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

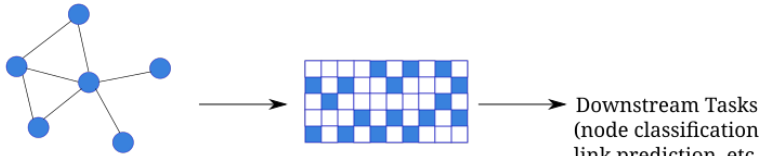
Universiteit Leiden
The Netherlands

This is the title page

This is the second page

Introduction to Graph Embeddings

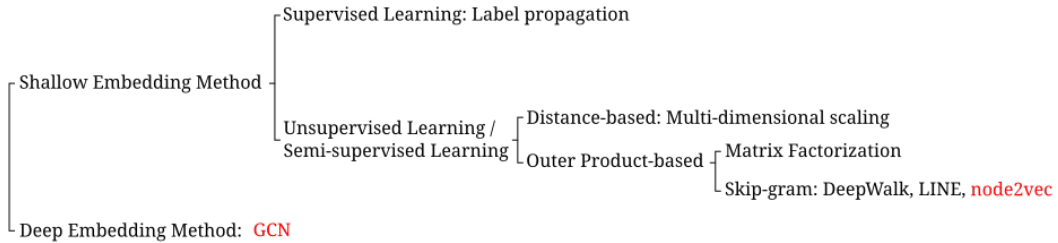
- Represent graph-structured data.
- Applications:
  Social network analysis, recommender systems, molecular structure modelling, etc.
- Challenge: Limitations of traditional methods.
- Development of techniques specially for graph representations.



- node2vec is a feature learning framework.

Related Work

A taxonomy of graph embedding techniques[1].



[1] **murphy2022**.

Feature Learning Framework

- node2vec is a feature learning framework in nature.
- Goal: 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 $|V| \cdot d$.

Now we start to talk about methodology of node2vec. As we have shown on previous slides, node2vec is a feature learning framework in nature. So let's give feature learning framework a formal defination. Feature learning framework aims to find a projection f, which projects nodes set V to vector space. In other words, the task of a feature learning framework like node2vec is to generate a d-dimension embedding vector representation for each node. So, in mathematics form, f can be formulated as a matrix of size v times d.

Feature Learning Framework

Extending skip gram architecture to networks.

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

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

$N_S\left(a\right)$ is the network neighborhood set generated by neighborhood sampling strategy S for node a.

Important: $N_S\left(a\right)$ isn't equavalent to direct local neighborhood.

For NLP: Given a literal data: *This is a [feature learning framework social network].*

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

For Graph data:

$$N_S\left(a\right) = \{b, c, d, e\}$$
$$Pr\left(\{b, c, d, e\} | a\right) = Pr\left(N_s\left(a\right) | a\right)$$

Put that graph here!!

So how to determine this projection f? Inspired by skip gram architecture in Natural language processing area, the authors formulate this problem as a maximum likelihood optimization problem. In NLP area, the given dataset contains literal data and we want to generate embedding vectors for each word. And skip gram architecture is to use a sliding window to obtain nearest neighbors for a word. Like here in the rightside example, in this sentence, feature - learning - social - network is in the sliding window of the word framework. Then it computes and optimizes the likelihood probability for these four words in the sliding window to a maximum value. Then the literal structure information such as order of words in a sentence can be maintained most in the words embedding. In a similar way, now we are given a network data and wants to generate embedding vectors for each node. And at the same time we want our embedding to maintain as much as possible graph structure information. So we should have something similar to a sliding window in NLP problem. Perhaps for network data, it's not as intuitional as NLP problem to define this sliding window. In fact, the equavalent sliding window for network data is called network neighborhood Ns. Here please pay attention to definition of Ns. This network neighborhood isn't equivalent to direct local neighborhood. We know the commonly used direct local neighborhood is totally decided by the graph structure. But this Ns is not only related to the graph structure, but also related to a sampling strategy. And then we can compute the likelihood probability just like what we do in NLP area.

Feature Learning Framework

Extending skip gram architecture to networks.

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

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

$N_S\left(a\right)$ is the network neighborhood set generated by neighborhood sampling strategy S for node a.

Important: $N_S\left(a\right)$ isn't equavalent to direct local neighborhood.

For NLP: Given a literal data: *This is a [feature learning framework social network].*

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

For Graph data:

$$N_S\left(a\right) = \{b, c, d, e\}$$
$$Pr\left(\{b, c, d, e\} | a\right) = Pr\left(N_s\left(a\right) | a\right)$$

Put that graph here!! Two problems to be solved:

1. How to define $N_S\left(a\right)$?
2. How to compute $Pr\left(N_S\left(a\right) | a\right)$?

So here comes two key problems. First is how to define this network neighborhood and its corresponding sampling strategy S? The second is given Ns, how can we compute this likelihood probability for each node.

Maximum Likehood Optimization

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

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

Two standard assumptions:

1. Conditional independence:

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

2. Symmetry in feature space:

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

Let's see the second question first. To compute this formula, the authors made two assumptions to simplify the calculation. The first is conditional independence, which means the likelihood probablity to observe a neighborhood node is independent from observing any other neighborhood node. So because of the property of independent event, the likelihood probability can be rewritten in the form of this multiplication way. The second is symmetry in feature space, which means a source node and neighborhood node have a symmetric effect over each other in feature space. And then the authors model the likelihood probability for each pair of nodes as a softmax function.

Maximum Likehood Optimization

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

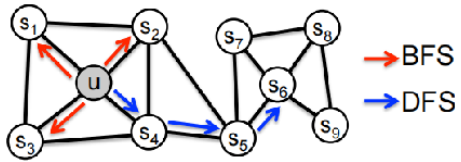
$$\max_f \sum_{u \in V} \left[-\log \left(\sum_{u \in V} \exp\left(f(u) \cdot f(v)\right) \right) + \sum_{n_i \in N_S(u)} f(n_i) \cdot f(u) \right]$$

Use stochastic gradient decent to obtain projection f.

So, with these two assumptions, the original optimization problem can be converted into the rightside formula. And for this formula, we can apply stochastic gradient decent method to obtain the optimal 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 s_1)
2. structural equivalence (such as u and s_6)

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

How to discover both kinds of similarities?

Now we come to the first problem, which is the most important part of node2vec, how to define sampling strategy and network neighborhood. Perhaps you are thinking why don't we directly apply commonly used local neighborhood as sliding window. That's because there are two kinds of similarities, one is homophily, the other is structural equivalence. Let me explain in this example, homophily means two nodes are in the same group or community, such as u and s1. Structural equivalence means the nodes play a similar role in network, such as a bridge or a hub. The node u and s6 in this graph are both hub in their community so they share a structural equivalence. Directly applying local neighborhood is like using classic search strategy BFS, it tends to discover structural equivalence. On the contrary, there are another classic search strategy, DFS, tends to discover homophily. So, applying sampling method like this cannot discover both kind of two similarities at the same time and then fail to capture entire network features information. Therefore, we should come up with a better sampling method to combine BFS and DFS.

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:

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

c_i: the i-th node in the walk.

v: current node.

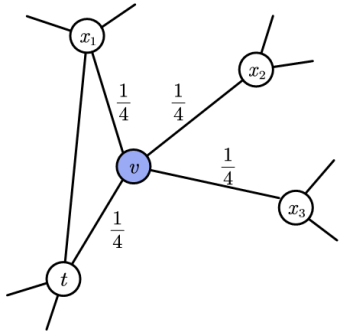
π_{vx}: unnormalized transition probability.

Z: normalization constant.

Then we have a method called random walk which allows us to achieve it. Random walk is a method to take L steps to visit several nodes. And we use the nodes which are visited in a walk as the network neighborhood of the source node. The most important thing for random walk is to decide which node to go in the next step. The formula on the slides gives out the transition probability. And you can see in this formula, there's a pi v x, the unnormalized transition probability, needed to be determined.

Network Neighborhood Sampling Strategy

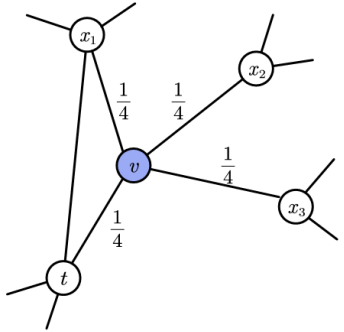
π_{vx}: Often set $\pi_{vx} = w_{vx}$ in weighted graphs.
In unweighted graph: $\pi_{vx} = 1$.



Usually, the pi vx is set equal to the edge weight. And in unweighted garph, pi vx is set to 1, which means the all the neighborhood nodes will equally share the probability of being visited in the next step. So in this example, v is the current node, so node t and x1 2 3 will each have 1 over 4 probability to be visited in the next step.

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.

Random walk can allows us to visit and sample local nodes which are near to the source node, but also allows us to visit distant nodes which are far away from the source node. So it can combine features of DFS and BFS, and then can discover both kinds of similarities. But it's still not enough. Because the entire walking process is random, it's hard for us to guide and control the walking process.

Network Neighborhood Sampling Strategy

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

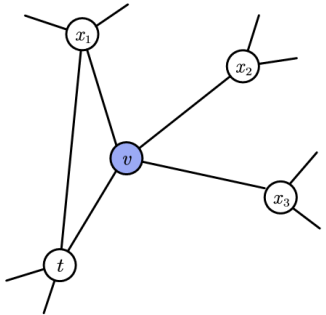
$$\pi_{vx} = \alpha_{pq}(t,x) \cdot w_{vx}$$

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

v: current node.

t: last node in the walk.

x: next node to be chosen.

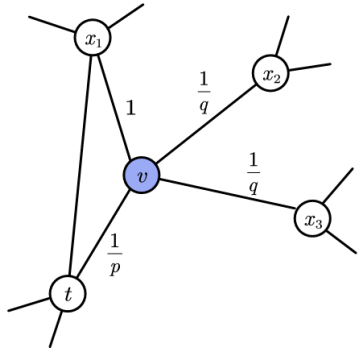


So the authors came up with an upgraded version, the second order bias random walk, which uses a bias value alpha, together with edge weight to decide the transition probability pi vx. And suppose currently we are in node v and want to decide which node to go in the next step. There are always three choices for us. The first choice is to step back, which means to return to node t, where we come from. The second is to stay around, which means to stay in the ego network of node t. The third choice is to go away, which means to go to a new node and jump out of the ego network of t. And in this second order bias random walk, we give these three choices different transition probability. As you can see in the formula, dtx= zero one two means we choose to step back, stay around, and go away respectively. And these transition probability is controlled by two hyper parameters p and q

Network Neighborhood Sampling Strategy

p: return parameter.

q: in-out parameter.



p :

- High value: less likely to sample an already visited node.
- Low value: likely to step back, then walk locally near the source node u.

q :

- High value: biased towards nodes close to t, act more similarly to BFS.
- Low value: biased towards nodes distant to t, act more similarly to DFS.

And for this example graph, the unnormalized transition probability shall be like this. Through controlling the value of p and q, we can also get control of the walking process. For example, for p, a high value will decrease the transition probability to step back, then the sampling strategy is less likely to sample an already visited node. And for q, a high value will decrease the transition probability for the walk to go away. Then the walk process will be biased towards nodes close to t, and act more similarly to BFS. On the contrary, a low q value will make the walking process biased towards nodes distant from t, and act more similarly to DFS.

Learning Edge Features

We have found a projection $f : V \to R^d$ with node2vec, 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?

Up to now, we have worked out the two problems for node2vec. And then we can allocate each node a vector embedding representation, which can be used in node-related downstream task such as node classification. But how can we learn edge features and deal with edge-related downstream task?

Learning Edge Features

We have found a projection $f : V \to R^d$ with node2vec, 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?

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	$\|\cdot\|_{\bar{2}}$	$\|f(u) \cdot f(v)\|_{\bar{2}_i} = \|f_i(u) - f_i(v)\|^2$

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

The idea is very simple. Since each edge is composed of two nodes, so we can apply a binary operation to obtain a projection g and its corresponding edge embedding vectors. Here this table shows four commonly used binary operator. As long as we have obtained projection f, we can obtain projection g and edge embedding in this simple way.

Experiment 1: Multi-label Classification

- Task description
  - Labels from a finite set $\mathcal{L}$
  - Training: A fraction of nodes and all their labels.
  - Predict the labels for the remaining nodes.
- Data

Dataset	Nodes	Edges	Labels
BlogCatalog	10,312	333,983	39
Protein-Protein Interactions (PPI)	3,890	76,584	50
Wikipedia	4,777	184,812	40

- Metrics: Macro-F1 score.

Experiment 1: Multi-label Classification

- Results

Algorithm	**Dataset**		
	BlogCatalog	PPI	Wikipedia
Spectral Clustering	0.0405	0.0681	0.0395
DeepWalk	0.2110	0.1768	0.1274
LINE	0.0784	0.1447	0.1164
node2vec	**0.2581**	**0.1791**	**0.1552**
node2vec settings (p, q)	0.25, 0.25	4, 1	4, 0.5
Gain of node2vec [%]	**22.3**	**1.3**	**21.8**

- node2vec outperforms the other benchmark algorithms.

Experiment 2: Link Prediction

- Task description
  - A network with a fraction of edges removed.
  - Predict these missing edges.
- Data

Dataset	Nodes	Edges
Facebook	4,039	88,234
Protein-Protein Interactions (PPI)	19,706	390,633
arXiv ASTRO-PH	18,722	198,110

- Metrics: Area Under Curve (AUC) score.

Experiment 2: Link Prediction

- Results

Algorithm	**Dataset**		
	Facebook	PPI	arXiv
Common Neighbors	0.8100	0.7142	0.8153
Jaccard's Coefficient	0.8880	0.7018	0.8067
Adamic-Adar	0.8289	0.7126	0.8315
Pref. Attachment	0.7137	0.6670	0.6996
Spectral Clustering	0.6192	0.4920	0.5740
DeepWalk	**0.9680**	0.7441	0.9340
LINE	0.9490	0.7249	0.8902
node2vec	**0.9680**	**0.7719**	**0.9366**

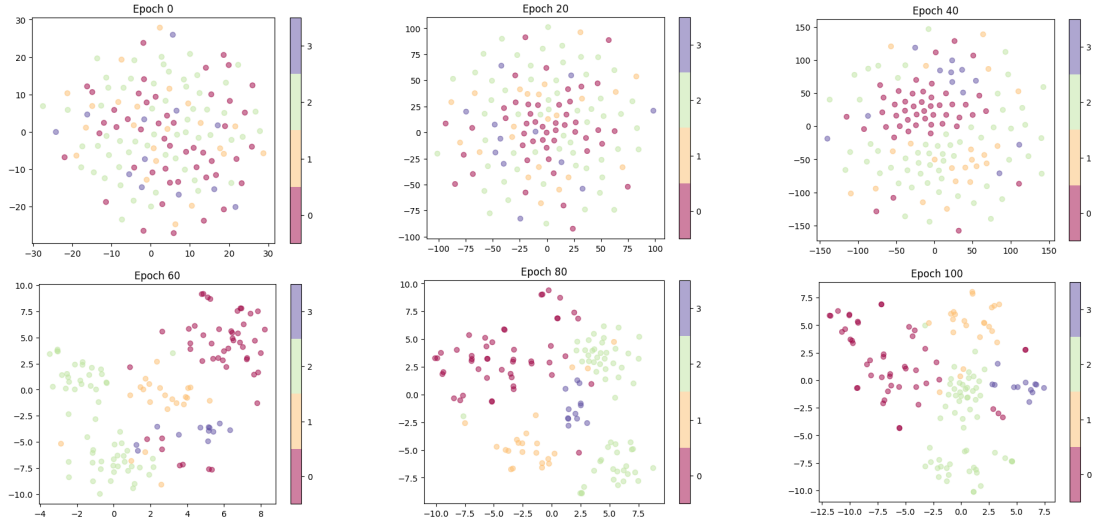
- The learned feature representations outperform heuristic scores. node2vec achieves the best AUC.

Summary of node2vec

- An efficient graph embedding learning algorithm.
- Search strategy: Both flexible and controllable exploring network neighborhoods.

Our Contributions

1. During the training of node2vec, the intermediate state of node embeddings is a black box.
   Our solution: Visualize the node embeddings during the training of node2vec with t-SNE technique.
2. Randomly initialized inputs in GNN affect the robustness of model performance and extend the model training time.
   Our solution: Propose a novel method that uses the pretrained embeddings from node2vec as the meta information for GNN.
3. Effectiveness of algorithms.
   Our solution: Evaluate node2vec, GNN, and our proposed method on five real-world data sets with metrics that are different from the original paper.



AIFB dataset: Each node has a category label. There are 4 classes in total.

During the training of node2vec, nodes embedding are changed from chaos into order.

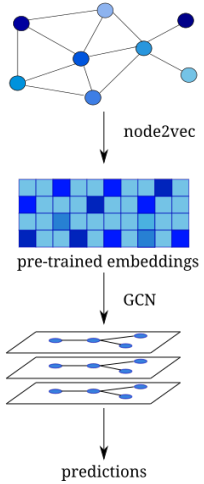
Proposed Method

node2vec + GNN

- Inspired by the concept of meta learning.
- An improved version of GNN.
- Choose the graph convolutional network (GCN) in our experiments.
  - GCN iteration formula:

$$h^{(k)} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}h^{(k-1)}W^{(k)})$$

with $\hat{A} = A + I$, where A is the adjacency matrix and $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$.



Future work

1. Apply node2vec, GCN, and our proposed model to node classification and link prediction.
2. Try different strategies of hyperparameter tuning.

Graph Neural Network

Graph Neural Network is a deep learning framework for graph.

General GNN iteration formula:

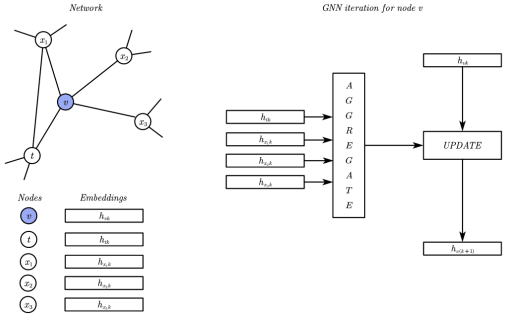
$$h_u^{(k)} = \sigma(W_{\text{self}}^{(k)} h_u^{(k-1)} + W_n^{(k)} \sum_{v \in N(u)} h_v^{(k-1)} + b^{(k)})$$

$h_u^{(k)}$: 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 embeddings of u's neighborhood.

Update: To update u's embeddings using aggregation result and previous u's embeddings.

Graph Neural Network

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

Final output embeddings can be used for calculating predictions. Then, we can use these predictions to compute the loss and optimize parameters with back propagation.

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

1. Use one-hot vector for each node.
   - Drawback: The total number of nodes is large. Using one-hot vector for each node will cause the input tensor to be very sparse.
2. Transfer pretrained embeddings from other similar tasks.
   - Drawback: There can't always be a similar task with pretrained embeddings for every network.
3. Use a trainable embedding layer to allocate randomly initialized embeddings for each node.
   - Drawback: Randomly initialized embeddings could have negative impact on training process.

Combine node2vec and GNN

node2vec can produce embeddings for each node with graph information.

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

We can use node2vec to produce initial input nodes' embeddings for GNN to improve training 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.