On Generalizing Neural Node Embedding Methods to Multi-Network Problems

Mark Heimann
University of Michigan
mheimann@umich.edu

Danai Koutra
University of Michigan
dkoutra@umich.edu

ABSTRACT

Representation learning has attracted significant interest in the community and has been shown to be successful in tasks involving one graph, such as link prediction and node classification. In this paper, we conduct an empirical study of two leading deep learning based node embedding methods, node2vec and SDNE, to examine their suitability for problems that involve *multiple* graphs. Although they have been shown to preserve properties necessary for the success of canonical tasks on a single graph, we find that different runs of the same algorithm even on the same graph yield different embeddings. For node embedding methods to apply to multi-graph problems, we note that this finding motivates additional work in learning how to embed different graphs similarly.

KEYWORDS

representation learning, graph alignment, node embedding

ACM Reference format:
Mark Heimann and Danai Koutra. 2017. On Generalizing Neural Node Embedding Methods to Multi-Network Problems. In *Proceedings of Mining and Learning with Graphs Workshop, Halifax, Nova Scotia Canada, August 2017 (MLG'17)*, 4 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Inspired by the success of deep learning, several methods have been proposed for learning representations of nodes in graphs that can be used in downstream prediction, clustering, and visualization tasks. Examples include node2vec [6], DeepWalk [10], LINE [12], SDNE [13], and others. These methods learn embeddings of the nodes in a graph in a low-dimensional latent space that aim to preserve similarities between similar nodes. They have gained popularity for graph mining, as the representations they learn have been shown to lead to state-of-the-art performance in many canonical tasks such as link prediction and multi-label classification.

Many important graph mining tasks, however, involve more than just a single graph. Some problems such as network alignment [1] or graph similarity [7] are inherently defined in terms of multiple graphs. In other cases, it is desirable to perform analysis across a collection of multiple graphs, such as the brain graphs of several MRI patients [4] or snapshots of a time-evolving graph [11].

Representations of all the nodes in the various graphs could be learned by running any of the above mentioned algorithms on each of the individual graphs, but to facilitate effective cross-graph analysis, the embeddings learned from different runs of the algorithm would have to be comparable. In particular, it would be desirable for similar nodes in different graphs, or at least different runs of the algorithm, to be embedded similarly.

We investigate the meta-consistency of the embeddings produced by state-of-the-art node representation learning methods: the extent to which they remain similar from run to run of the algorithm. Our analysis suggests that while similarities within a graph may be preserved, as was shown in [6, 13], similarities across runs of the algorithm may not be so neatly preserved. Thus, we contend that current methods are limited in their applicability to multiple graph problems, and call for work to address the problem of learning embeddings that are forced to align more closely.

2 NODE EMBEDDING METHODS: OVERVIEW

Intuitively, most deep learning inspired node embedding methods learn embeddings by maximizing an objective function that is designed to preserve various similarities in the graph. That is, similar nodes (such as neighbors, or nodes with several common neighbors) should have more similar embeddings.

node2vec. For example, node2vec has the following objective function:

$$\max_f \sum_{u \in V} \left[-\log \sum_{v \in V} \exp(f(u) \cdot f(v)) + \sum_{n_i \in N_s(u)} f(n_i) \cdot f(u) \right] \quad (1)$$

Here V is the vertex set of graph, and f the representations. In particular, $f(u)$, the feature representation for a node u is a d-dimensional vector, where d may be chosen by the user. This objective function seeks to learn these representations by maximizing the likelihood of preserving node neighborhoods [6].

When node2vec learns a representation for a node u, it first samples a neighborhood of nodes via a random walk starting from u. (The sampling procedure is flexible, with users able to govern some of its properties by setting hyperparameters p and q that bias the random walk; if the random walk is left unbiased by setting $p = q = 1$, then node2vec is equivalent to the related method DeepWalk [10].) Nodes sampled are considered "in context" and thus similar to u, and the second term in the bracketed outer sum is maximized when they are embedded close to u. On the other hand, other nodes in the graph are likely much less similar to u, and thus the first term in the sum, which is negative, is maximized when their embeddings are less similar to that of u. For efficiency's sake, this term is often approximated in practice with negative sampling, where it is computed from samples of other nodes from the graph not in the neighborhood $N_s(u)$ but not from all the nodes in V [6].

Within the context of a single graph, this sampling procedure was empirically shown to work well [6] (although many theoretical

MLG'17, August 2017, Halifax, Nova Scotia Canada

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

guarantees for the general skip-gram procedure with negative sampling remain unknown–cf. [8].) However, the procedure inherently ties the results of the algorithm to quantities specific to the run of the algorithm on that graph (the specific IDs of nodes sampled in the random walk).

The dependency of node2vec and its predecessor DeepWalk on nodes sampled by ID in a random walk further complicates things for problems such as graph alignment, where the standard assumption is that the graphs are permuted and nodes that should align do not necessarily have the same ID. To be specific, suppose there are two graphs with respective vertex sets A and B, where A_i is the node in A with ID i. Perhaps A_1 is a neighbor of, say, A_{47}, but because the IDs are different, this does not transfer to graph 2. It is very likely that B_{47} is not a neighbor of B_1, and thus the idea that B_{47} should be in the context of B_1 as its neighbor or that their embeddings should be similar does not carry over. We thus note that node embedding methods that use random walks to sample contexts may struggle with generalizing to multi-graph problems, at least in their current form.

SDNE. Another state-of-the-art node embedding method, Structural Deep Network Embedding (SDNE), takes a different approach to preserving similarities that does not depend on a potentially fragile context sampling procedure. SDNE learns features with an autoencoder trying to reconstruct the adjacency matrix $\mathbf{S}$ of the n-node graph G. Its loss function is designed to preserve node similarities based on a combination of first-order proximities (direct connections between nodes in the graph) and second-order proximities (nodes that have several mutual neighbors).

The first-order loss is given by

$$\sum_{i,j=1}^{n} \mathbf{S}_{ij} ||f_i - f_j||_2^2 \tag{2}$$

Here S is the adjacency matrix. If $\mathbf{S}_{ij}$ is nonzero (the nodes i and j are connected), then the this term is minimized the closer the embeddings of nodes i and j are. This constraint is reminiscent of Laplacian eigenmaps, a spectral technique for node embedding [2].

The second-order constraint is imposed by the reconstruction error of the autoencoder. Specifically, according to [13], the reconstruction criterion can make nodes with similar neighborhood structures (that is, many similar neighbors) have similar embeddings. Furthermore, due to the sparsity of real-world networks, it is especially important that observed links (nonzero entries) be reconstructed accurately. Thus, the second-order criterion is

$$||(\hat{\mathbf{S}} - \mathbf{S}) \odot \mathbf{B}||_F^2 \tag{3}$$

where $\mathbf{S}$ is the adjacency matrix and $\hat{\mathbf{S}}$ the reconstruction learned by the autoencoder. $\mathbf{B}$ is a penalty matrix that consists of a user-specified term $\beta > 1$ for every nonzero entry in the adjacency matrix (zeros elsewhere), to further penalize failure to reconstruct observed links in the adjacency matrix.

Note that the node representations are the hidden representations at the last layer of the autoencoder, with the input (the adjacency matrix) being at the first layer, and after that the hidden representations being defined recursively as follows:

$$f_i^k = \sigma(\mathbf{W}^k f_i^{(k-1)} + \mathbf{b}^{(k)}), k = 2, \ldots, k \tag{4}$$

Here σ is a nonlinear activation function and the $\mathbf{W}$ and $\mathbf{b}$ terms are weights (to be learned) and biases respectively. The reconstruction $\hat{\mathbf{X}}$ is given by a decoder that reverses this encoding process.

The combined loss function that SDNE minimizes is a combination of the first-order and second-order loss (Equations 2 and 3 respectively), along with a regularization term on the weights learned by the autoencoder. (The user must set hyperparameters to determine how much emphasis to put on each component.) It is based on global structural properties depending on the adjacency matrix and not just a sampled neighborhood or specific node IDs. However, it is nonconvex and may not converge to the same solution every time, and little is proven about its consistency. Like node2vec, SDNE function tries to enforce consistency within a run of the algorithm on the graph, but does not by any means guarantee consistency across graphs or runs of the algorithm.

In this work, we focus on node2vec and SDNE. Among deep learning inspired node embedding methods, these two represent arguably the two main categories of node embedding methods: ones that explicitly sample contexts via random walks to use as contextual information (node2vec, DeepWalk [6, 10]), and ones that look to structural properties of the adjacency matrix to preserve first- and second-order similarities (SDNE, LINE [12, 13]). They were shown to outperform competing techniques [6, 13], with a recent empirical survey of node embedding methods [5] further finding that SDNE outperformed other methods, including node2vec, on a variety of canonical tasks.

The success of the above methods lies in the effectiveness of their loss functions in encouraging similar nodes *within* a graph to have similar embeddings [6, 13]. If this were also true *across* graphs (the best case being running the methods on two identical graphs), then the learned representations would be useful for identifying similarities between nodes in different graphs. The loss functions are tailored to a single graph, of course, but if methods such as node2vec or SDNE learned consistent embeddings each time they were run, then assuming the graphs are similar, the embeddings learned for the corresponding nodes would still be similar. The question then becomes: How consistent in fact are these methods?

3 EXPERIMENTAL ANALYSIS

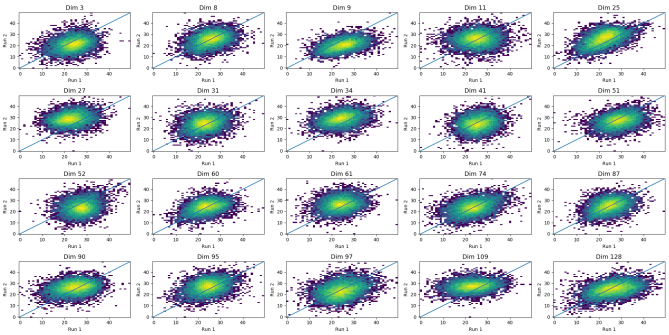
In this section, we study how the performance of the two node embedding methods changes when run different times. Even when the graphs are hardly different (simple permutations of each other), or not even different at all, we observe significant changes.

Data. For our experiments, we run the two network embedding methods on each of the following networks, which have both been used for evaluating node embedding methods [6] and yield consistent results.

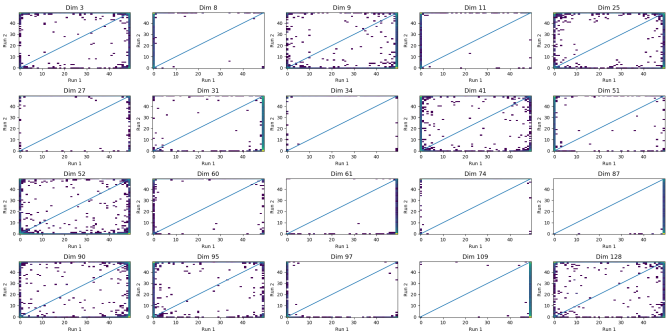
Table 1: Real Data

Name	Nodes	Edges	Description
BlogCatalog [14]	10 312	333 983	blog social network
PPI [3]	3 890	76 584	*Homo Sapiens* protein-protein interaction

Experimental setup. For both methods, we used mostly the default settings, since we are not trying to optimize predictive accuracy on a task but merely visualize standard results of the algorithms. For node2vec, we set p and $q = 1$, leaving the random walk unbiased (this amounts to performing DeepWalk, which suffices to



(a) node2vec (dimensions from top left: 3, 8, 9, 11, 25, 27, 31, 34, 41, 51, 52, 60, 61, 74, 87, 90, 95, 97, 109, 128)



(b) SDNE (dimensions from top left: 3, 8, 9, 11, 25, 27, 31, 34, 41, 51, 52, 60, 61, 74, 87, 90, 95, 97, 109, 128)

Figure 1: BlogCatalog: Comparison of embeddings—dimension by dimension (for 20 randomly chosen dimensions)—from two runs of (a) node2vec and (b) SDNE. x axis: node embedding value in run 1; y axis: node embedding value in run 2. The $y = x$ is shown in each plot.

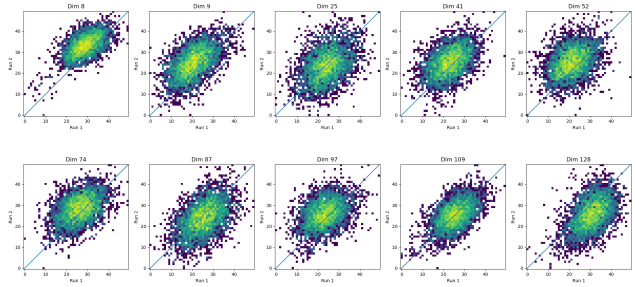
get a flavor for what the embeddings might look like). We performed 10 walks of length 80 per source node, and set a neighborhood size of 10. For SDNE, we set the first hidden layer of the autoencoder to have 1000 nodes and the second, the output to have 128 (the dimensionality of the features). We chose β, the reconstruction penalty for nonzero elements to be 15, and put weights of 1 on the first-order loss and 5 on the second-order loss.

Per dimension comparison. For both node2vec and SDNE, we run the algorithm twice to learn two embeddings of the nodes with two different runs of the algorithm. For each dimension, we plot the nodes with their value in the first embedding on the x axis and their value in the second embedding on the y axis. If the embeddings perfectly corresponded–the feature values are the same in both embeddings–we should see the points falling on a straight line $y = x$.

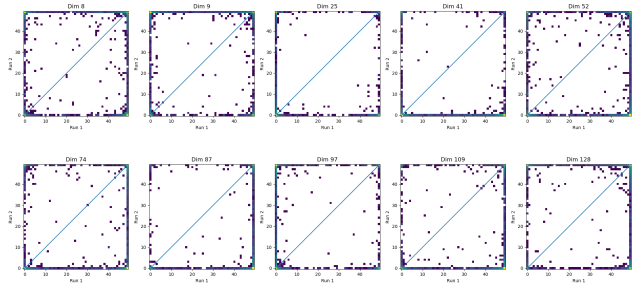
Instead, in Figure 1, we visualize 100 nodes from the BlogCatalog dataset along 20 randomly chosen dimensions and see that the correspondence is not nearly so clear. For SDNE in particular, we note that many points that have similar values along a given dimension in one embedding have quite different values along the same dimension in the other embedding. Even node2vec, however, demonstrates a noisy at best correspondence between values of the same feature in different embeddings. Indeed, we note that there is no reason that latent feature i in one embedding should correspond to latent feature i in another embedding. We observe similar results for the PPI dataset in Figure 2. We visualize only 10 randomly chosen dimensions, as the others are similar.

The differences between embeddings are also observable on a macroscopic level, which we illustrate by visualizing the embeddings of the BlogCatalog dataset. We run each method with the same settings twice on the original graph. Additionally, we run the method on the simplest possible experiment involving a different network: a noiseless permutation of the graph, where the structure is not changed (the permuted graph is isomorphic to the original one) and only the node IDs are randomly shuffled.

Low-dimensional embedding comparison. We learn a 128-dimensional representation and, following the protocol of [13], use



(a) node2vec (dimensions from top left: 8, 9, 25, 41, 52, 74, 87, 97, 109, 128)



(b) SDNE (dimensions from top left: 8, 9, 25, 41, 52, 74, 87, 97, 109, 128)

Figure 2: PPI: Comparison of embeddings—dimension by dimension (for 10 randomly chosen dimensions)—from two runs of (a) node2vec and (b) SDNE. x axis: node embedding value in run 1; y axis: node embedding value in run 2. The $y = x$ is shown in each plot.

t-SNE [9] to further learn a low-dimensional embedding that allows us to visualize the embeddings in 2-dimensional space. To further simplify the figure, we subsample the same 500 nodes at random to plot for each figure. The results for node2vec are shown in Figure 3, and those for SDNE are shown in Figure 4.

Looking at the embeddings as a whole, we see that the general shapes are visibly different. Qualitatively, we note that SDNE's embeddings are less ragged than node2vec's, which may reflect its