Graph Encoder Embedding

Cencheng Shen

University of Delaware

Collaborators: Qizhe Wang, Xihan Qin, Carey E. Priebe, Youngser Park, Jonathan Larsson, Ha Trinh.

Acknowledgment: NSF DMS, NSF TRIPODS, DARPA D3M, Microsoft Research.

Manuscript: `https://arxiv.org/abs/2109.13098`

Code: `https://github.com/cshen6/GraphEmd`

Overview

Section 1

Introduction

Introduction

Graph data has gained many recent interests. It arises naturally in modern data collection and captures interactions among objects.

Introduction

Graph data has gained many recent interests. It arises naturally in modern data collection and captures interactions among objects.

Given n vertices and s edges, a graph can be represented by an $n \times n$ adjacency matrix $\mathbf{A}$ where $\mathbf{A}(i, j)$ is the edge weight between ith vertex and jth vertex.

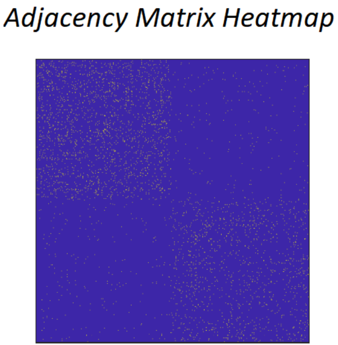
Introduction

Graph data has gained many recent interests. It arises naturally in modern data collection and captures interactions among objects.

Given n vertices and s edges, a graph can be represented by an $n \times n$ adjacency matrix $\mathbf{A}$ where $\mathbf{A}(i, j)$ is the edge weight between ith vertex and jth vertex.

In practice, a graph is typically stored by an $s \times 3$ edgelist $\mathbf{E}$, where the first two columns store the vertex indices of each edge and the last column is the edge weight.

Example



Graph Embedding

To utilize graph data for subsequent inference, graph embedding is arguably the most popular approach, which learns a low-dimensional Euclidean representation of each vertex.

Graph Embedding

To utilize graph data for subsequent inference, graph embedding is arguably the most popular approach, which learns a low-dimensional Euclidean representation of each vertex.

Some popular methods include:

- Adjacency / Laplacian Spectral Embedding : applying SVD to the adjacency or Laplacian matrix.

Graph Embedding

To utilize graph data for subsequent inference, graph embedding is arguably the most popular approach, which learns a low-dimensional Euclidean representation of each vertex.

Some popular methods include:

- Adjacency / Laplacian Spectral Embedding : applying SVD to the adjacency or Laplacian matrix.
- Deepwalk / Node2vec : random-walk based method per vertex.

Graph Embedding

To utilize graph data for subsequent inference, graph embedding is arguably the most popular approach, which learns a low-dimensional Euclidean representation of each vertex.

Some popular methods include:

- Adjacency / Laplacian Spectral Embedding : applying SVD to the adjacency or Laplacian matrix.
- Deepwalk / Node2vec : random-walk based method per vertex.
- Graph Convolutional Network : based on adjacency matrix and gradient descent.

Applications

Community Detection: A graph data has community structure if the vertices can be grouped into different classes based on the edge connectivity.

Applications

Community Detection: A graph data has community structure if the vertices can be grouped into different classes based on the edge connectivity.

In case of supervised learning, some vertices come with ground-truth labels and serve as the training data; while in case of unsupervised learning, the graph data has no known label.

Applications

Community Detection: A graph data has community structure if the vertices can be grouped into different classes based on the edge connectivity.

In case of supervised learning, some vertices come with ground-truth labels and serve as the training data; while in case of unsupervised learning, the graph data has no known label.

And many more graph-based questions: hypothesis testing, signal subgraph, outlier detection in graph time-series, hierarchical community detection, etc.

Motivations

Modern graph data often come with very large number of vertices and edges with sparse structure. Existing graph methods often suffer from some of those limitations:

Motivations

Modern graph data often come with very large number of vertices and edges with sparse structure. Existing graph methods often suffer from some of those limitations:

- Many tuning parameters to choose from or cross-validate;

Motivations

Modern graph data often come with very large number of vertices and edges with sparse structure. Existing graph methods often suffer from some of those limitations:

- Many tuning parameters to choose from or cross-validate;
- Computationally expensive in running time and memory;

Motivations

Modern graph data often come with very large number of vertices and edges with sparse structure. Existing graph methods often suffer from some of those limitations:

- Many tuning parameters to choose from or cross-validate;
- Computationally expensive in running time and memory;
- Complicated algorithm dependency;

Motivations

Modern graph data often come with very large number of vertices and edges with sparse structure. Existing graph methods often suffer from some of those limitations:

- Many tuning parameters to choose from or cross-validate;
- Computationally expensive in running time and memory;
- Complicated algorithm dependency;
- Lack of theoretical understanding, or

Motivations

Modern graph data often come with very large number of vertices and edges with sparse structure. Existing graph methods often suffer from some of those limitations:

- Many tuning parameters to choose from or cross-validate;
- Computationally expensive in running time and memory;
- Complicated algorithm dependency;
- Lack of theoretical understanding, or
- Suboptimal empirical performance;

Motivations

Modern graph data often come with very large number of vertices and edges with sparse structure. Existing graph methods often suffer from some of those limitations:

- Many tuning parameters to choose from or cross-validate;
- Computationally expensive in running time and memory;
- Complicated algorithm dependency;
- Lack of theoretical understanding, or
- Suboptimal empirical performance;

Motivations

Modern graph data often come with very large number of vertices and edges with sparse structure. Existing graph methods often suffer from some of those limitations:

- Many tuning parameters to choose from or cross-validate;
- Computationally expensive in running time and memory;
- Complicated algorithm dependency;
- Lack of theoretical understanding, or
- Suboptimal empirical performance;

We would like a scalable embedding method that is easy to implement, theoretically sound, numerically superior, and capable of processing billions of edges in minutes!

Section 2

Graph Encoder Embedding

Matrix Version with Partial Labels

- **Input**: An adjacency matrix $\mathbf{A}$, and a label vector $\mathbf{Y}$ of K classes (unknown labels are set to 0).

Matrix Version with Partial Labels

- **Input**: An adjacency matrix $\mathbf{A}$, and a label vector $\mathbf{Y}$ of K classes (unknown labels are set to 0).
- **Step 1**: Compute the number of observations per-class n_k.

Matrix Version with Partial Labels

- **Input**: An adjacency matrix $\mathbf{A}$, and a label vector $\mathbf{Y}$ of K classes (unknown labels are set to 0).
- **Step 1**: Compute the number of observations per-class n_k.
- **Step 2**: For the given label vector, compute the one-hot encoding matrix $\mathbf{W} \in \mathbb{R}^{n \times K}$.

Matrix Version with Partial Labels

- **Input**: An adjacency matrix $\mathbf{A}$, and a label vector $\mathbf{Y}$ of K classes (unknown labels are set to 0).
- **Step 1**: Compute the number of observations per-class n_k.
- **Step 2**: For the given label vector, compute the one-hot encoding matrix $\mathbf{W} \in \mathbb{R}^{n \times K}$.
- **Step 3**: For each $k \in [1, \ldots, K]$, do

$$\mathbf{W}[\mathbf{Y} = k, k] = \mathbf{W}[\mathbf{Y} = k, k]/n_k.$$

Matrix Version with Partial Labels

- **Input**: An adjacency matrix $\mathbf{A}$, and a label vector $\mathbf{Y}$ of K classes (unknown labels are set to 0).
- **Step 1**: Compute the number of observations per-class n_k.
- **Step 2**: For the given label vector, compute the one-hot encoding matrix $\mathbf{W} \in \mathbb{R}^{n \times K}$.
- **Step 3**: For each $k \in [1, \ldots, K]$, do

$$\mathbf{W}[\mathbf{Y} = k, k] = \mathbf{W}[\mathbf{Y} = k, k]/n_k.$$

- **Step 4**: $\mathbf{Z} = \mathbf{AW}$.

Matrix Version with Partial Labels

- **Input**: An adjacency matrix $\mathbf{A}$, and a label vector $\mathbf{Y}$ of K classes (unknown labels are set to 0).
- **Step 1**: Compute the number of observations per-class n_k.
- **Step 2**: For the given label vector, compute the one-hot encoding matrix $\mathbf{W} \in \mathbb{R}^{n \times K}$.
- **Step 3**: For each $k \in [1, \ldots, K]$, do

$$\mathbf{W}[\mathbf{Y} = k, k] = \mathbf{W}[\mathbf{Y} = k, k]/n_k.$$

- **Step 4**: $\mathbf{Z} = \mathbf{AW}$.
- **Output**: The final embedding $\mathbf{Z} \in \mathbb{R}^{n \times K}$.

Matrix Version with Partial Labels

- **Input**: An adjacency matrix $\mathbf{A}$, and a label vector $\mathbf{Y}$ of K classes (unknown labels are set to 0).
- **Step 1**: Compute the number of observations per-class n_k.
- **Step 2**: For the given label vector, compute the one-hot encoding matrix $\mathbf{W} \in \mathbb{R}^{n \times K}$.
- **Step 3**: For each $k \in [1, \ldots, K]$, do

$$\mathbf{W}[\mathbf{Y} = k, k] = \mathbf{W}[\mathbf{Y} = k, k]/n_k.$$

- **Step 4**: $\mathbf{Z} = \mathbf{AW}$.
- **Output**: The final embedding $\mathbf{Z} \in \mathbb{R}^{n \times K}$.

Matrix Version with Partial Labels

- **Input**: An adjacency matrix $\mathbf{A}$, and a label vector $\mathbf{Y}$ of K classes (unknown labels are set to 0).
- **Step 1**: Compute the number of observations per-class n_k.
- **Step 2**: For the given label vector, compute the one-hot encoding matrix $\mathbf{W} \in \mathbb{R}^{n \times K}$.
- **Step 3**: For each $k \in [1, \ldots, K]$, do

$$\mathbf{W}[\mathbf{Y} = k, k] = \mathbf{W}[\mathbf{Y} = k, k]/n_k.$$

- **Step 4**: $\mathbf{Z} = \mathbf{AW}$.
- **Output**: The final embedding $\mathbf{Z} \in \mathbb{R}^{n \times K}$.

The ith row of $\mathbf{Z}$ represents the vertex embedding for vertex i, while the kth column represents the connectivity to class k. One may further normalize each row into norm 1.

Example

```
>> A

A =

     0     1     1     0     0
     1     0     1     0     0
     1     1     0     1     0
     0     0     1     0     1
     0     0     0     1     0

>> Y

Y =

     1
     1
     1
     2
     2
```

```
>> W=onehotencode(categorical(Y),2)

W =

     1     0
     1     0
     1     0
     0     1
     0     1

>> W(Y==1,1)=W(Y==1,1)/sum(Y==1)

W =

    0.3333         0
    0.3333         0
    0.3333         0
         0    1.0000
         0    1.0000
```

```
>> W(Y==2,2)=W(Y==2,2)/sum(Y==2)

W =

    0.3333         0
    0.3333         0
    0.3333         0
         0    0.5000
         0    0.5000

>> Z=A*W

Z =

    0.6667         0
    0.6667         0
    0.6667    0.5000
    0.3333    0.5000
         0    0.5000
```

The final embedding **Z** exhibits clear separation of community structure.

Edgelist Version

Instead of operating on adjacency matrix, it can be implemented on edgelist directly. All it took is to iterate the edgelist just twice!

Edgelist Version

Instead of operating on adjacency matrix, it can be implemented on edgelist directly. All it took is to iterate the edgelist just twice!

The method is applicable to directed or weighted graphs, as well as graph Laplacian

$$\mathbf{Z} = \mathbf{D}^{-0.5}\mathbf{A}\mathbf{D}^{-0.5}\mathbf{W}.$$

This can also be achieved by iterating through the edgelists two more times.

Edgelist Version

Instead of operating on adjacency matrix, it can be implemented on edgelist directly. All it took is to iterate the edgelist just twice!

The method is applicable to directed or weighted graphs, as well as graph Laplacian

$$\mathbf{Z} = \mathbf{D}^{-0.5}\mathbf{A}\mathbf{D}^{-0.5}\mathbf{W}.$$

This can also be achieved by iterating through the edgelists two more times.

The time and storage complexity are $O(nk + s)$, i.e., linear with respect to the number of vertices and number of edges.

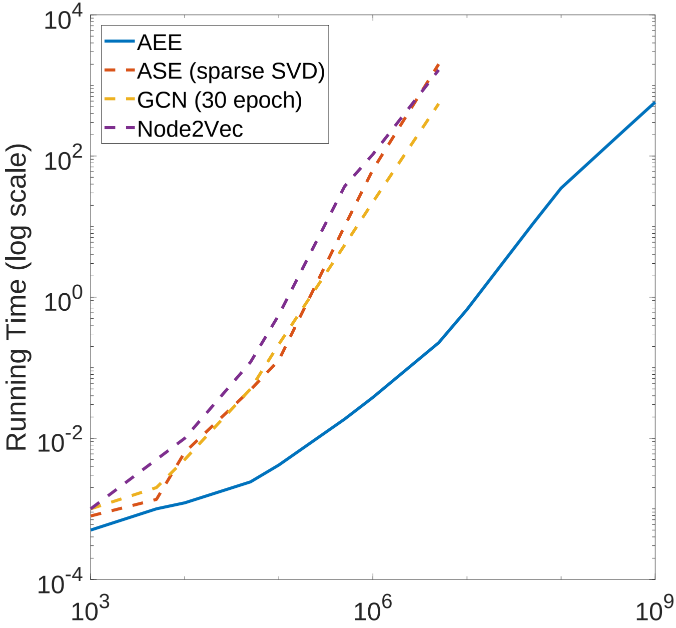
Section 3

Running Time Advantage

In the next figure we plot the average running time of graph encoder embedding using 50 Monte Carlo replicates, on a random graph with $K = 10$, average degree 100, and increasing graph size.

In the next figure we plot the average running time of graph encoder embedding using 50 Monte Carlo replicates, on a random graph with $K = 10$, average degree 100, and increasing graph size.

The number of edges increases from one thousand to one billion. At 1 billion edges with 10 million vertices, the encoder embedding only requires 20GB memory and finishes in 10 minutes. All other methods exceed maximum memory capacity at 10 million edges.


AEE
ASE (sparse SVD)
GCN (30 epoch)
Node2Vec
Running Time (log scale)
10^4
10^2
10^0
10^{-2}
10^{-4}
10^3
10^6
10^9

To validate the running time, we conducted extensive time comparison among various implementations of spectral embedding and node2vec on MATLAB, R, and Python.

[1] https://github.com/microsoft/graspologic/
[2] https://github.com/aditya-grover/node2vec
[3] https://https://github.com/eliorc/node2vec
[4] https://https://github.com/krishnanlab/PecanPy

To validate the running time, we conducted extensive time comparison among various implementations of spectral embedding and node2vec on MATLAB, R, and Python.

For spectral embedding, we compared the MATLAB sparse SVD, R igraph spectral embedding, and Microsoft Research's spectral embedding on Python[1].

---

[1] https://github.com/microsoft/graspologic/

[2] https://github.com/aditya-grover/node2vec

[3] https://https://github.com/eliorc/node2vec

[4] https://https://github.com/krishnanlab/PecanPy

To validate the running time, we conducted extensive time comparison among various implementations of spectral embedding and node2vec on MATLAB, R, and Python.

For spectral embedding, we compared the MATLAB sparse SVD, R igraph spectral embedding, and Microsoft Research's spectral embedding on Python[1].

For node2vec, we compared various implementations from original authors' C and Python code[2], another Python implementation[3], R version, Python code from Microsoft, and PecanPy[4].

[1] https://github.com/microsoft/graspologic/

[2] https://github.com/aditya-grover/node2vec

[3] https://https://github.com/eliorc/node2vec

[4] https://https://github.com/krishnanlab/PecanPy

Section 4

Theoretical Properties

Stochastic Block Model

SBM is arguably the most fundamental community-based random graph model.

Stochastic Block Model

SBM is arguably the most fundamental community-based random graph model.

Each vertex i is associated with a class label $Y_i \in \{1, \ldots, K\}$. The class label may be fixed a-priori, or generated by a categorical distribution with prior probability $\{\pi_k \in (0,1)$ with $\sum_{k=1}^{K} \pi_k = 1\}$.

Stochastic Block Model

SBM is arguably the most fundamental community-based random graph model.

Each vertex i is associated with a class label $Y_i \in \{1, \ldots, K\}$. The class label may be fixed a-priori, or generated by a categorical distribution with prior probability $\{\pi_k \in (0, 1) \text{ with } \sum_{k=1}^{K} \pi_k = 1\}$.

Then a block probability matrix $\mathbf{B} = [\mathbf{B}(k, l)] \in [0, 1]^{K \times K}$ specifies the edge probability between a vertex from class k and a vertex from class l: for any $i < j$,

$$\mathbf{A}(i, j) \overset{i.i.d.}{\sim} \text{Bernoulli}(\mathbf{B}(Y_i, Y_j)),$$
$$\mathbf{A}(i, i) = 0, \quad \mathbf{A}(j, i) = \mathbf{A}(i, j).$$

Degree-Corrected SBM

The DC-SBM graph is a generalization of SBM to better model the sparsity of real graphs.

Degree-Corrected SBM

The DC-SBM graph is a generalization of SBM to better model the sparsity of real graphs.

Everything else being the same as SBM, each vertex i has an additional degree parameter θ_i, and the adjacency matrix is generated by

$$\mathbf{A}(i,j) \sim \text{Bernoulli}(\theta_i \theta_j \mathbf{B}(Y_i, Y_j)).$$

Degree-Corrected SBM

The DC-SBM graph is a generalization of SBM to better model the sparsity of real graphs.

Everything else being the same as SBM, each vertex i has an additional degree parameter θ_i, and the adjacency matrix is generated by

$$\mathbf{A}(i,j) \sim \text{Bernoulli}(\theta_i \theta_j \mathbf{B}(Y_i, Y_j)).$$

The degree parameters typically require certain constraint to ensure a valid probability. In this paper we simply assume they are non-trivial and bounded, i.e., $\theta_i \overset{i.i.d.}{\sim} F_\theta \in (0, M]$, which is a very general assumption.

Random Dot Product Graph

Another popular random graph model is RDPG. Under RDPG, each vertex i is associated with a latent position vector $X_i \stackrel{i.i.d.}{\sim} F_X \in [0,1]^p$.

Random Dot Product Graph

Another popular random graph model is RDPG. Under RDPG, each vertex i is associated with a latent position vector $X_i \stackrel{i.i.d.}{\sim} F_X \in [0,1]^p$.

F_X is constrained such that $X_i^T X_j \in (0,1]$, i.e., the inner product shall be a valid probability. Then the adjacency matrix is generated by

$$\mathbf{A}(i,j) \sim \text{Bernoulli}(X_i^T X_j).$$

Random Dot Product Graph

Another popular random graph model is RDPG. Under RDPG, each vertex i is associated with a latent position vector $X_i \stackrel{i.i.d.}{\sim} F_X \in [0,1]^p$.

F_X is constrained such that $X_i^T X_j \in (0,1]$, i.e., the inner product shall be a valid probability. Then the adjacency matrix is generated by

$$\mathbf{A}(i,j) \sim \text{Bernoulli}(X_i^T X_j).$$

To generate communities under RDPG, it suffices to use a K-component mixture distribution, i.e., let $(X_i, Y_i) \stackrel{i.i.d.}{\sim} F_{XY}$ be a distribution on $\mathbb{R}^p \times [K]$.

Central Limit Theorem

Theorem 1

The graph encoder embedding is asymptotically normally distributed under SBM, DC-SBM, or RDPG. Specifically, as n increases, for a given ith vertex of class y it holds that

$$Diag(\vec{n})^{0.5} \cdot (\mathbf{Z}_i - \mu) \xrightarrow{d} \mathcal{N}(0, \Sigma),$$

where $\vec{n} = [n_1, n_2, \cdots, n_k] \in \mathbb{R}^K$, *and* $Diag(\cdot)$ *is the diagonal matrix of a vector.*

Central Limit Theorem

Theorem 1

The graph encoder embedding is asymptotically normally distributed under SBM, DC-SBM, or RDPG. Specifically, as n increases, for a given ith vertex of class y it holds that

$$Diag(\vec{n})^{0.5} \cdot (\mathbf{Z}_i - \mu) \xrightarrow{d} \mathcal{N}(0, \Sigma),$$

where $\vec{n} = [n_1, n_2, \cdots, n_k] \in \mathbb{R}^K$, and $Diag(\cdot)$ is the diagonal matrix of a vector. The expectation and covariance are:

- *under SBM, $\mu = \mathbf{B}(:, y)$ and $\Sigma = \Sigma_{\mathbf{B}(:,y)}$;*
- *under DC-SBM, $\mu = \theta_i \mathbf{B}(:, y) \odot \bar{\Theta}^{(1)}$ and $\Sigma = \theta_i^2 Diag(\bar{\Theta}^{(2)}) \cdot \Sigma_{\mathbf{B}(:,y)}$;*
- *under RDPG, $\mu = \bar{\lambda}_{x_i}^{(1)}$ and $\Sigma = Diag(\bar{\lambda}_{x_i}^{(1)} - \bar{\lambda}_{x_i}^{(2)})$.*

Central Limit Theorem

- Under SBM with block matrix $\mathbf{B}$, define $\Sigma_{\mathbf{B}(:,y)}$ as the $K \times K$ diagonal matrix with

$$\Sigma_{\mathbf{B}(:,y)}(k,k) = \mathbf{B}(k,y)(1-\mathbf{B}(k,y)) \in [0, \frac{1}{4}].$$

Central Limit Theorem

- Under SBM with block matrix $\mathbf{B}$, define $\Sigma_{\mathbf{B}(:,y)}$ as the $K \times K$ diagonal matrix with

$$\Sigma_{\mathbf{B}(:,y)}(k,k) = \mathbf{B}(k,y)(1-\mathbf{B}(k,y)) \in [0, \frac{1}{4}].$$

- Under DC-SBM with $\{\theta_j \overset{i.i.d.}{\sim} F_\theta\}$, for any tth moment we define:

$$\bar{\theta}_k^{(t)} = E(\theta_j^t | Y_j = k),$$
$$\bar{\Theta}^{(t)} = [\bar{\theta}_{(1)}^{(t)}, \bar{\theta}_{(2)}^{(t)}, \cdots, \bar{\theta}_{(K)}^{(t)}] \in \mathbb{R}^K.$$

Central Limit Theorem

- Under SBM with block matrix $\mathbf{B}$, define $\Sigma_{\mathbf{B}(:,y)}$ as the $K \times K$ diagonal matrix with

$$\Sigma_{\mathbf{B}(:,y)}(k,k) = \mathbf{B}(k,y)(1-\mathbf{B}(k,y)) \in [0, \frac{1}{4}].$$

- Under DC-SBM with $\{\theta_j \overset{i.i.d.}{\sim} F_\theta\}$, for any tth moment we define:

$$\begin{aligned}\bar{\theta}_k^{(t)} &= E(\theta_j^t | Y_j = k),\\ \bar{\Theta}^{(t)} &= [\bar{\theta}_{(1)}^{(t)}, \bar{\theta}_{(2)}^{(t)}, \cdots, \bar{\theta}_{(K)}^{(t)}] \in \mathbb{R}^K.\end{aligned}$$

- Under RDPG where $(X,Y) \sim F_{XY} \in \mathbb{R}^p \times [K]$ is the latent distribution, define

$$\begin{aligned}\bar{\lambda}_k^{(t)}(x_i) &= E^t(X^T x_i | Y = k),\\ \bar{\lambda}_{x_i}^{(t)} &= [\bar{\lambda}_1^{(t)}(x_i), \bar{\lambda}_2^{(t)}(x_i), \cdots, \bar{\lambda}_K^{(t)}(x_i)] \in \mathbb{R}^K\end{aligned}$$

for any fixed vector $x_i \in \mathbb{R}^p$.

Law of Large Numbers

Corollary 1 (Consistency)

Using the same notation as in Theorem 1. It always holds that

$$\|\mathbf{Z}_i - \mu\|_2 \overset{n\rightarrow\infty}{\rightarrow} 0.$$

Law of Large Numbers

Corollary 1 (Consistency)

Using the same notation as in Theorem 1. It always holds that

$$\|\mathbf{Z}_i - \mu\|_2 \stackrel{n \to \infty}{\to} 0.$$

Alternatively, dimension-wise the above translates to:

$$\mathbf{Z}_i[k] \to E[\mathbf{A}_{ij} | Y_j = k],$$

which estimates the probability of vertex i being adjacent to a random vertex from class k, thus very informative and interpretable on its meaning.

Law of Large Numbers

Corollary 1 (Consistency)

Using the same notation as in Theorem 1. It always holds that

$$\|\mathbf{Z}_i - \mu\|_2 \stackrel{n \to \infty}{\to} 0.$$

Alternatively, dimension-wise the above translates to:

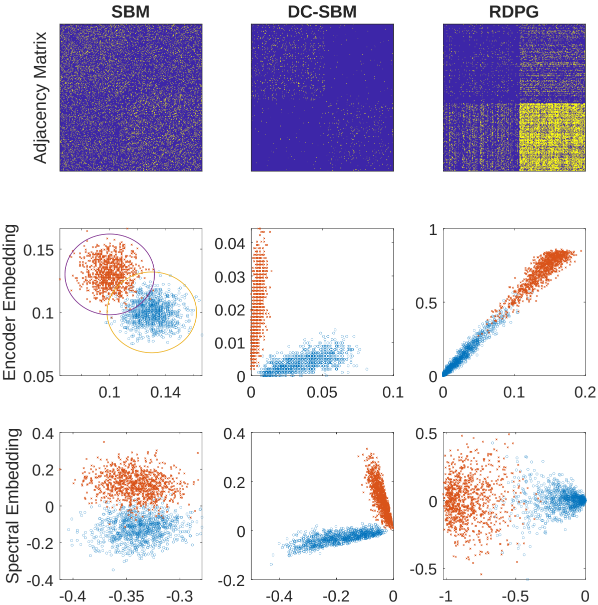
$$\mathbf{Z}_i[k] \to E[\mathbf{A}_{ij} | Y_j = k],$$

which estimates the probability of vertex i being adjacent to a random vertex from class k, thus very informative and interpretable on its meaning.

The asymptotic normality and asymptotic convergence also hold for weighted graphs.

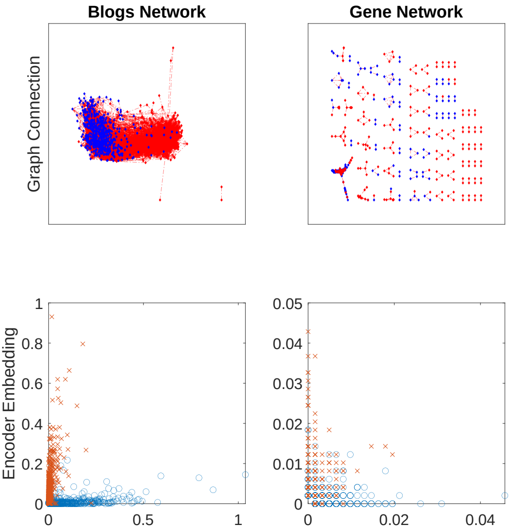
Visualization

We first compare the graph encoder embedding to the spectral embedding under SBM, DC-SBM, and RDPG graphs at $K = 2$. While both methods exhibit clear community separation, the encoder embedding provides better estimation for the model parameters.


SBM
DC-SBM
RDPG
Adjacency Matrix
Encoder Embedding
Spectral Embedding
0.15
0.1
0.05
0.1
0.14
0.04
0.03
0.02
0.01
0
0
0.05
0.1
1
0.5
0
0
0.1
0.2
0.4
0.2
0
-0.2
-0.4
-0.4
-0.35
-0.3
0.4
0.2
0
-0.2
-0.4
-0.2
0
0.5
0
-0.5
-1
-0.5
0

Visualization

Then we inspect graph encoder embedding for the Political Blogs (1490 vertices with 2 classes) and the Gene Network (1103 vertices with 2 classes). Both graphs are sparse. The average degree is 22.4 for the Political Blogs and 1.5 for the Gene Network.


Blogs Network
Gene Network
Graph Connection
Encoder Embedding
1
0.8
0.6
0.4
0.2
0
0
0.5
1
0.05
0.04
0.03
0.02
0.01
0
0
0.02
0.04

Section 5

Vertex Classification

Simulated Data

An immediate and important use case herein is vertex classification. The vertex embedding with known class labels are the training data (labels with class 1 to K), while the vertex embedding with unknown labels are the testing data (labels set to 0 in the method).

Simulated Data

An immediate and important use case herein is vertex classification. The vertex embedding with known class labels are the training data (labels with class 1 to K), while the vertex embedding with unknown labels are the testing data (labels set to 0 in the method).

We consider five graph embedding methods: adjacency encoder embedding (AEE), Laplacian encoder embedding (LEE), adjacency spectral embedding (ASE), Laplacian spectral embedding (LSE), and node2vec.

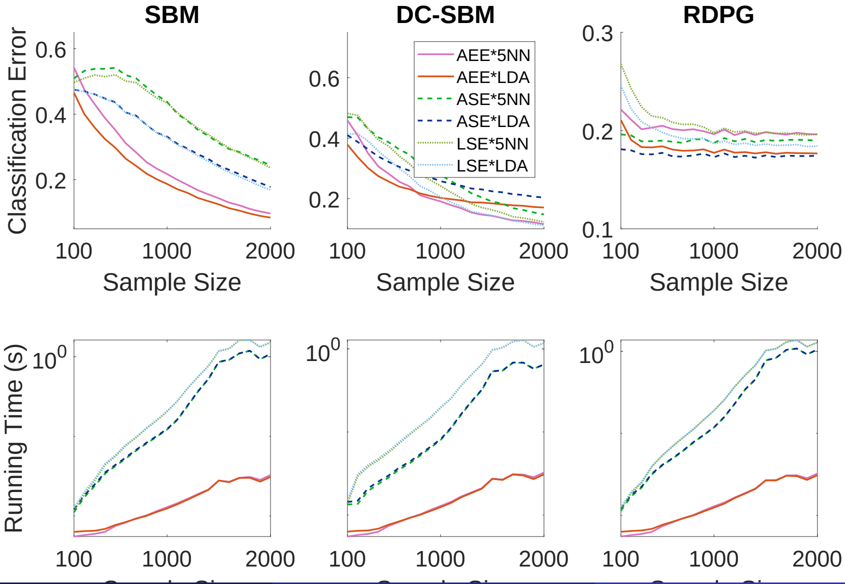
Simulated Data

An immediate and important use case herein is vertex classification. The vertex embedding with known class labels are the training data (labels with class 1 to K), while the vertex embedding with unknown labels are the testing data (labels set to 0 in the method).

We consider five graph embedding methods: adjacency encoder embedding (AEE), Laplacian encoder embedding (LEE), adjacency spectral embedding (ASE), Laplacian spectral embedding (LSE), and node2vec.

For every embedding, we use linear discriminant analysis (LDA) and 5-nearest-neighbor (5NN) as the follow-on classifiers. Other classifier like logistic regression, random forest, and neural network can also be used. We observe similar accuracy regardless of the classifiers, implying that the learning task largely depends on the embedding method.

Simulated Data



Real Data

We downloaded a variety of public real graphs with labels, including three graphs from network repository[5]:

- Cora Citations (2708 vertices, 5429 edges, 7 classes),
- Gene Network (1103 vertices, 1672 edges, 2 classes),
- Industry Partnerships (219 vertices, 630 edges, 3 classes);

and three more graphs from Stanford network data[6]:

- EU Email Network (1005 vertices, 25571 edges, 42 classes),
- LastFM Asia Social Network (7624 vertices, 27806 edges, 17 classes),
- Political Blogs (1490 vertices, 33433 edges, 2 classes).

[5] `http://networkrepository.com/`

[6] `https://snap.stanford.edu/`

Classification Error

	AEE	LEE	ASE	LSE	N2v	Chance
Cora	16.3%	**15.5%**	31.0%	33.1%	16.3%	69.8%
Email	30.6%	28.3%	30.8%	39.5%	**26.1%**	89.2%
Gene	17.1%	**16.5%**	27.2%	36.2%	21.9%	44.4%
Industry	**29.7%**	30.7%	38.8%	39.2%	32.9%	39.3%
LastFM	15.5%	15.0%	20.1%	16.5%	**14.5%**	79.4%
PolBlog	4.9%	5.0%	5.5%	**4.0%**	4.5%	48.0%

Running Time (seconds)

	AEE	LEE	ASE	LSE	N2v	
Cora	**0.01**	**0.01**	1.55	1.60	2.1	
Email	**0.02**	0.03	0.12	0.15	1.2	
Gene	**0.01**	**0.01**	0.15	0.18	0.80	
Industry	**0.01**	**0.01**	0.02	0.02	0.25	
LastFM	**0.02**	0.03	13.0	15.3	9.2	
PolBlog	**0.01**	0.02	0.27	0.28	1.2	

Section 6

Vertex Clustering

Many graph data are collected without ground-truth vertex labels.
Therefore, we also design a no-label version of graph encoder embedding.

Many graph data are collected without ground-truth vertex labels. Therefore, we also design a no-label version of graph encoder embedding.

Starting with random label initialization, we run encoder embedding and k-means clustering to iteratively refine the vertex embedding and label assignments.

Many graph data are collected without ground-truth vertex labels. Therefore, we also design a no-label version of graph encoder embedding.

Starting with random label initialization, we run encoder embedding and k-means clustering to iteratively refine the vertex embedding and label assignments.

The algorithm stops when the labels no longer change or the maximum iteration limit is reached.

Graph Encoder Embedding Without Labels

- **Input**: An adjacency matrix $\mathbf{A}$ (or edgelist), desired number of classes K, and maximum iteration limits M (by default we set $M = 20$).

Graph Encoder Embedding Without Labels

- **Input**: An adjacency matrix $\mathbf{A}$ (or edgelist), desired number of classes K, and maximum iteration limits M (by default we set $M = 20$).
- **Step 1**: Randomly initialize a label vector of K classes for each vertex, denoted by $\mathbf{Y}_2$.

Graph Encoder Embedding Without Labels

- **Input**: An adjacency matrix $\mathbf{A}$ (or edgelist), desired number of classes K, and maximum iteration limits M (by default we set $M = 20$).
- **Step 1**: Randomly initialize a label vector of K classes for each vertex, denoted by $\mathbf{Y}_2$.
- **Step 2**: Run GEE with label, and output the encoder embedding $\mathbf{Z}$.

Graph Encoder Embedding Without Labels

- **Input**: An adjacency matrix $\mathbf{A}$ (or edgelist), desired number of classes K, and maximum iteration limits M (by default we set $M = 20$).
- **Step 1**: Randomly initialize a label vector of K classes for each vertex, denoted by $\mathbf{Y}_2$.
- **Step 2**: Run GEE with label, and output the encoder embedding $\mathbf{Z}$.
- **Step 3**: Run K-means on $\mathbf{Z}$ to output a new cluster index $\mathbf{Y}$.

Graph Encoder Embedding Without Labels

- **Input**: An adjacency matrix $\mathbf{A}$ (or edgelist), desired number of classes K, and maximum iteration limits M (by default we set $M = 20$).
- **Step 1**: Randomly initialize a label vector of K classes for each vertex, denoted by $\mathbf{Y}_2$.
- **Step 2**: Run GEE with label, and output the encoder embedding $\mathbf{Z}$.
- **Step 3**: Run K-means on $\mathbf{Z}$ to output a new cluster index $\mathbf{Y}$.
- **Step 4**: If $ARI(\mathbf{Y}, \mathbf{Y}_2) = 1$ or max iteration reached, stop the algorithm. Otherwise set $\mathbf{Y}_2 = \mathbf{Y}$ and repeat from Step 2.

Graph Encoder Embedding Without Labels

- **Input**: An adjacency matrix $\mathbf{A}$ (or edgelist), desired number of classes K, and maximum iteration limits M (by default we set $M = 20$).
- **Step 1**: Randomly initialize a label vector of K classes for each vertex, denoted by $\mathbf{Y}_2$.
- **Step 2**: Run GEE with label, and output the encoder embedding $\mathbf{Z}$.
- **Step 3**: Run K-means on $\mathbf{Z}$ to output a new cluster index $\mathbf{Y}$.
- **Step 4**: If $ARI(\mathbf{Y}, \mathbf{Y}_2) = 1$ or max iteration reached, stop the algorithm. Otherwise set $\mathbf{Y}_2 = \mathbf{Y}$ and repeat from Step 2.
- **Output**: The final embedding $\mathbf{Z} \in \mathbb{R}^{n \times K}$, and final label vector $\mathbf{Y}$.

This is both an embedding and clustering algorithm. The running time is $O(M(nK^2 + s))$, which is still linear with respect to the number of edges and the number of vertices.

This is both an embedding and clustering algorithm. The running time is $O(M(nK^2 + s))$, which is still linear with respect to the number of edges and the number of vertices.

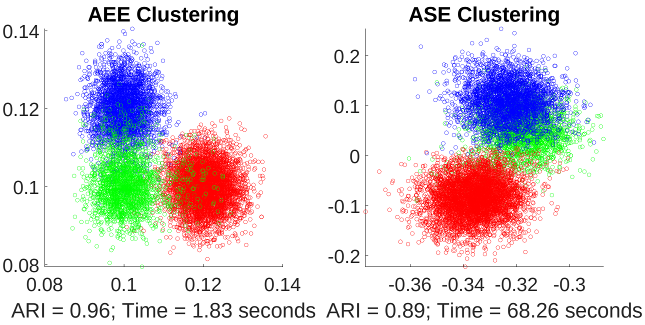
The clustering performance is measured by the adjusted rand index (ARI) between the clustering results and ground-truth labels. ARI lies in $(-\infty, 1]$, with larger positive number implying better matchedness and 1 for perfect match.

This is both an embedding and clustering algorithm. The running time is $O(M(nK^2 + s))$, which is still linear with respect to the number of edges and the number of vertices.

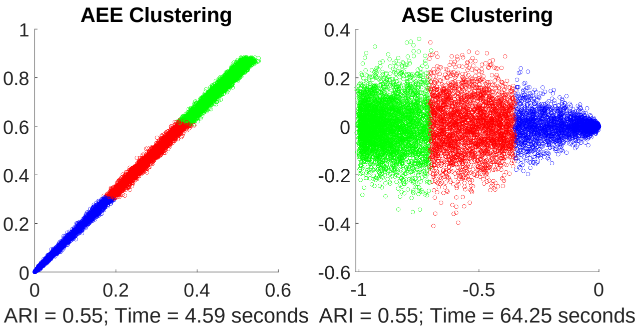
The clustering performance is measured by the adjusted rand index (ARI) between the clustering results and ground-truth labels. ARI lies in $(-\infty, 1]$, with larger positive number implying better matchedness and 1 for perfect match.

Instead of a determined K, we have enhanced the algorithm to work for a range of K then pick the best possible group based on certain metric. Moreover, to guard against possible bad initialization, the algorithm also has a parameter to re-do initialization.

SBM Simulation



RDPG Simulation



Clustering ARI on Same Real Graphs

	AEE	LEE	ASE	LSE	N2v
Cora	0.12	0.07	0.08	0.01	**0.24**
Email	**0.40**	0.39	0.11	0.21	0.34
Gene	0.01	0.01	0.01	0.01	0.00
Industry	**0.13**	0.03	0.01	0.02	**0.13**
LastFM	0.34	0.19	0.03	**0.47**	0.43
PolBlog	**0.80**	0.58	0.07	**0.80**	**0.80**

Running Time (seconds)

	AEE	LEE	ASE	LSE	N2v
Cora	**0.11**	0.12	1.6	1.7	2.2
Email	0.18	0.28	**0.13**	0.20	1.3
Gene	**0.03**	**0.03**	0.17	0.20	0.90
Industry	0.02	0.02	0.02	0.02	0.40
LastFM	**0.35**	0.39	13.6	15.5	9.5
PolBlog	**0.05**	0.07	0.27	0.29	1.4

Conclusion

We developed graph encoder embedding, that is:
1. Simple and elegant to implement in any programming language (we implemented MATLAB, R, Python).

Conclusion

We developed graph encoder embedding, that is:
1. Simple and elegant to implement in any programming language (we implemented MATLAB, R, Python).

2. The most scalable graph embedding method to date, capable of handling billions of edges on personal computer for vertex classification or clustering.

Conclusion

We developed graph encoder embedding, that is:

1. Simple and elegant to implement in any programming language (we implemented MATLAB, R, Python).

2. The most scalable graph embedding method to date, capable of handling billions of edges on personal computer for vertex classification or clustering.

3. Theoretically sound: satisfy central limit theorem and law of large numbers; estimate the block probability and latent positions (up-to transformation); and asymptotically optimal for vertex classification.

Conclusion

We developed graph encoder embedding, that is:
1. Simple and elegant to implement in any programming language (we implemented MATLAB, R, Python).

2. The most scalable graph embedding method to date, capable of handling billions of edges on personal computer for vertex classification or clustering.

3. Theoretically sound: satisfy central limit theorem and law of large numbers; estimate the block probability and latent positions (up-to transformation); and asymptotically optimal for vertex classification.

4. Excellent numerical performance: achieved excellent performance in classification error and clustering ARI throughout synthetic and real data experiments. The performance is on par with existing approaches but much faster.