Node Embedding

Yujia Zhu

In traditional machine learning for graphs, we need to use hand-craft features. But feature engeering is always time-consuming and complicated. We can change to **Representation Learning** to automatically learn the features. In short, representation learning is to find a function from the node to a low-dimensional vector (embedding space), f : node $u \rightarrow R^d$. The low-dimensional vector is called **Feature Representation/Embedding**.

The goal of node embedding is to encode nodes so that similarity in the embedding space approximates similarity in the graph. The steps of node embedding is as follows.

1. Encode the nodes to the embedding
2. Define a node similarity function (i.e., a measure of similarity in the original network)
3. Transmit the similarity from the embedding vector (calculated by definition) back to the nodes
4. Optimize the parameters of the encoder so that the similarity among the nodes approximate the similarity among the embedding vector

So, there still exist several problems to deal with. Among the problems, the most critical problems include:

1. The structure of the map function from nodes to embedding vectors
2. The measurement of simialrity among the embedding vectors
3. The measurement of simialrity among the nodes

The first and second questions is easy to handle. For similarity measurement of the embedding vectors, the most commonly-used vector similarity measurement is dot production. Namely the similarity $s(u, v)$ between vectors u and v is defined as

$$s(u, v) = u \cdot v = u^T v$$

For the structure of mapping function, the easiest encoding approach is to use an embedding-lookup. Suppose the graph has $|V|$ nodes and the embedding vector is d-dimensional. The map from nodes to vectors is to use a linear transformation, $ENC(z) = Zv$. Z is a matrix with dimension $d \times |V|$. So, the encoder of v-th node is the v-th column of the matrix Z. All the elements in the matrix are the parameters which needs to be optimized.

The last question is the most difficult one. We can use **random walk** (which will be introduced in the next section) to solve it.

Node Embedding is an unspuervised/self-supervised learning task, so it is task-independent. They are not trained for a specific task but can be used for any downstream tasks.

Random Walk

Random walk on a graph can be defined by giving a graph and a starting point. We select a neighbour of the node randomly and move to it. Then we select a neighbour of this point randomly and move to it,etc. After the specified number of steps is reached, we stop and get a sequence of nodes. The sequence of nodes visited in this way is a random walk on the graph.

If random walk starting from node u visits v with high probability, then u and v are similar (because they have high-order multi-hop information). In addition, using random walk to compute similarity is very efficient because we do not need to consider all the node pairs during training. We only need to consider pairs that co-occur in the random walk.

Suppose z_u is the embedding of node u (and the mapping from u to z_u is what we want to learn). Since the goal of embedding is that the similarity among the nodes approximate the similarity among the embedding vectors, we hope $z_u^T z_v \approx$ probability that u, v co-occur on a random walk over the graph. Let $N_R(u)$ is the multiset of nodes visited on random walks starting from node u. Note that $N_R(u)$ can have repeated elements since nodes can be visited for multiple times on random walks. We need to maximize:

$$\max_f \sum_{u \in V} \log P(N_R(u)|z_u)$$

It is equivalent to minimize:

$$L = \sum_{u \in V} \sum_{v \in N_R(u)} -\log P(v|z_u)$$

This means we need to optimize z_u to maximize the likelihood of random walk co-occurrences/minimize the loss. We can parameterize $P(v|z_u)$ by using dot product and softmax:

$$P(v|z_u) = \frac{\exp(z_u^T z_v)}{\sum_{n \in V} \exp(z_n^T z_v)}$$

So, the loss function can be derived as:

$$L = \sum_{u \in V} \sum_{v \in N_R(u)} -\log \left(\frac{\exp(z_u^T z_v)}{\sum_{n \in V} \exp(z_n^T z_v)} \right)$$

We need to optimize random walk embedding so that the embedding vector will minimize the loss L. But optimizing this loss function is too expensive. $\sum_{u \in V}$ part leads to $O(|V|)$ time complexity and $\sum_{n \in V} \exp(z_n^T z_v)$ part also leads to $O(|V|)$ time complexity, so computing L naively gives $O(|V|^2)$ complexity.

The method to reduce time complexity is **negative sampling**. Instead of summing the dot product over all node (i.e. $\sum_{n \in V} \exp(z_n^T z_v)$), we can sample some nodes and only sum dot product over the sampled nodes (i.e. $\sum_{n \sim P_v} \exp(z_n^T z_v)$). The loss function will change to:

$$L = \sum_{u \in V} \sum_{v \in N_R(u)} -\log \left(\frac{\exp(z_u^T z_v)}{\sum_{n \sim P_v} \exp(z_n^T z_v)} \right)$$

$n \sim P_v$ means we need to sample k negative nodes **with proportional to its degree**. The number of negive samples (k) is recommended to be 5-20.

We use stochastic gradient descent to minimize loss function. Each time we only compute loss function on a certain node. For example, suppose node u is selected, the present loss is

$$L^{(u)} = \sum_{v \in N_R(u)} -\log \left(\frac{\exp(z_u^T z_v)}{\sum_{n \sim P_v} \exp(z_n^T z_v)} \right)$$

Then for each j, calculate the gradient $\frac{\partial L^{(u)}}{\partial z_j}$ and update z_j by $z_j := z_j - \eta \frac{\partial L^{(u)}}{\partial z_j}$.

DeepWalk just uses this idea for node embedding. The algorithm steps can be summarized as:

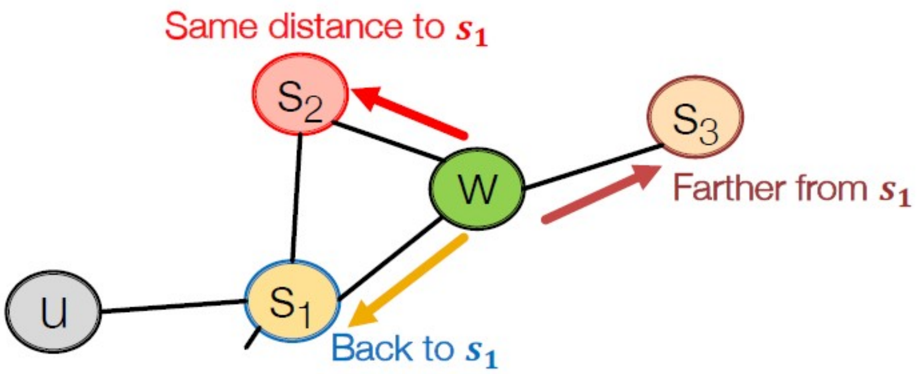
1. Run **short fixed-length random walks** starting from each node on the graph
2. For each node u collect $N_R(u)$, the multiset of nodes visited on random walks starting from u
3. Optimize embeddings using Stochastic Gradient Descent on the loss function (with negative sampling)

Node2Vec

DeepWalk only run fixed-length, unbiased random walks starting from each node. This may cause the restriction of measuring similarity. Node2Vec generalizes this by a new random walk method. Node2Vec uses a biased random walk.

Suppose random walk is now at node w and it has just traversed edge (s_1, w). Now random walk has three choices with different given probability:

- walk back to node s_1
- walk to a node s_2 which has the same distance to s_1
- walk to a node s_3 which is farther from s_1



Moving to a node with same distance is similar to BFS, which captures local/microscopic views of the network, while moving to a farther node is similar to DFS, which captures global/macroscopic views of the network.

We will use two hyper-parameters to allocate the moving probability. p is the return parameter and q is the in-out parameter (the ratio of BFS against DFS). If we give a weight of 1 to moving to a node with same distance, the weight of moving farther and moving back are $1/q, 1/p$ respectively. We normalize three weights$(1, 1/q, 1/p$, so that their sum equals to 1) and get the probability.

Node2Vec only changes the way of random walk. Other parts of the algorithm are the same to the corresponding parts of DeepWalk.

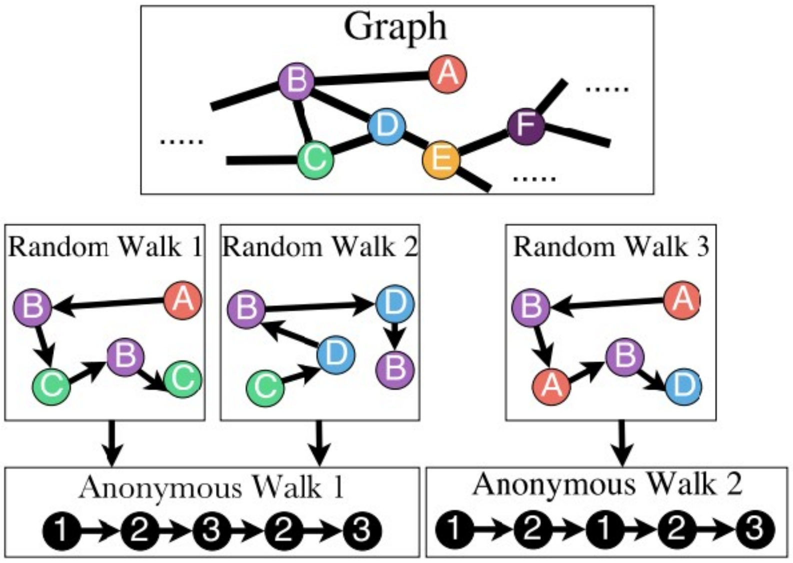
Embed Entire Graph

Now we want to embed the whole graph/subgraph.

The first approach is to first run a node embedding technique on the graph and use the sum/mean/elementwise-max of node embeddings to represent the graph.

The second approach is to add a virtual node. The virtual node is connected with all the nodes in the graph. We will run a node embedding technique on the new graph and use the embedding of the virtual node as the graph embedding.

The third approach is to use anonymous walk embeddings. Anonymous walk only cares about the relative position relationship but not absolute position relationship. Using the following figure as example, the relative position of nodes in random walk 1 and random walk 2 are the same, so the two walks have the same anonymous walk representation.



We will simulate the anonymous walks of l steps for m times and record their counts and we can represent the structure of the graph by the probability distribution over this walks.

The embedding of the graph G, Z_G will be learned together with all the anonymous walk embeddings z_i. The walk embeddings are $Z = \{z_i, i = 1, 2, ..., \eta\}, \eta$ is the number of sampled random walks. For each anonymous random walk, we want to predict the walk in a $\Delta-$size window(i.e. if $\Delta = 1$, then we want to predict w_2 given w_1 and w_3, predict w_3 given w_2 and w_4,etc.). We need to maximize the following equation:

$$\max \sum_{v \in V} \sum_{t=\Delta}^{T-\Delta} \log P(w_t | w_{t-\Delta}, ..., w_{t-1}, w_{t+1}, ..., w_{t+\Delta}, Z_G)$$

This means w_t will not only be predicted by the embedding of node $w_{t-\Delta}$ to $w_{t+\Delta}$ but also the embedding of the graph Z_G. Note that the embedding of the node and the graph should all be learned in the process of optimization. We can parameterize the probability of $P(w_t | w_{t-\Delta}, ..., w_{t-1}, w_{t+1}, ..., w_{t+\Delta}, Z_G)$ by the embedding and softmax. Negative sampling is still needed to speed up calculation:

$$P(w_t | w_{t-\Delta}, ..., w_{t-1}, w_{t+1}, ..., w_{t+\Delta}, Z_G) = \frac{\exp(y(w_t))}{\sum_{i \sim P_w} \exp(y(w_i))}$$

where $y(w_i)$ is concatenation of average node embeddings and graph embedding:

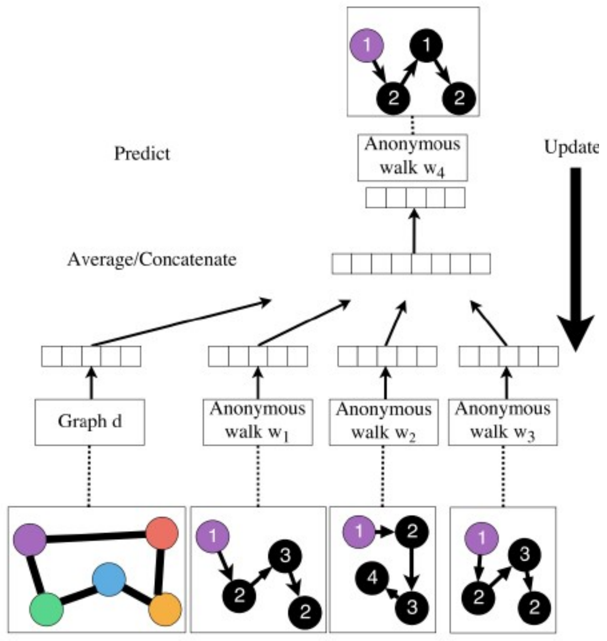
$$y(w_i) = b + U \left(\left[\frac{1}{2\Delta} \sum_{j=-\Delta}^{\Delta} z_{i+j}, Z_G \right] \right)$$

[.] means the concatenation of two vectors, b, U are all learnable parameters. So, after optimization(using SGD), the embedding of the graph Z_G can be obtained and we can use Z_G to make predictions.

In short, Anonymous Walk Embedding include two main steps:

1. Sample the anonymous walks
2. Train the embedding of the graph and embedding of anonymous nodes simultaneously using the architecture in the following figure

Overall Architecture



Summary

In this file, we cover node/graph representation learning, a way to learn node and graph embeddings for downstream tasks, without feature engineering. The learned embeddings can be used in several tasks such as clustering/community detection, node classification, link prediction, graph classification and so on.

Node embedding can be learned through random walk. DeepWalk and Node2Vec are two representing algorithms which use unbiased and biases random walk respectively.

Graph embedding can be extended by node embedding aggregation. In addition, anonymous walk embedding is also an effective representation method.