Graph Embedding Algorithms

Rahul Singh
Khoury College of Computer and Information Science, Northeastern University
singh.rahu@northeastern.edu

December 16, 2020

1 Objectives and Significance

1.1 Importance and Project Motivation:

Network structures are quickly becoming critical in the way that information is being transmitted, stored and encoded on the Internet. Graph embedding is a topic of increasing importance, since it allows us to frame questions about graphs as machine learning tasks, such as node classification, community detection, and edge detection. A graph embedding is a function that takes in a graph and projects each node into a lower-dimensional space such that some proximity measure is preserved. Being able to translate nontraditional data structures like networks and graphs to more interpretable and meaningful representations like low-dimensional embeddings is important for tasks such as data visualization, ensemble learning, link prediction, community detection, node classification clustering, and network compression. The hope is, as with all embeddings, that the vector representations of each node captures the structure of the graph in a lower-dimensional space.

1.2 Goal of the project:

Over the course of this project, I initially aimed to accomplish 2 things. First, I wanted to re-implement the LINE architecture using a combination of first and second order proximity objectives, train it on the BlogCatalog, Flickr and YouTube networks, and evaluate it against another type of embedding generating algorithm ie node2vec. However, due to various constraints that presented themselves, I had to relax and/or change some of the procedures I carried out. My final objective ended up being to implement and evaluate the above mentioned graph embedding techniques against each other, with various networks having different qualities. Then I ran the produced embeddings through a logistic regression classifier in order to classify vertices against their ground truth labels, and evaluate them based on their performance. I plan on using libraries like TensorFlow, networkx, sklearn, numpy, etc. in order to re-implement these methods based

on their original papers. I am rebuilding these algorithms from scratch, but using some high-level APIs like gensim's Word2Vec to speed up expensive parts of the process. I aimed to evaluate our models using micro-F1 scores, among other accuracy measures, and test our claims that the node2vec architecture will produce better embeddings by doing visualizations of the produced embeddings compared to the others. In doing this project, my results correlated relatively well with those found empirically in the original node2vec paper [2] which outperforms LINE in most scenarios, on the task of vertex classification.

2 Background

2.1 Important concepts and background information:

Graphs, such as social networks, word co-occurrence networks, and communication networks, occur naturally in various real-world applications. Analyzing them yields insight into the structure of society, language, and different patterns of communication. Graph embedding approaches mainly face challenges in terms of what properties of the graph they preserve. A "good" vector representation of nodes should preserve the structure of the graph and the connection between individual nodes. The first challenge is choosing the property of the graph which the embedding should preserve. Given the plethora of distance metrics and properties defined for graphs, this choice can be difficult and the performance may depend on the application. One such important preserved property is homophily, which states that nodes that are more closely connected should be embedded closely together. Another such important preserved property is structural equivalence, which states that nodes that have similar structural roles in networks should be embedded closely together, which doesn't necessarily imply close connection. These kinds of tradeoffs are important to consider in the context of your application, and therefore, your choice of embedding function.

2.2 Algorithmic Taxonomy of Embedding Approaches

Palash et al. categorize the embedding methods into three broad categories:

1. Factorization Based Methods: Factorization based algorithms represent the connections between nodes in the form of a matrix and factorize this matrix to obtain the embedding. The matrices used to represent the connections include node adjacency matrix, Laplacian matrix, node transition probability matrix, and Katz similarity matrix, among others. Approaches to factorize the representative matrix vary based on the matrix properties. If the obtained matrix is positive semidefinite, e.g. the Laplacian matrix, one can use eigendecomposition. For unstructured matrices, one can use gradient descent methods to obtain the embedding in linear time. Example: LINE

2. Random Walk Based Methods:

   Random walks have been used to approximate many properties in the graph including node centrality and similarity. They are especially useful when one can either only partially observe the graph, or the

graph is too large to measure in its entirety, and can be parallelized to create large corpora of walks effortlessly. Example: Node2Vec

3. Deep Learning Methods:

   The growing research on deep learning has led to a deluge of deep neural networks based methods applied to graphs. Deep autoencoders have been used for dimensionality reduction due to their ability to model non-linear structure in the data.

2.3 node2vec

Node2vec is a random-walk based embedding technique that is used to learn continuous representations of nodes in a network, that uses novel sampling methods to avoid some well-known issues in graph sampling. The novelty that node2vec brings is its continuous flexibility with representing "local" communities. Through the use of two parameters, an in-out parameter and a return parameter (p and q, respectively), the model is able to model either more BFS-like (local, homophily-preserving, low p-value) behavior, or more DFS-like (nonlocal, structural equivalence-preserving, low q-value) behavior. Balancing these parameters can help random walks alleviate issues like 2-hop similarity (repeated visiting of the source node in a graph triangle) and can encourage better exploration of a graph in more nuanced ways.

2.4 LINE: Large-scale Information Network Embedding

LINE is a method of learning graph embeddings at unprecedented scales. Graph embeddings mainly try to preserve two qualities: first-order (local pairwise proximity between two vertices) and second-order proximity (similarity between the neighborhood network structure of two vertices). Previous work in creating graph embeddings suffered from not accounting for the global structure of the network, or not providing a specific objective that directly optimized first and second-order proximity, therefore creating sub-optimal embeddings for given networks. Moreover, previous graph embedding models suffered at scale because of lack of second-order information, which came from inefficiency with dealing with huge networks. It is suitable for a variety of networks including directed, undirected, binary or weighted edges. Its goal is to minimize the difference between the input and embedding distributions (nodes in the graph), which is usually achieved using KL divergence. It defines two joint probability distributions for each pair of nodes, and then then minimizes the KL divergence of the distributions.

3 Methods

3.1 Constraints

As mentioned earlier in the background, I ran into issues mostly related to scaling and compute availability . In terms of datasets, I quickly realized that some of our datasets (YouTube, Flickr), were prohibitively

large. In order to combat this issue, I retrieved more data sets from the Stanford SNAP network repository, and wrote batch jobs to train my models on the Discovery cluster, which although it couldn't process even our biggest networks, helped spread the processing load.

3.2 Choice of data

I decided to use a varied set of networks, in order to test out various aspects of our algorithms. The YouTube, Flickr and BlogCatalog datasets were obtained from the LINE paper [1]. The wiki and homosapiens datasets were obtained from the Stanford SNAP repository [2]. In the below chart we can see the average degrees of the different datasets and their number of nodes and edges.

Dataset	Description	Average Degree	Number of Nodes	Number of Edges
BlogCatalog	The nodes are accounts, the links are social connections, and the categories are topic categories	64.8	10312	333983
Wiki	Is a word co-occurrence network, meaning that nodes are words and links are when they are used together.	38.7	4777	92406
Homosapiens	Is a protein to protein interaction network of annotated gene sets, created for the node2vec paper [2]	19.7	3890	38292
Flickr	This is a dataset from a popular image sharing site.	146.6	80513	5899882
YouTube	Is a popular video sharing website, the labels in this dataset represent users with common video topic interests	5.25	1138499	2990443

Table 1: A table containing useful information about the datasets and their respective sizes and complexity

We can see that the densest graph is Flickr with a degree more than double that of any of the other datasets. While YouTube is the largest dataset, it also has the smallest average degree, having more than 10 times the nodes of Flickr but with nearly half the edges. These datasets give us a large variability in both size, sparsity and domain which enables us to have a robust platform on which to test and evaluate LINE and node2vec.

3.3 Overall methodology:

Before I started each specific algorithm, I created my own internal representation of a graph and its vertex embeddings that I could use in order to maintain interface parity between my models, which allowed me to work on evaluation and visualization code asynchronously. This allowed me one central place where I could load the datasets, retrieve train/test splits for them for classification, etc. The data initially comes in compressed sparse matrix format, within a matlab file, that my representation then parses into a standard adjacency list format. From then on out, each method was built out into its own class, so it could encapsulate all the state it needed to generate its own embeddings. Once a method produced embeddings, they were fed through logistic regression training and test sets to generate models and metrics.

3.3.1 LINE

The LINE algorithm creates two types of embeddings, ones which preserve the first order proximity between nodes and second preserving the second order proximity. The first order proximity is preserved by minimizing the KL divergence between two probability distributions. This distribution is calculated between each pair

of vertices in the edge list of the input graph by the following formula (Given two vertices v_i and v_j):

$$p_1(v_i, v_j) = \frac{1}{1 + \exp(-\vec{u}_i^T . \vec{u}_i)}$$

The second order proximity assumes that vertices sharing many connections to other vertices are similar to each other, thus each vertex which has a similar neighborhood to an another vertex should have embeddings closer to one another. Thus we check the similarities between the neighbourhood structures of two vertices. Given two vertices v_i and v_j and $\vec{u}_i$ is the representation of v_i when it is treated as a vertex while $\vec{u}_i^{'}$ is the representation of v_i when it is treated as a specific context [neighbourhood structure] and $|V|$ is the number of vertices or contexts. The probability distribution is calculated as follows:

$$p_2(v_j|v_i) = \frac{\exp(\vec{u}_j^{'T} . \vec{u}_i)}{\sum_{k=1}^{|V|} \exp(\vec{u}_k^{'T} . \vec{u}_i)}$$

Similar to the first order proximity, the second order proximity is also preserved by minimizing the KL divergence between two probability distributions. Now that we have an understanding of the theoretical underpinnings of LINE, we can discuss how it is implemented in our code. I use TensorFlow's Keras library to generate initial embeddings (which are just integer number to vector conversion). Then layers of first and second order embeddings are created. This represents our model. I run this model for multiple epochs using KL divergence for minimization and Adam as the optimizer. In order to speed up the process, Alias sampling is implemented and a mini-batch of 1024 nodes is selected as default. This algorithm iterates over the edges to calculate the probability distributions, hence it was much slower for Flickr dataset (dense), whereas for YouTube it was able to run 4 epochs, even though the YouTube dataset was much sparser than Flickr, its overall edges were still too large for our machines to build those embeddings.

```
def LINE (graph, embedding_size, negative_ratio, proximity_order):

    convert vertex index to embeddings [number to vector]
    create model using embeddings and adam optimizer with line loss function

    for i in range(epoch):

        create batches based on batch_size
        Train Model to calculate first and second order embeddings
        Use Alias Sampling for speedup
        store embeddings for next iter

    return embeddings
```

Figure 1: LINE Pseudocode

3.3.2 node2vec

As mentioned earlier, node2vec utilizes two parameters, p and q, to bias towards BFS and DFS-like random walks respectively.

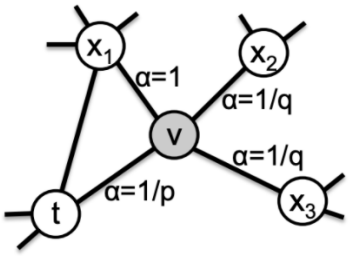


Figure 2: How the in-out and return parameters affect random walks

In this sample, where we've moved from node t to vertex v, we see how the parameters p and q affect the probability distribution along the choice of which neighbor to move onto in this biased random walk. Given that we now have a non-uniform distribution that we now need to sample from each time we want to advance in each walking step, we divide our algorithm into 3 parts. First, we precompute the transition probabilities for each node and edge in our graph, utilizing Vose's alias sampling algorithm to sample neighbors based on certain configurations of p and q. Second, with our probabilities built up, we can perform biased random walks over the graph in constant time. Third, after building up these walk corpora, we can use it as input to a Skip-gram model, and have it produce embeddings for us. Overall, this process can be parallelized relatively easily, however, the initial step of computing transition probabilities for every node and edge ended up being very expensive, prohibiting me from running it on any larger datasets like Flickr or YouTube.

3.4 Evaluation tasks, metrics and strategy

3.4.1 Vertex classification

Often in networks, a fraction of nodes are labeled. In social networks, labels may indicate interests, beliefs, or demographics. In language networks, a document may be labeled with topics or keywords, whereas the labels of entities in biology networks may be based on functionality. Due to various factors, labels may be unknown for large fractions of nodes. For example, in social networks many users do not provide their demographic information due to privacy concerns. Missing labels can be inferred using the labeled nodes and the links in the network. The task of predicting these missing labels is also known as vertex classification. Feature-based models generate features for nodes based on their neighborhood and local network statistics and then apply a classifier like logistic regression and naive Bayes to predict the labels. Embeddings can be interpreted as automatically extracted node features based on network structure. Predicting node labels using network topology is widely popular in network analysis and has a variety of applications, including document classification and interest prediction. A good network embedding should capture the network

structure and hence be useful for vertex classification. I compare the effectiveness of embedding methods on this task by using the generated embedding as node features to classify the nodes.

3.4.2 Evaluation metrics and strategy

There are five main metrics that I used for evaluating our models: micro-F1, macro-F1, accuracy, precision and recall. Let's define Y as the set of true labels, and $h(x)$ as the predicted label. Now we can define precision for multilabel as $\frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i \cap h(x_i)|}{|h(x_i)|}$, which is the ratio of how often the predicted value is correct. Recall is defined as $\frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i \cap h(x_i)|}{|Y_i|}$, which is the ratio of how many of the labels were predicted correct. Accuracy is defined as $\frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i \cap h(x_i)|}{|Y_i \cap h(x_i)|}$ [9]. Macro-F1 is $\frac{(\beta^2+1)precision \quad recall}{\beta^2 precision \quad recall}$ Micro-F1 is $2\frac{precision \quad recall}{precision \quad + \quad recall}$. I used the sklearn package to compute these metrics on our datasets and evaluations.

Each dataset is run on LINE and node2vec to generate embeddings. These embeddings are then trained using one vs rest logistic regression on test sets consisting of [5, 10, 20, 30, 40, 50, 60, 70, 80, 90] % of the data and the rest as validation sets. Using the validation I then generate our metrics mentioned above. I then can run this process 5 times for each of the different dataset and graph embedding combinations, to generate averages and standard deviations for the metrics.

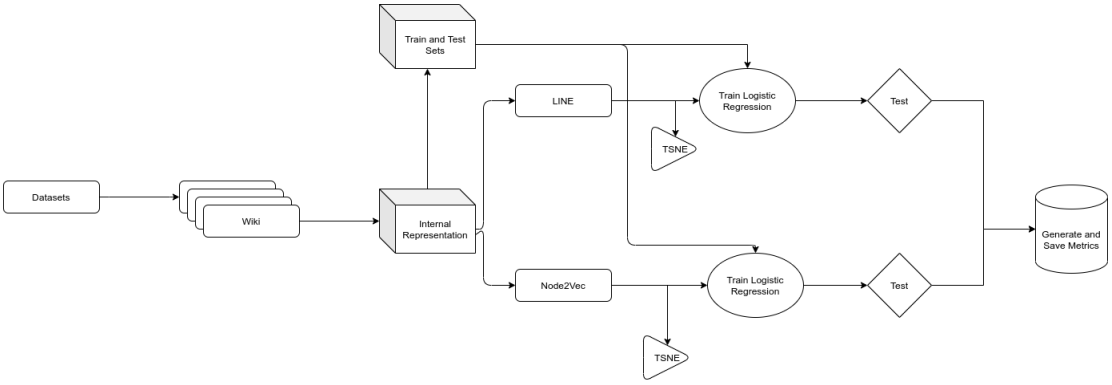


Figure 3: A visualization of the workflow that the code goes through to produce embedding and metrics.

In order to do our evaluation, I need to run all of the methods on all of our datasets for many different configurations. To do this, I first used my local machine to find what seemed to be the best hyperparameters, also basing these values off of the values used in the papers. I then moved my computations over to the Northeastern University Discovery cluster, where I had access to machines with enough RAM and CPU to run the larger datasets. Here I ran the methods using machines with 24 CPUs and between 128 and 192 GB of RAM. Then after producing all of our metrics I used a script to aggregate them and produce the averages, standard deviations, and graphs necessary for this paper.

4 Results

4.1 Hyperparameter Choice

After using a local grid search as well as basing initial values off of [1] and [2], these are the hyperparameters that I chose to use for our experiments: **LINE**: embedding dimension: 128, proximity order: 1st + 2nd, batch size: 1024, epochs: 10, negative ratio: 5, model optimizer: Adam, loss function: KL divergence, sampling technique: alias. **node2vec**: embedding dimension: 128, in-out parameter: 0.25, return parameter: 0.25, walk length: 30, window size: 10, sampling technique: alias.

4.2 Results and Findings

4.2.1 BlogCatalog

Metric	Algorithm	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Precision	LINE	22.92	24.39	24.57	25.74	26.45	27.22	27.48	28.77	28.93	31.33
	Node2Vec	28.86	29.79	30.84	32.33	32.19	33.22	34.1	35.15	35.97	37.41
Recall	LINE	21.67	24.42	26.15	27.83	28.49	29.72	30.21	30.96	31.23	31.88
	Node2Vec	26.63	29.1	31.92	33.45	34.2	35.24	35.75	36.71	37.07	37.03
Micro-F1	LINE	21.67	24.42	26.15	27.83	28.49	29.72	30.21	30.96	31.23	31.88
	Node2Vec	26.63	29.1	31.92	33.45	34.2	35.24	35.75	36.71	37.07	37.03
Macro-F1	LINE	10.25	12.45	14.18	15.86	16.34	17.55	17.94	18.19	18.7	19.43
	Node2Vec	13.27	16.4	19.06	21.02	21.76	22.99	23.49	24.7	24.48	24.36

Table 2: Results of multi-label classification in percents on the BlogCatalog dataset

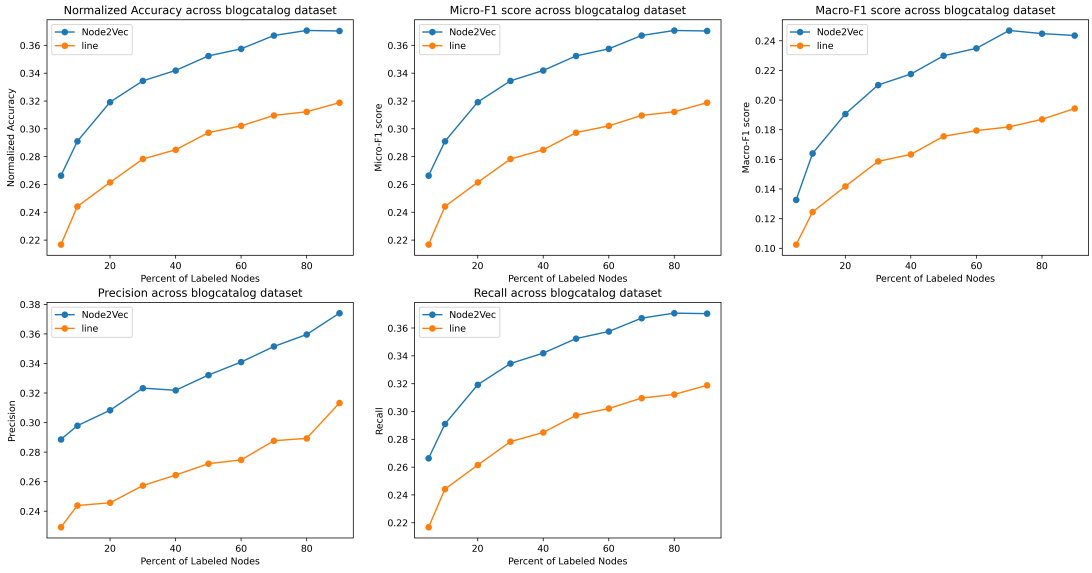


Figure 4: Micro-F1, macro-F1, precision, recall and normalized accuracy of node classification for BlogCatalog data sets varying the percentage of labeled nodes used for training.

Node2vec performed on average 5-6% better for all the metrics, across the major percentage range of labels selected, only varying between the initial 5-20 and 80-90% mark. Even though my hyperparameters were slightly different than the original papers, my results were on average 10% lower than the original paper (across the entire percent range).

4.2.2 Flickr

Metric	Algorithm	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Precision	LINE	22.71	22.42	22.73	23.7	24.38	24.96	25.7	26.05	27.37	28.76
Recall	LINE	24.04	25.44	27.26	28.48	29.18	29.85	30.41	30.58	31.07	31.25
Micro-F1	LINE	24.04	25.44	27.26	28.48	29.18	29.85	30.41	30.58	31.07	31.25
Macro-F1	LINE	9.13	11.28	13.29	14.7	15.35	15.89	16.19	16.41	16.79	16.66

Table 3: Results of multi-label classification in percents on the Flickr dataset

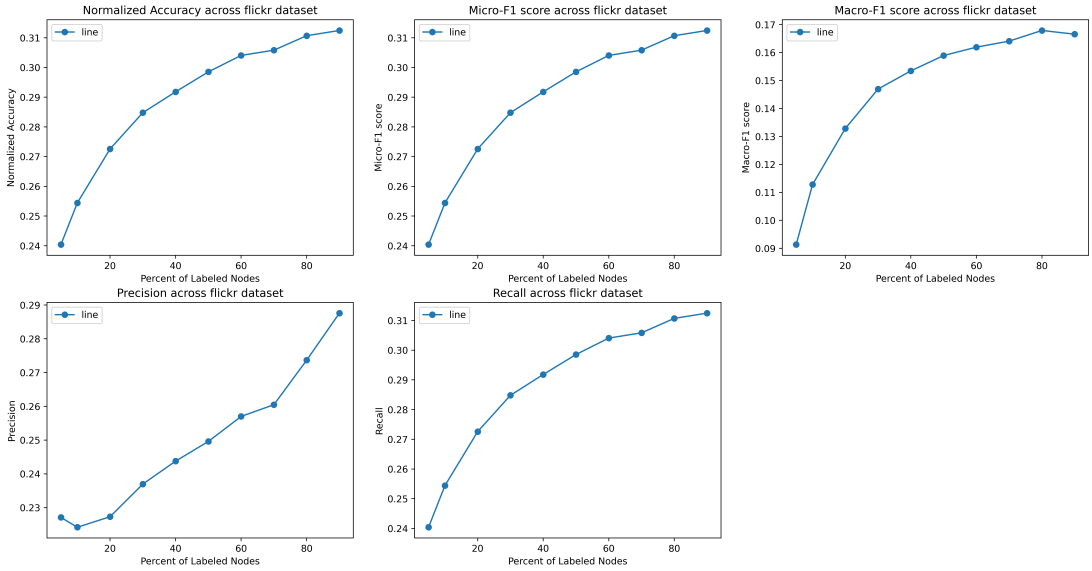


Figure 5: Micro-F1, macro-F1, precision, recall and normalized accuracy of node classification for Flickr data sets varying the percentage of labeled nodes used for training.

Node2vec was unable to run on this dataset (even for a single run over a 24 hour period). The claims from the LINE paper of having an average micro-F1 and macro-F1 scores of $\approx 64\%$ were unable to be replicated and on average, a 27% micro-F1 and only 14% macro-F1 score was achieved.

4.2.3 Wikipedia

Metric	Algorithm	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Precision	LINE	31.59	30.17	30.62	31.73	32.65	33.03	33.38	34.89	34.8	35.19
	Node2Vec	26.59	26.96	29.97	31.03	31.87	32.05	32.77	33.72	33.65	35.43
Recall	LINE	29.46	29.44	30.29	31.32	32.29	32.78	32.62	33.67	35.06	33.72
	Node2Vec	25.75	27.18	29.73	31.23	32.22	32.72	33.56	34.37	34.4	34.33
Micro-F1	LINE	29.46	29.44	30.29	31.32	32.29	32.78	32.62	33.67	35.06	33.72
	Node2Vec	25.75	27.18	29.73	31.23	32.22	32.72	33.56	34.37	34.4	34.33
Macro-F1	LINE	3.63	4.4	5.28	5.48	6.1	6.83	7.59	8.48	9.19	10.94
	Node2Vec	3.2	3.86	5.11	5.6	6.05	6.5	7.04	8.1	9.15	11.81

Table 4: Results of multi-label classification in percents on the Wikipedia dataset

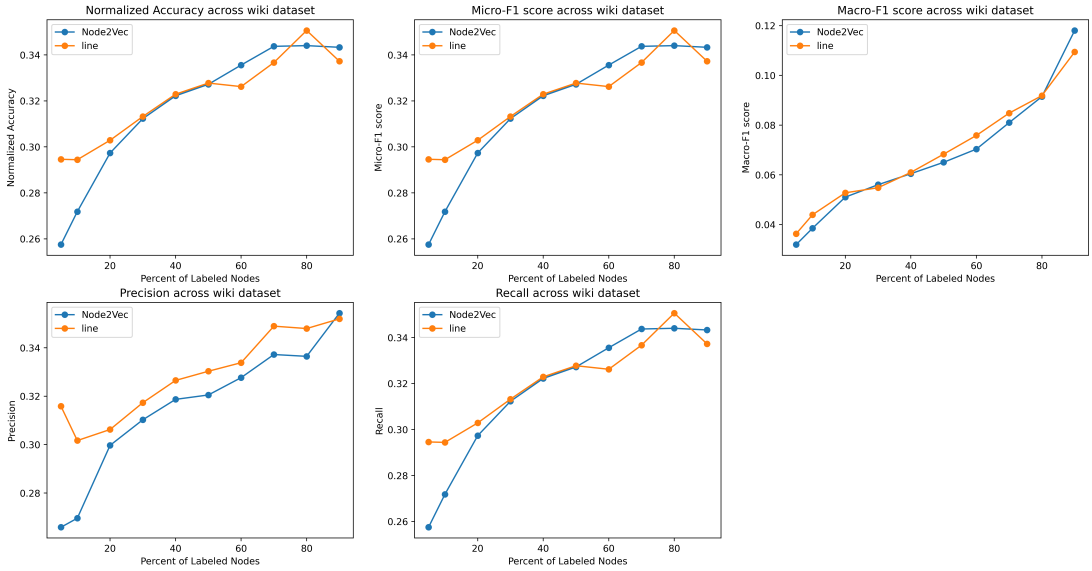


Figure 6: Micro-F1, macro-F1, precision, recall and normalized accuracy of node classification for Wikipedia data sets varying the percentage of labeled nodes used for training.

LINE and node2vec had similar performance except on the lower percent range where LINE had a slight advantage and upper percent range where node2vec did (over each other). These results were consistent across all the metrics tracked except precision where LINE had a consistent 0.5% advantage over the entire percentage range.

4.2.4 Homosapiens

Metric	Algorithm	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Precision	LINE	30.39	26.59	19.1	17.18	18.95	18.65	18.54	19.87	22.07	22.29
	Node2Vec	16.61	13.93	13.55	13.5	13.83	14.31	14.85	14.85	15.1	19.73
Recall	LINE	8.99	10.85	12.39	13.97	13.98	14.11	15.12	15.34	17.2	15.73
	Node2Vec	9.77	11.42	12.93	14.12	14.89	15.51	16.3	16.08	15.82	17.74
Micro-F1	LINE	8.99	10.85	12.39	13.97	13.98	14.11	15.12	15.34	17.2	15.73
	Node2Vec	9.77	11.42	12.93	14.12	14.89	15.51	16.3	16.08	15.82	17.74
Macro-F1	LINE	4.23	5.25	6.51	8.11	8.41	8.61	9.41	9.96	10.5	9.96
	Node2Vec	5.12	6.69	8.15	8.99	9.82	10.35	10.73	10.44	10.26	12.38

Table 5: Results of multi-label classification in percents on the Homosapiens dataset

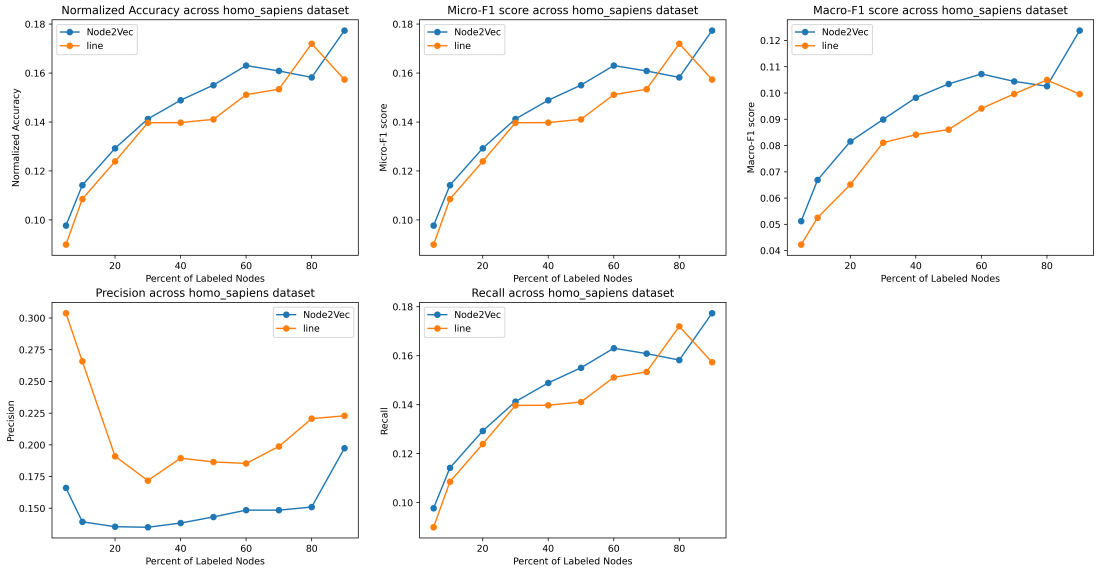


Figure 7: Micro-F1, macro-F1, precision, recall and normalized accuracy of node classification for Homosapiens data sets varying the percentage of labeled nodes used for training.

LINE consistently underperformed by a margin of 1-2% over the [40-70]% range of embeddings picked. This was true for macro-F1, micro-F1 and recall metrics but in terms of precision, LINE consistently outperformed the node2vec by a margin of 5-10% over the entire percentage range of embeddings picked.

4.3 Analysis

Overall, the results are in line with what I expected to see given the existing empirical evidence comparing the two methods from the original node2vec paper. Given that each of our datasets labels exhibited a good deal of homophily as well as structural equivalence, it was clear that no mix of any one objective would have been good enough to lead to meaningful embeddings.The initial values of 0.25 for both the in-out and return hyperparameters seemed to work incredibly well across most networks, given their sparsity and average degree. An analysis of how node2vec would perform on an incredibly large, dense network would have been really interesting, however the time/memory requirements did not allow for this to happen. Instead, as a proxy for a rather big network, I ran an analysis of this in-out parameter on the wiki dataset, which, from the literature [2], needed to encode a decent amount of 2nd and 1st order information in its embedding in order for it to perform well on node classification.

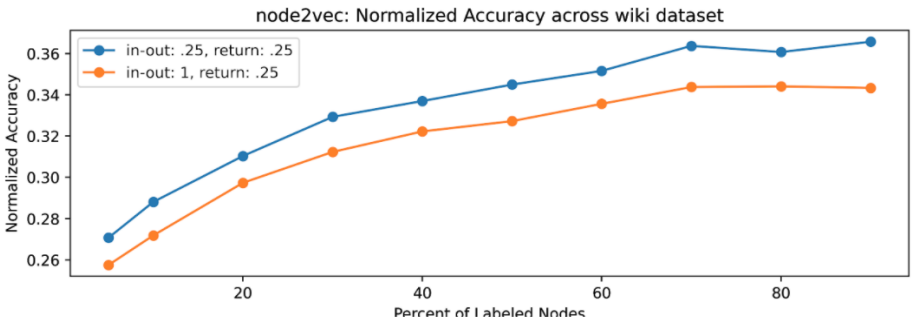
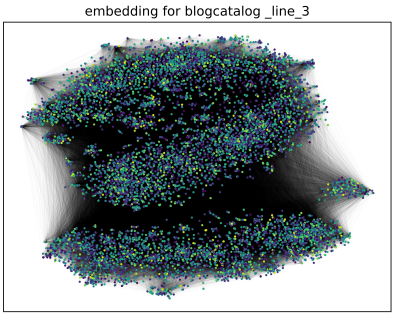


Figure 8: The effect of in-out and return parameters on normalized accuracy

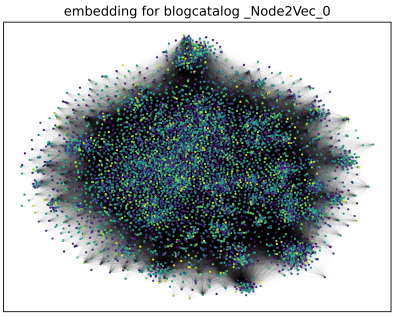
From this we can see that as the in-out parameter increases, we tend to get more local BFS-like behavior, which for dense graphs, can lead to a lot of walking within the same immediate neighborhood, hurting performance. It is this notion of tunable exploration that makes node2vec more flexible than LINE for this series of networks.

4.4 Embedding visualization

A common method used to visualize high-dimensional vectors in 2-D space is t-SNE (t-distributed stochastic neighbor embedding) [12]. I generated a t-SNE diagram for every run of an algorithm over a dataset. These graphs allow us to see visually how the embeddings that they produced look projected into 2D space. This provides some opportunity for comparison and analysis of the structure that the algorithms are able to glean from the graphs. I have included a few of the t-SNE diagrams from the runs, that show some of the general patterns that occurred in them. The LINE diagrams tended to look quite different from node2vec, but it was not uncommon to see some similar sections of the graph. From run to run within a dataset and algorithm pair the diagrams did have some variability, which is to be expected given the random nature of the algorithms but they often maintained the same general structure which lends confidence to the reusability and repeatability of the algorithms.

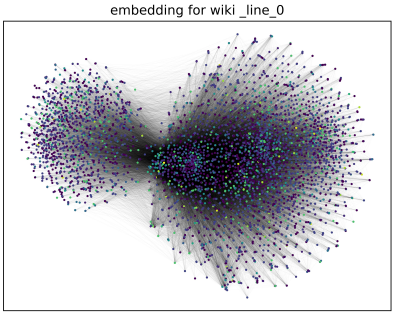


(a) t-SNE diagram for LINE on BlogCatalog

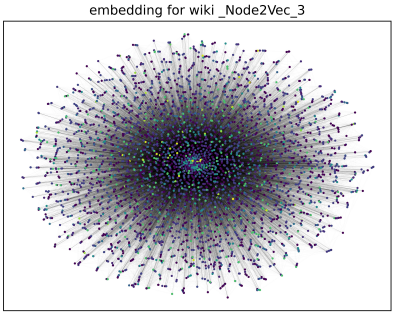


(b) t-SNE diagram for node2vec on BlogCatalog

Figure 9: Some of the TSNEs generated for the BlogCatalog dataset

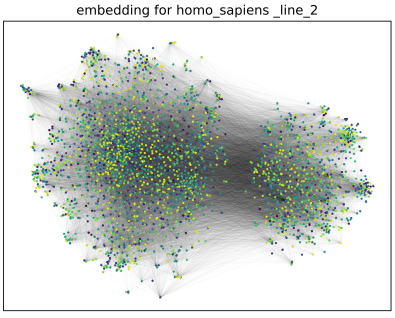


(a) t-SNE diagram for LINE on Wikipedia

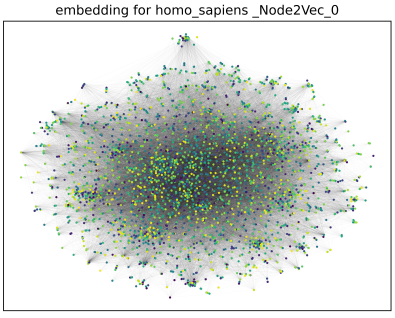


(b) t-SNE diagram for node2vec on Wikipedia

Figure 10: Some of the TSNEs generated for the Wikipedia dataset

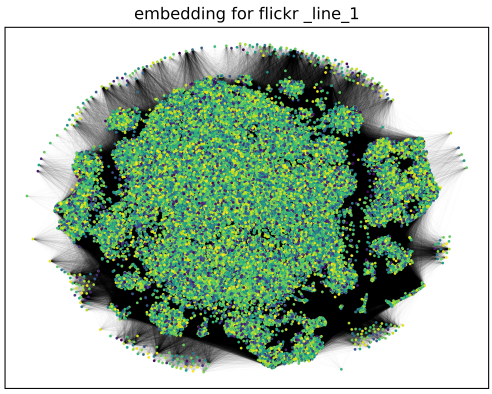


(a) t-SNE diagram for LINE on Homosapiens



(b) t-SNE diagram for node2vec on Homosapiens

Figure 11: Some of the TSNEs generated for the Homosapiens dataset



(a) t-SNE for LINE run on Flickr

Figure 12: t-SNE diagrams generated for the Flickr dataset

5 Conclusions

Overall I was able to come to some important conclusions. While I was unable to get LINE to work at the same high classification rate that was claimed in [1], I was able to replicate the results seen in [2]. LINE performed worse than expected in the experiments. But there is a rationale for this, LINE is unable to explore the graph in the same way that node2vec can. It merely focuses on near neighbors and as a result it cannot understand the greater structure of the graph especially in sparse networks. Also the only hyperparameters different from the original LINE paper were the changing learning rate, whereas I had a default learning rate of 0.001 for Adam optimizer. Plus the original paper used a mini-batch size of 1 for having stochastic gradient descent, but I chose a mini batch size of 1024 in order to speed up the evaluation. Note that basic evaluations of mini-batch size of 1 were used on smaller datasets, but they didn't show any improvements.

Given that node2vec has $O(|V|d)$ time complexity whereas LINE has $O(|E|d)$ time complexity, where d is the dimensionality of the embedding I choose, it is better to choose LINE if the graph has a sparse substructure. LINE can be used for tasks where a continuous regeneration of embeddings are required at the cost of reduced accuracy. Whereas for overall best node classification of the algorithms, node2vec should be chosen as it commonly performed better than the other.

6 Code

Code and the dataset are uploaded to github and access is provided to all users who have access to assignment: https://github.com/R911/NLP_Project

7 References

[1] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, & Q. Mei, (2015, March 12). LINE: Large-scale Information Network Embedding, Microsoft Research Asia. Retrieved from https://arxiv.org/pdf/1503.03578v1.pdf

[2] Grover, A., & Leskovec, J. (2016, August 01). Node2vec: Scalable Feature Learning for Networks. Retrieved from https://dl.acm.org/doi/abs/10.1145/2939672.2939754

[3] Tong F. (2019, May 6) Graph Embedding for Deep Learning. Towards Data Science. https://towardsdatascience.com/overview-of-deep-learning-on-graph-embeddings-4305c10ad4a4

[4] K. Henderson, B. Gallagher, T. Eliassi-Rad, H. Tong, S. Basu, L. Akoglu, D. Koutra, C. Faloutsos, and L. Li. RolX: structural role extraction & mining in large graphs. In KDD, 2012

[5] P. D. Hoff, A. E. Raftery, and M. S. Handcock. Latent space approaches to social network analysis. J. of the American Statistical Association, 2002.

[6] M. Zhang and Z. Zhou, "A Review on Multi-Label Learning Algorithms," in IEEE Transactions on Knowledge and Data Engineering, vol. 26, no. 8, pp. 1819-1837, Aug. 2014, doi: 10.1109/TKDE.2013.39.

[7] Palash Goyal and Emilio Ferrara, "Graph embedding techniques, applications, and performance: A survey", in Knowledge-Based Systems, 2018. Retrieved from https://arxiv.org/pdf/1705.02801.pdf

[8] Adams, Ryan, "The Alias Method: Efficient Sampling with Many Discrete Outcomes — Laboratory for Intelligent Probabilistic Systems.", 2013. Retrieved December 15, 2020, from https://lips.cs.princeton.edu/the-alias-method-efficient-sampling-with-many-discrete-outcomes/

[9] Wattenberg, M., Viégas, F., Johnson, I. (2016, October 13). How to Use t-SNE Effectively. Retrieved December 15, 2020, from https://distill.pub/2016/misread-tsne/