node2vec: Scalable Feature Learning for Networks

Authors: Aditya Grover and Jure Leskovec

Chenyu Shi and Shupei Li

Leiden Institute of Advanced Computer Science

November 18, 2022

Universiteit Leiden
The Netherlands

Introduction to Graph Embeddings

Related Work

Feature Learning Framework

Aim: Given a network G=(V, E), find a projection $f: V \rightarrow R^d$

Generate a d-dimesion vector representation for each node

f can be formulated as a matrix of size $|V| \cdot d$

Feature Learning Framework

Extending skip graph architecture to networks.

Formulate feature learning in networks as a maximum likelihood optimization problem:

$$\max_{f} \sum_{u \in V} \log Pr(N_S(u) | f(u))$$

$N_s(a)$ is the network neighborhood set generated by neighborhood sampling strategy S for node a

Important: $N_s(a)$ isn't equavalent to direct local neighborhood

For NLP: This is a $\boxed{\text{feature learning framework social network}}$

$Pr(\{\text{"feature"}, \text{"learning"}, \text{"social"}, \text{"network"}\} | \text{"framework"})$

For Graph: $N_s(a) = \{b, c, d, e\}$

$Pr(\{b, c, d, e\} \mid a) = Pr(N_s(a) \mid a)$

Feature Learning Framework

Extending skip graph architecture to networks.

Formulate feature learning in networks as a maximum likelihood optimization problem:

$$\max_f \sum_{u \in V} \log Pr(N_S(u)|f(u))$$

$N_s(a)$ is the network neighborhood set generated by neighborhood sampling strategy S for node a

Important: $N_s(a)$ isn't equavalent to direct local neighborhood

For NLP: This is a $\boxed{\text{feature learning framework social network}}$

$Pr(\ \{\text{"feature"}, \text{"learning"}, \text{"social"}, \text{"network"}\} |\ \text{"framework"}\ )$

For Graph: $N_s(a) = \{b, c, d, e\}$

$Pr(\{b, c, d, e\} \mid a) = Pr(N_s(a) \mid a)$

Two problems to be solved:

1. How to define $N_s(a)$?
2. How to compute $Pr(N_s(a) \mid a)$?

Maximum Likehood Optimization

Formulate feature learning in networks as a maximum likelihood optimization problem:

$$\max_f \sum_{u \in V} \log Pr(N_S(u)|f(u))$$

Two standard assumptions:

1. Conditional independence:

$$Pr(N_S(u)|f(u)) = \prod_{n_i \in N_S(u)} Pr(n_i|f(u))$$

2. Symmetry in feature space:

$$Pr(n_i|f(u)) = \frac{\exp(f(n_i) \cdot f(u))}{\sum_{v \in V} \exp(f(v) \cdot f(u))}$$

Maximum Likehood Optimization

Formulate feature learning in networks as a maximum likelihood optimization problem:

$$\max_{f} \sum_{u \in V} \log Pr(N_S(u)|f(u))$$

Two standard assumptions:

1. Conditional independence:

$$Pr(N_S(u)|f(u)) = \prod_{n_i \in N_S(u)} Pr(n_i|f(u))$$

2. Symmetry in feature space:

$$Pr(n_i|f(u)) = \frac{\exp(f(n_i) \cdot f(u))}{\sum_{v \in V} \exp(f(v) \cdot f(u))}$$

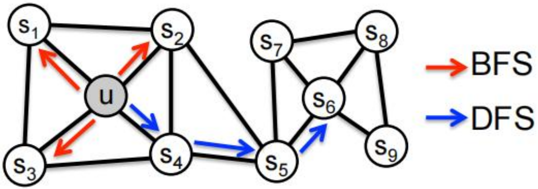
Finally, the optimization problem is converted into the form of:

$$\max_{f} \sum_{u \in V} [-\log(\sum_{u \in V} \exp(f(u) \cdot f(v))) + \sum_{n_i \in N_s(u)} f(n_i) \cdot f(u)]$$

Use gradient decent stochastic to obtain projection f

Network Neighborhood Sampling Strategy

Use classic search strategies:
Breadth-first Sampling (BFS) and Depth-first Sampling (DFS)



There are two kinds of similarities:

1. homophily (such as u and s1)
2. structural equivalence (such as u and s6)

DFS tends to discover homophily , BFS tends to discover structural equivalence

How to discover both kinds of similarities?

Network Neighborhood Sampling Strategy

Use basic random walk to discover both
homophily and structural equalvalence similarities

basic random walk with length l from source node u :

$$P(c_i = x \mid c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases}$$

c_i : the i-th node in the walk

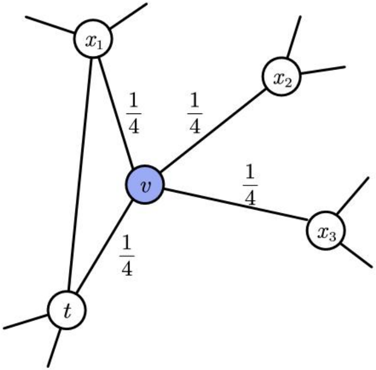
v : current node

π_{vx} : unnormalized transition probability

Z : normalization constant

Network Neighborhood Sampling Strategy

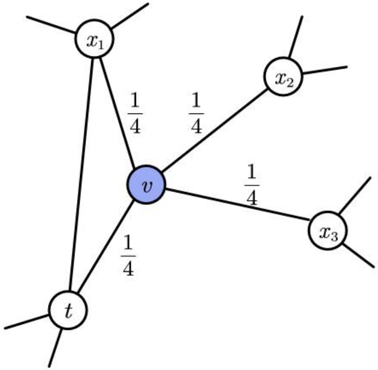
π_{vx} : often set $\pi_{vx} = w_{vx}$ in weighted graphs.
in unweighted graph : $\pi_{vx} = 1$



π_{vx} : often set $\pi_{vx} = w_{vx}$ in weighted graphs.
in unweighted graph : $\pi_{vx} = 1$

Network Neighborhood Sampling Strategy

π_{vx} : often set $\pi_{vx} = w_{vx}$ in weighted graphs.
in unweighted graph : $\pi_{vx} = 1$



Ramdom walk can combine features of DFS and BFS, and discovery both two kinds of similarities

Still not enough:
It's hard for us to guide and control the walking process

Network Neighborhood Sampling Strategy

Use second order bias random walk to get control of the walking process

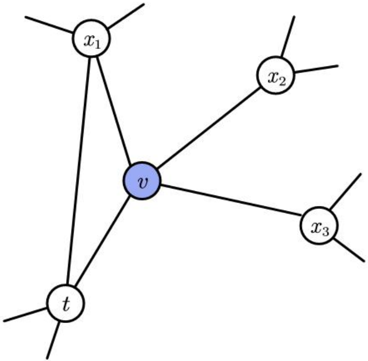
$$\pi_{vx} = \alpha_{pq}(t,x) \cdot w_{vx}$$

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

v : current node

t : last node in the walk

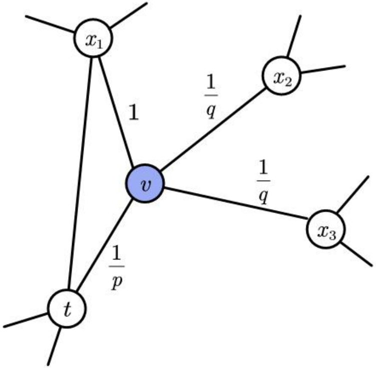
x : the next node to be chosed



Network Neighborhood Sampling Strategy

p : return parameter

q : in-out parameter



p :

- when set a high value: less likely to sample an already visited node
- when set a low value: likely to step back, then walk locally near the source node u

q :

- when set a high value: biased towards nodes close to t, act more similarly to BFS
- when set a low value: biased towards nodes distant to t, act more similarly to DFS

Learning Edge Features

Use node2vec, we have found a projection $f: V \rightarrow R^d$, which allocates each node vector embedding representation.

These embedding vectors can be used in node-related downstream tasks

But how to learn edge features and deal with edge-related downstream tasks?

Learning Edge Features

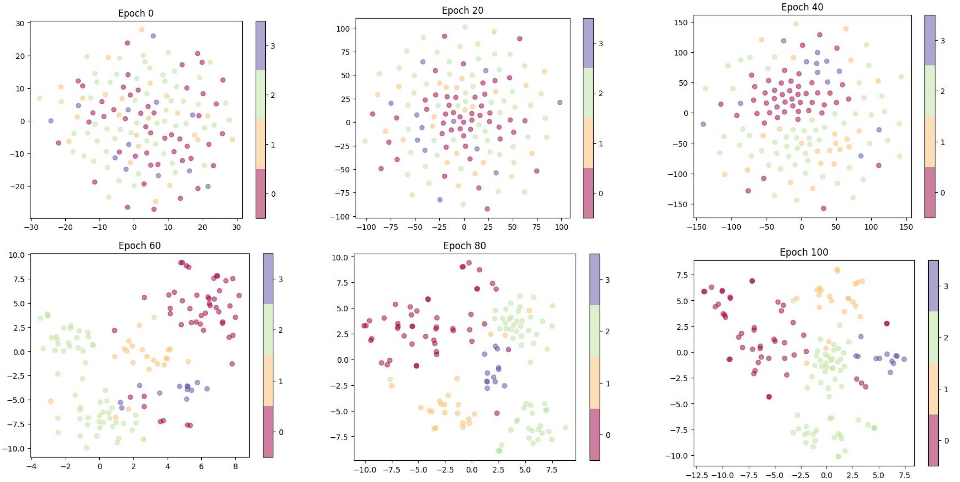
Use node2vec, we have found a projection $f: V \rightarrow R^d$, which allocates each node vector embedding representation.

These embedding vectors can be used in node-related downstream tasks

But how to learn edge features and deal with edge-related downstream tasks?

Given projection f obtained by node2vec and two nodes u and v along with edge (u,v), apply binary operator on f(u) and f(v) to generate representation g(u,v), where $g: V \times V \rightarrow R^{d'}$

Operator	**Symbol**	**Definition**
Average	$\boxplus$	$[f(u) \boxplus f(v)]_i = \frac{f_i(u)+f_i(v)}{2}$
Hadamard	$\boxdot$	$[f(u) \boxdot f(v)]_i = f_i(u) * f_i(v)$
Weighted-L1	$\|\cdot\|_{\bar{1}}$	$\|f(u) \cdot f(v)\|_{\bar{1}i} = \|f_i(u) - f_i(v)\|$
Weighted-L2	$\|\cdot\|_{\bar{2}}$	$\|f(u) \cdot f(v)\|_{\bar{2}i} = \|f_i(u) - f_i(v)\|^2$



AIFB dataset: each node has a category label, there are 4 claases in total

With training node2vec, nodes embedding change from chaos to

Graph Neural Network

Graph Neural Network is a deep learning framework for graph

General GNN iteration formula:

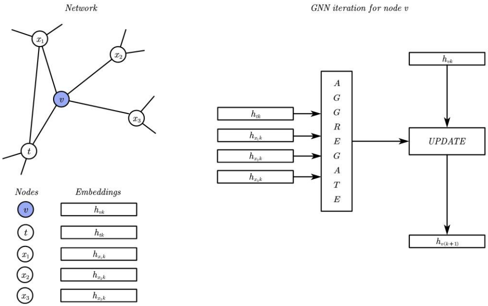
$$h_u^{(k)} = \sigma(W_{self}^{(k)} h_u^{(k-1)} + W_n^{(k)} \sum_{v \in N(u)} h_v^{(k-1)} + b^{(k)})$$

$h_u^{(k)}$: k-th layer output embedding of node u

$W^{(k)}$: weights of k-th layer (trainable)

$b^{(k)}$: bias of k-th layer (trainable)

σ : activation function



Aggregate: To aggregate embedings of u’s neighborhood

Update: To update u’s embbeding using aggregate result and previous u’s embbeding

Graph Neural Network

$$h_u^{(k)} = \sigma(W_{self}^{(k)} h_u^{(k-1)} + W_n^{(k)} \sum_{v \in N(u)} h_v^{(k-1)} + b^{(k)})$$

Fianl output embedding can be used for calculating predictions, and use these predictions to compute loss and use back propagation to train parameters

Graph Neural Network

$$h_u^{(k)} = \sigma(W_{self}^{(k)} h_u^{(k-1)} + W_n^{(k)} \sum_{v \in N(u)} h_v^{(k-1)} + b^{(k)})$$

Fianl output embedding can be used for calculating predictions, and use these predictions to compute lossand use back propagation to train parameters

Problem: how to produce h_u^0 for each node?

1. use one-hot vector for each node
-- drawback: the total number of nodes are large, use one-hot vector for each node will cause the input tensor too sparse

Graph Neural Network

$$h_u^{(k)} = \sigma(W_{self}^{(k)} h_u^{(k-1)} + W_n^{(k)} \sum_{v \in N(u)} h_v^{(k-1)} + b^{(k)})$$

Fianl output embedding can be used for calculating predictions, and use these predictions to compute lossand use back propagation to train parameters

Problem: how to produce h_u^0 for each node?

1. use one-hot vector for each node
-- drawback: the total number of nodes are large, use one-hot vector for each node will cause the input tensor too sparse

2. transfer pretrained embedding from other similar tasks
-- drawback: not all networks can find a similar tasks to import embedding

Graph Neural Network

$$h_u^{(k)} = \sigma(W_{self}^{(k)} h_u^{(k-1)} + W_n^{(k)} \sum_{v \in N(u)} h_v^{(k-1)} + b^{(k)})$$

Fianl output embedding can be used for calculating predictions, and use these predictions to compute lossand use back propagation to train parameters

Problem: how to produce h_u^0 for each node?

1. use one-hot vector for each node
-- drawback: the total number of nodes are large, use one-hot vector for each node will cause the input tensor too sparse

2. transfer pretrained embedding from other similar tasks
-- drawback: not all networks can find a similar tasks to import embedding

3. Use trainable embbeding layer to allocate random initialized embedding for each node
-- drawback: random initialized embedding could have negative impact on training process

Combine node2vec and GNN

node2vec can produce embedding for each node with graph information

GNN lacks a good general method to initialize its input nodes' embedding

We could use node2vec to produce initial input nodes' embedding for GNN to improve traning process of GNN and obtain better performance

It's a general method because there is no specific requirement of graphs when applying node2vec and GNN