = nn.kneighbors(X)
    V = indices[:, 1:] # the closest neighbor is itself, so remove it
    #print(indices)

    # For each point i do the following:
    for i in range(n):
        # (1) Find its closest K neighbors, say Vi;
        # Let r : Vi → {1, 2, · · · , K } denote an indexing map;
        closest_k_neighbors = X[V[i]]
        # (2) Compute the K × K local covariance matrix C,
        # Cr(j),r(k) =  (xj − xi , xk − xi.
        closest_k_neighbors_diffs = closest_k_neighbors - X[i]
        C = np.cov(closest_k_neighbors_diffs)
        # (3) Solve for u, minimize u^T C u subject to u ≥ 0 , u^T · 1 = 1
        # where 1 denotes the K −vector of 1's.
        objective = lambda u: u.dot(C.dot(u))
        constraints = (
            {'type': 'eq', 'fun': lambda u: u.dot(np.ones(K)) - 1},
            {'type': 'ineq', 'fun': lambda u: u})
        u0 = np.ones(K)
        result = minimize(objective, u0, method='SLSQP', constraints=constraints)
        u = result.x
        # (4) Set Bi,j = ur(j) for j ∈ Vi.
        B[i, V[i]] = u
    assert B.shape == (n, n)
    #print(B)
```

```python
    # (2) Solving the Eigen Problem:
    # (1) Create the (typically sparse) matrix L = (I − B)^T (I − B);
    L = (np.eye(n) - B).T @ (np.eye(n) - B)
    #print(L)
    # (2) Find the bottom d + 1 eigenvectors of L (the bottom eigenvector
    # would be [1, · · · , 1]^T associated to eigenvalue 0) {e1, e2, · · · , ed+1};
    eigenvalues, eigenvectors = np.linalg.eig(L)
    sorted_eigenvectors = eigenvectors[eigenvalues.argsort()]
    # (3) Discard the last vector (the constant eigenvector) and insert all other
    # eigenvectors as rows into matrix Y:
    # Y = [e2^T, ..., ed+1^T]^T ∗ D−1/2
    Y = sorted_eigenvectors[1:d+1]
    # Output: {y1, · · · , yn} ⊂ Rd as columns from
    # [ y1 | · · · | yn ] = Y
    assert Y.shape == (d, n)
    return Y
```

Python