APRENDENDO REPRESENTAÇÕES EGOCÊNTRICAS PARA NÓS EM GRAFOS

TIAGO PIMENTEL MARTINS DA SILVA

APRENDENDO REPRESENTAÇÕES EGOCÊNTRICAS PARA NÓS EM GRAFOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Adriano Veloso
Coorientador: Nivio Ziviani

Belo Horizonte
Fevereiro de 2018

TIAGO PIMENTEL MARTINS DA SILVA

FAST NODE EMBEDDINGS: LEARNING EGO-CENTRIC REPRESENTATIONS

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Adriano Veloso
Co-Advisor: Nivio Ziviani

Belo Horizonte

February 2018

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Silva, Tiago Pimentel Martins da

S586f Fast node embeddings: learning ego-centric representations / Tiago Pimentel Martins da Silva / – Belo Horizonte, 2017.
xxvi, 44 f.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Adriano Alonso Veloso
Coorientador: Nívio Ziviani

1. Computação – Teses. 2. Aprendizado do computador. 3. Teoria dos grafos. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*82(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Fast Node Embeddings: Learning Ego-Centric Representations

TIAGO PIMENTEL MARTINS DA SILVA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Adriano Alonso Veloso - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Nivio Ziviani - Coorientador
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Fabricio Murai Ferreira
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Rodrigo Coelho Barros
Escola Politécnica - PUC/RS

Belo Horizonte, 26 de Fevereiro de 2018.

To my fiancée, who puts up with me.

Acknowledgments

I'd like to first thank my family, friends and fiancée, for all their support during my masters. All the colleagues I met at UFMG, with whom I discussed this whole project several times. And all the Professors who helped me through this journey, most of all Prof. Adriano Veloso and Nivio Ziviani, who advised me during this whole process. I'd also like to thank Kunumi for their financial and intellectual support during this work.

"There is no such thing as nothingness, and zero does not exist. Everything is something. Nothing is nothing. Man lives more by affirmation than by bread."

(Victor Hugo)

Resumo

O aprendizado de representações é um dos fundamentos de Deep Learning e permitiu avanços importantes em várias tarefas de Aprendizado de Máquina, como Tradução utilizando Redes Neurais, Respostas Automáticas para Perguntas e Reconhecimento de Fala. Trabalhos recentes propuseram novos métodos para aprender representações de nós e arestas em grafos. Vários desses métodos são baseados no algoritmo SkipGram. Eles processam um grande número de nós para produzir o contexto em que essas representações são aprendidas, com esses nós processados estando localizados numa vizinhança com distancia k de um nó raiz. Neste trabalho, é proposto um novo método, efetivo e eficiente, para gerar embeddings para nós em grafos, o qual utiliza um número restrito de permutações da vizinhança direta ($k = 1'$) de um nó como contexto para gerar seus embeddings, resultando em representações egocêntricas.

Este trabalho apresenta uma avaliação detalhada mostrando que o método proposto supera métodos do estado da arte em seis datasets diferentes relacionados com os problemas predição de arestas e classificação de nós, sendo uma a três ordens de magnitude mais rápido que os baselines. O novo algoritmo apresentado também é avaliado numa tarefa de desambiguação de nomes de autores, mostrando que embeddings de nós podem ser utilizados com sucesso nessa tarefa. Finalmente, uma avaliação em detalhes das representações aprendidas é feita sob a ótica da assortatividade apresentada por elas, sendo assortatividade a tendência de nós similares estarem conectados.

Palavras-chave: Embeddings para Nós, Grafos, Apredizado de Representações, Aprendizado de Máquina.

Abstract

Representation learning is one of the foundations of Deep Learning and allowed important improvements on several Machine Learning tasks, such as Neural Machine Translation, Question Answering and Speech Recognition. Recent works have proposed new methods for learning representations for nodes and edges in graphs. Several of these methods are based on the SkipGram algorithm, and they usually process a large number of multi-hop neighbors in order to produce the context from which node representations are learned. In this paper, we propose an effective and also efficient method for generating node embeddings in graphs that employs a restricted number of permutations over the immediate neighborhood of a node as context to generate its representation, thus ego-centric representations.

We present a thorough evaluation showing that our method outperforms state-of-the-art methods in six different datasets related to the problems of link prediction and node classification, being one to three orders of magnitude faster than baselines when generating node embeddings for very large graphs. We also evaluate our algorithm in an author name disambiguation task, showing that node embedding algorithms can be applied successfully to this problem. Finally, we further present an in-depth analysis of our algorithm in terms of the assortativity of the learned representations.

Palavras-chave: Node Embeddings, Graphs, Learning Representations, Machine Learning.

List of Figures

List of Tables

List of Symbols

Graphs

V	Vertices
E	Edges
b	Branching factor
v_i	Vertice i
$e_{i,j}$	Edge between nodes i and j
d	Embeddings size

SDNE

x_i	Adjacency vector of node i
y_i	Embedding of node i
$\hat{x_i}$	Reconstructed adjacency vector of node i
α	Hyper-parameter which controls the strength of the first order proximity term in the loss function
ν	Hyper-parameter which controls the strength of the regularization term in the loss function
β	Hyper-parameter which controls the priority given to existing links, versus non-existing ones, in the second order proximity loss function

SkipGram

S	All sentences in corpus
s	Single sentence in corpus
k	Size of SkipGram window
r_w	Embedding of word / node w
r'_w	Output embedding of word / node w

DeepWalk and Node2Vec

r	Number of walks per node
l	Length of walks
p	Random walk return parameter
q	Random walk in-out parameter
k	Size of SkipGram window

NBNE

n	Number of permutations
k	Number of neighbors in each sentence and size of SkipGram window

Contents

Chapter 1

Introduction

Many important problems involving graphs require the use of learning algorithms to make predictions about nodes and edges, such as link prediction [Lü and Zhou, 2011; Al Hasan et al., 2006] and node/edge classification [Yang et al., 2011; Radivojac et al., 2013]. These predictions and inferences on nodes and edges from a graph are typically done using classifiers with carefully engineered features [Grover and Leskovec, 2016]. These features, besides taking time and manual labor to be developed and acquired, usually do not generalize well to other problems or contexts.

The field of Natural Language Processing (NLP) has seen many advances due to the use of algorithms that learn word representations, instead of manually extracted features. Originally proposed by Bengio et al. [2003] and commonly used with Word2Vec algorithms like CBOW and SkipGram [Mikolov et al., 2013a], word embeddings are used in many state-of-the-art solutions for neural machine translation [Luong and Manning, 2016; Firat et al., 2016], question answering [Andreas et al., 2016] and natural language generation [Wen et al., 2015].

These advances can't be directly transfered to graphs. While text can be seen as one dimensional, each node in a graph has a different number of connections and there is no straight forward way to read a graph. At the same time, while the same word appears several times in a text corpus, each node only appears in one place in the graph. Recent works have proposed new methods for learning representations for nodes and edges in graphs, based on random walks [Perozzi et al., 2014; Grover and Leskovec, 2016] or auto-encoding adjacency vectors [Wang et al., 2016].

In this work, we propose a new general purpose method for generating node embeddings in very large graphs, which we call Neighborhood Based Node Embeddings (or simply NBNE). NBNE is based on the SkipGram algorithm and uses nodes neighborhoods as contexts. NBNE outperforms state-of-the-art DeepWalk [Perozzi et al., 2014] and Node2Vec [Grover and Leskovec, 2016] for the tasks of link prediction and node classification on six collections, being one to three orders of magnitude faster.

The main reason for this improvement on effectiveness and efficiency is that we concentrate learning on the "predictable" parts of the graph. A study by Facebook research [Edunov et al., 2016] found that each person in the world (at least among the people active on Facebook) is connected to every other person by an average 3.57 other people. In a graph of this magnitude and connectedness, learning node embeddings by maximizing the log-probability of predicting nearby nodes in a random walk (with a window size of 5) can be highly inefficient and make it "harder" for the embeddings to be constructed, even if these random walks are biased as in Node2Vec. We conjecture this can also make learning them numerically more unstable, which would explain why they need more iterations before embedding convergence.

When compared to SDNE [Wang et al., 2016], NBNE produces better results in most experiments, while being 68 $\sim$ 2,009 times faster when SDNE is trained on a GPU and 1,261 $\sim$ 44,896 times faster than SDNE on CPU, despite NBNE being only run on CPU for all experiments. We also evaluated NBNE on an author name disambiguation task, comparing it to a simpler baseline, since Node2Vec, DeepWalk and SDNE had prohibitive computational times for this problem. On this task, NBNE beat our baseline in all 14 conducted experiments.

The main contributions of this work are:

- We present a **new general purpose method** for generating node embeddings in graphs which is more effective and more efficient than state-of-the-art methods.

- Experimental results in solving the link prediction and node classification problems for six graph sets show that **our method outperforms** DeepWalk and Node2Vec, in terms of effectiveness and efficiency.

- In spite of the fact that our method has the same time complexity of the two baselines DeepWalk and Node2Vec, we were able to **improve the training time by one to three orders of magnitude**, which is important when dealing with very large graphs. For instance, to learn node embeddings for a graph containing 317,080 nodes and 1,049,866 edges, collected from the DBLP[1] repository [Yang and Leskovec, 2012], NBNE took approximately 14m30s minutes, DeepWalk approximately 164m34s and Node2Vec approximately 3,285m59s (more than 54 hours).

[1]http://dblp.uni-trier.de

- We provide a thorough evaluation of our method in real and synthetic graphs, **motivating our choice for a semi-supervised algorithm**. Our method has a single tunable parameter (number of permutations, which will be explained later) that can be tuned at once on the training set to avoid overfitting the representations.
- We show that node embedding algorithms can be **successfully used in author name disambiguation, and provide a new large dataset** for this purpose, which consists of 14 separate graphs.

In Chapter 2, we present a bibliographical revision of node embedding algorithms, while Chapter 3 presents our new proposed method, NBNE. Chapter 4 presents experiments on seven real graph datasets and a full comparison with DeepWalk, Node2Vec and SDNE. Chapter 5 presents a further analysis of the algorithm's properties, such as time complexity, choice of hyper-parameters and an assortativity analysis. Finally, Chapter 6 presents conclusions and future works.

Chapter 2

Related Work

The definition of node similarity and finding general purpose node and/or edge representations are non-trivial challenges [Lü and Zhou, 2011]. Many definitions of similarity in graphs use the notion of first and second order proximity. First-order proximity is the concept that connected nodes in a graph should have similar properties, while the second-order proximity indicates that nodes with similar neighborhoods should have common characteristics.

Some earlier works on finding these embeddings use various matrix representations of the graph, together with dimensionality reduction techniques, to obtain the nodes' representations [Roweis and Saul, 2000; Tenenbaum et al., 2000]. A problem with these approaches is that they usually depend on obtaining the matrix' eigenvectors, which is infeasible for large graphs ($O(n^{2.376})$) with the Coppersmith-Winograd algorithm [Coppersmith and Winograd, 1987]). Recent techniques attempt to solve this problem by dynamically learning representations for nodes in a graph using non-linear techniques based either on first and second order proximities [Tang et al., 2015; Wang et al., 2016] or random walks [Perozzi et al., 2014; Grover and Leskovec, 2016].

Other recent works focus on finding representations for specific types of graphs. TriDNR [Pan et al., 2016] uses a graph structure together with node content and labels to learn node representations in two citation networks. Their work can be directly applied to any graph where nodes have labels and/or text contents. TEKE [Wang and Li, 2016] and KR-EAR [Lin et al., 2016] find representations for entities in knowledge graphs and metapath2vec [Dong et al., 2017] finds node representations in heterogeneous networks, in which different edges can have different meanings. The method LINE [Tang et al., 2015] finds a d dimensional representation for each node based on first and second-order graph proximities, not being feasible for large graphs, because its cost function depends on the whole adjacency matrix ($O(|V|^2)$).

Another method, Structural Deep Network Embedding (SDNE) [Wang et al., 2016], is also based on first and second order proximities. It uses autoencoders to learn a compact representation for nodes based on their adjacency matrix (second-order proximity), while forcing representations of connected nodes to be similar (first-order proximity) by using an hybrid cost function. SDNE is also not feasible for large graphs, since the autoenconders are trained on the complete adjacency vectors. Each vector has size $O(|V|)$ and is created at least once, creating a lower bound on time complexity $O(|V|^2)$.

The method DeepWalk [Perozzi et al., 2014] generates k random walks starting on each vertex in the graph to create sentences where each “word” is a node. These sentences are then trained using the SkipGram algorithm to generate node embeddings. This method has a time complexity bounded by $O(|V|\log|V|)$.

Node2Vec [Grover and Leskovec, 2016] also uses random walks with SkipGram and can be seen as a generalization of DeepWalk. The difference between the two methods is that Node2Vec's random walks are biased by two pre-assigned parameters p and q. During the creation of the walks, these parameters are used to increase the chance of the walk returning to a parent node or going farther from it. This method uses a semi-supervised approach which requires several models to be generated and a small sample of labeled nodes to be used so that the best parameters p and q can be chosen. Node2Vec is not efficient for densely connected graphs, since its time and memory dependencies on the graph's branching factor b are $O(b^2)$.

In this work, we considered DeepWalk [Perozzi et al., 2014] and Node2Vec [Grover and Leskovec, 2016] as our main baselines, since they are scalable, having a time complexity ($O(|V|\log|V|)$). The main differences between NBNE and the two baselines are: (i) we use a different sentence sampling strategy which is based in a node's neighborhood instead of random walks, (ii) NBNE is more effective than both Node2Vec and DeepWalk, as supported by our experiments in six different datasets, and (iii) NBNE is efficient for both dense and sparse graphs and scalable for very large applications, having a faster training time than both Node2Vec and DeepWalk.

We also compare our algorithm to SDNE [Wang et al., 2016], which is not scalable to large graphs, having a time complexity $O(|V|^2)$. This algorithm's operations, although having a large time complexity, can be highly parallelized in modern GPUs, so we run SDNE on both CPU and GPU to compare its performance to NBNE.[1]

Next, we describe the main algorithms and baselines used in this work. In Section

[1]Other more recent general purpose methods for generating node embeddings, such as GraphSAGE [Hamilton et al., 2017] and Graph Attention Networks [Veličković et al., 2018], were developed at the same time as this work and weren't used as baselines for this reason.

2.1, we describe SDNE. Section 2.2 presents the SkipGram method in further details. DeepWalk is further described in Section 2.3 and Node2Vec in Section 2.4.

2.1 Structural Deep Network Embedding (SDNE)

As mentioned before, SDNE uses an hybrid cost function to create representations based on both first and second order proximities. It jointly learns a function ($f(\cdot)$) to create an embedding representation (y_i) of a node by using its adjacency vector (x_i), and another function ($\hat{f}(\cdot)$) to recreate its adjacency vector ($\hat{x_i}$) from the embedding, as shown in Equation 2.1, with both $f(\cdot)$ and $\hat{f}(\cdot)$ being multi-layer perceptrons.

$$\begin{aligned} y_i &= f\left(x_i\right) \\ \hat{x_i} &= \hat{f}\left(y_i\right) \end{aligned} \tag{2.1}$$

SDNE uses an autoencoder loss function to learn compact representations for nodes based on their adjacency vectors, shown in Equation 2.2, thus, learning embeddings which present second-order proximity.

$$\mathcal{L}_{2nd} = \sum_{i=1}^{n} \left(||(\hat{x_i} - x_i) \odot b_i||_2^2 \right) \tag{2.2}$$

In this function, $\odot$ is the Hadamard product and $b_i = \{b_{i,j}\}_{j=1}^{n}$ is used to make existing links more relevant in the reconstruction process than non-links. Having $e_{i,j}$ represent the existence of an edge between nodes i and j and β an hyper-parameter tuned using a validation set:

$$\begin{cases} b_{i,j} = 1 & if \quad e_{i,j} = 0 \\ b_{i,j} = \beta > 1 & \quad else \end{cases}$$

At the same time, to force first-order proximities in its representations, SDNE uses another loss function to make the embeddings of connected nodes to be similar. This loss function is shown in Equation 2.3.

$$\mathcal{L}_{1st} = \sum_{i,j=1}^{n} \left(e_{i,j} \, ||y_i - y_j||_2^2 \right) \tag{2.3}$$

SDNE also uses an $\mathcal{L}2$-norm regularizer to prevent overfitting in the representations, which can be seen in Equation 2.4. In this function, $W^{(k)}$ and $\hat{W}^{(k)}$ are the k-th layer weights of the functions f and $\hat{f}$.

$$\mathcal{L}_{reg} = \frac{1}{2} \cdot \sum_{k=1}^{K} \left(\left|\left|W^{(k)}\right|\right|_F^2 + \left|\left|\hat{W}^{(k)}\right|\right|_F^2 \right) \tag{2.4}$$

In order to preserve both first and second order proximities, SDNE then adds all losses to create one hybrid loss function, shown in Equation 2.5. This function preserves both first and second order proximities, while also regularizing results with an $\mathcal{L}2$-norm. The strength of each loss term can be changed using hyper-parameters α and ν, which are also tuned using a validation set.

$$\mathcal{L}_{mix} = \mathcal{L}_{2nd} + \alpha \cdot \mathcal{L}_{1st} + \nu \cdot \mathcal{L}_{reg} \tag{2.5}$$

2.2 SkipGram

SkipGram [Mikolov et al., 2013b] is a word embedding algorithm, which aims to create n-dimensional representations for words in a given text corpus. Firth [1957] stated that “You shall know a word by the company it keeps”. SkipGram follows this statement and learns a word’s representation by using its embedding to predict the other words close to it, which form the context in which it appears.

Mathematically speaking, SkipGram maximizes the log likelihood of, given a word (w_i) in a sentence (s), predicting other words in its context (w_j), as stated in Equation 2.6.

$$\log\left(p\left(s|r\right)\right) = \frac{1}{|s|} \sum_{i=0}^{|s|} \left(\sum_{j=i-5, j\neq i}^{i+5} \left(\log\left(p\left(w_j|w_i, r\right)\right)\right) \right) \tag{2.6}$$

The probabilities in this model are learned using the feature vectors r_{w_i}, which are then used as the words’ representations. The probability $p\left(w_j|w_i, r\right)$ is given by:

$$p\left(w_o|w_i, r\right) = \frac{\exp\left(r\prime_{w_o}^T \times r_{w_i}\right)}{\sum_{w\in W}\left(\exp\left(r\prime_{w}^T \times r_{w_i}\right)\right)} \tag{2.7}$$

where $r\prime_{w_i}^T$ is the transpose of an auxiliary output representation used to learn the model. Doing this for all sentences in a corpus, we get Equation 2.8, which finds the set of representations r that maximize these log probabilities in SkipGram.

$$\max_{r} \quad \frac{1}{|S|} \sum_{s\in S} \left(\log\left(p\left(s|r\right)\right)\right) \tag{2.8}$$

These generated n-dimensional representations are arranged in this space such that similar words have similar representations (eg. cat $\approx$ dog; a $\approx$ an), and that arithmetic operations on them preserve semantic meaning (eg. women - men + king = queen).

2.3 Deep Walk

DeepWalk [Perozzi et al., 2014] uses random walks in the graph to create sentences where each "word" is a node, which are then used to learn the embeddings with the SkipGram algorithm. More specifically, it first generates r sentences of length l by doing random walks starting from each node in the graph, as described in Algorithm 1. With these sentences, it runs SkipGram on them with a window size k to generate its embeddings.

Algorithm 1 DeepWalk Sentence Sampling

```
1: procedure GETSENTENCES(graph, $r$, $l$)
2:     $sentences \leftarrow [\emptyset]$
3:     for $node$ in $graph$.nodes() do
4:         for $j$ in $0 : r$ do
5:             $sentence \leftarrow$ random_walk($graph, node, l$)
6:             $sentences$.append($sentence$)
```

2.4 Node2Vec

Node2Vec, similarly to DeepWalk, uses random walks in a graph to generate sentences, following Algorithm 1, which are later used with SkipGram to generate node embeddings. The difference between both algorithms is that Node2Vec is semi-supervised, having two parameters, p and q, which bias the way random walkers move to generate sentences and which are tuned using a validation set.

In Node2Vec, after arriving at node j from node i, the probability that the walker goes to node z depends on the distance ($d_{i,z}$) between nodes i and z:

$$\alpha(i,z) = \begin{cases} \frac{1}{p} & d_{i,z} = 0 \\ 1 & d_{i,z} = 1 \\ \frac{1}{q} & d_{i,z} = 2 \end{cases}$$

$$p(j \to z \mid i \to j) = \frac{\alpha(i,z)}{\sum_{m \in j.\text{neighbors()}} \alpha(i,m)} \tag{2.9}$$

In these equations, p can be seen as a **return parameter**, which controls the probability of a random walk immediately revisiting a node. Parameter q is called the **in-out parameter**, controlling the probability of staying in a certain region of the graph, or getting farther away from it.

Chapter 3

Neighborhood Based Node Embeddings

The context of a word is not a straightforward concept, but it is usually approximated by the words surrounding it. In graphs, a node's context is an even more complex concept. As explained in Chapter 2, DeepWalk and Node2Vec use random walks as sentences and consequently as contexts in which nodes appear.

In this work, the contexts are based solely on the neighborhoods of nodes, defined here as the nodes directly connected to it, focusing mainly on the second-order proximities. Consequently, nodes' representations will be mainly defined by their neighborhoods and nodes with similar neighborhoods (contexts) will be associated with similar representations.

NBNE separates a nodes' neighborhood in small groups and then maximizes the log likelihood of predicting a node given another in such a group. Group sampling is defined in Section 3.1 and the method to learn the representations from the groups is further explained in Section 3.2. Section 3.3 describes how to reduce overfitting in the representations.

3.1 Groups Generation

In our Neighborhood Based Node Embedding's (NBNE) method, as the name implies, groups are created based on the neighborhoods of nodes. There are two main challenges in forming groups from neighborhoods, as follows:

- Nodes have different connectivities/degrees, so groups containing all the neighbors from a node are difficult to treat.

- There is no explicit order in the nodes in a neighborhood. So there is no clear way to choose the order in which they would appear in a group.

In this work, the solution is to form small groups, with only k neighbors in each, using random permutations of these neighborhoods. Algorithm 2 presents the code for generating groups. As a trade-off between training time and increasing the training dataset the user can select the number of permutations n. Selecting a higher value for n creates a more uniform distribution on possible neighborhood groups, but also increases training time.

Algorithm 2 Groups Sampling

```
1: procedure getGroups(graph, $n$, $k$)
2:     $groups \leftarrow [\emptyset]$
3:     for $j$ in $0 : n$ do
4:         for $node$ in $graph$.nodes() do
5:             $neighbors \leftarrow$ random_permutation($node$.neighbors())
6:             for $i$ in $0 : \text{len}(neighbors)/k$ do
7:                 $group \leftarrow [node] + neighbors[i \cdot k : (i+1) \cdot k]$
8:                 $groups$.append($group$)
```

It is important to note that nodes which have a higher degree will be present in more groups, being further trained, while sparser nodes will be trained less, appearing "less frequently" during training. This is good, since nodes from which we have more information (more connections) will be trained more than sparser nodes, which could overfit to have representations too close to its neighbors.

3.2 Learning Representations

As described in Section 3.1, Algorithm 2 forms a set of groups S, where each member is a node from the graph. We train the vector representations of nodes by maximizing the log likelihood of predicting a node given another node in a group and given a set of representations r, making each node in a group predict all the others. The log likelihood maximized by NBNE is given by:

$$\max_r \quad \frac{1}{|S|} \sum_{s \in S} \left(\log \left(p \left(s | r \right) \right) \right) \tag{3.1}$$

where $p\left(s|r\right)$ is the probability of each group, which is given by:

$$\log\left(p\left(s|r\right)\right) = \frac{1}{|s|}\sum_{i\in s}\left(\sum_{j\in s, j\neq i}\left(\log\left(p\left(v_j|v_i,r\right)\right)\right)\right) \tag{3.2}$$

where v_i is a vertex in the graph and v_j are the other vertices in the same group. The probabilities in this model are learned using the feature vectors r_{v_i}, which are then used as the vertex representations. The probability $p\left(v_j|v_i,r\right)$ is given by:

$$p\left(v_j|v_i,r\right) = \frac{\exp\left(r\prime_{v_j}^{T} \times r_{v_i}\right)}{\sum_{v\in V}\left(\exp\left(r\prime_{v}^{T} \times r_{v_i}\right)\right)} \tag{3.3}$$

where $r\prime_{v_j}^{T}$ is the transposed output feature vector of vertex j, used to make predictions. The representations $r\prime_v$ and r_v are learned simultaneously by optimizing Equation 3.1.[1] This is done using stochastic gradient ascent with negative sampling [Mikolov et al., 2013b].

By optimizing this log probability, the algorithm maximizes the likelihood of predicting a neighbor given a node, creating node embeddings where nodes with similar neighborhoods have similar representations. Since there is more than one neighbor in each group, this model also makes connected nodes have similar representations, because they will both predict each others neighbors, resulting in representations also having some first order similarities. A trade-off between first and second order proximity can be achieved by changing the parameter k, which controls the size of the generated groups. A further discussion on this effect can be seen in Section 5.3.3.

3.3 Avoiding Overfitting Representations

When using large values of n (i.e., number of permutations) on graphs with few edges per node, some overfitting can be seen on the representations, as shown in details in Section 5.1 and in Section 5.3.2. This overfitting can be avoided by sequentially training on increasing values of n and testing the embeddings on a validation set every few iterations, stopping when performance stops improving, as shown in Algorithm 3.

[1]This is the same mathematical formulation as in SkipGram.

Algorithm 3 NBNE without Overfitting

```
1: procedure trainNBNE(graph, max_n)
2:     $groups \leftarrow$ get_groups($graph, max\_n$)
3:     $model \leftarrow$ [initialize_model()]
4:     for $j$ in $0 : log_2(max\_n)$ do
5:         $model \leftarrow$ train($model, groups[2^j : 2^{j+1}]$)
6:         $error \leftarrow$ test($new\_model, validation\_set$)
7:         if $error > old\_error$ then
8:             break
9:         $old\_error \leftarrow error$
```

Chapter 4

Experiments

NBNE was evaluated on three different tasks: link prediction, node classification, and author name disambiguation.[1] We used a total of seven datasets to evaluate NBNE and a brief description of each dataset can be found in Section 4.1. We present results for the link prediction problem in Section 4.2, for the node classification problem in Section 4.3 and for an author name disambiguation task in Section 4.4. For all experiments we used groups of size $k = 5$ and embeddings of size $d = 128$, while the number of permutations was run for $n \in \{1, 5, 10\}$. The best value of n was chosen according to the precision on the validation set and we used early stopping, as described in Section 3.3.

On the link prediction and node classification tasks, DeepWalk and Node2Vec were used as baselines, having been trained and tested under the same conditions as NBNE and using the parameters as proposed in [Grover and Leskovec, 2016]. More specifically, we trained them with the same training, validation and test sets as NBNE and used a window size of 10 (k), walk length (l) of 80 and 10 runs per node (r). For Node2Vec, which is a semi-supervised algorithm, we tuned p and q on the validation set, doing a grid search on values $p, q \in \{0.25; 0.5; 1; 2; 4\}$. For the author name disambiguation task, we did not use these methods as baselines due to their prohibitive computational costs. A further comparison between NBNE and SDNE on link prediction and node classification can be seen on Section 4.5.

4.1 Datasets

We used a total of seven graph datasets to evaluate NBNE. Next we briefly describe these datasets:

[1]`https://github.com/tiagopms/nbne`

1. Facebook [McAuley and Leskovec, 2012]: A snapshot of a subgraph of Facebook, where nodes represent users and edges represent friendships.

2. Astro [Leskovec et al., 2007]: A network that covers scientific collaborations between authors whose papers were submitted to the Astrophysics category in Arxiv.

3. Protein-Protein Interactions (PPI) [Breitkreutz et al., 2008]: We use the same subgraph of the PPI network for Homo Sapiens as in [Grover and Leskovec, 2016]. This subgraph contains nodes with labels from the hallmark gene sets [Liberzon et al., 2011] and represent biological states. Nodes represent proteins, and edges indicate biological interactions between pairs of proteins.

4. Wikipedia [Mahoney, 2011]: A co-occurrence network of words appearing in the first million bytes of the Wikipedia dump. Labels represent Part-of-Speech (POS) tags.

5. Blog [Zafarani and Liu, 2009]: A friendship network, where nodes are bloggers and edges indicate friendships. Each node in this dataset has one class which is referent to the blogger's group.

6. DBLP [Yang and Leskovec, 2012]: A co-authorship network where two authors are connected if they published at least one paper together.

7. DBLP-ambiguous (DBLP-amb): This is a new dataset that we created and used in the author name disambiguation problem. It contains 14 separate co-authorship networks (14 separate graphs). Each graph contains one of the most prolific authors' in DBLP and their co-authors' connections. To assemble it, the DBLP repository was crawled and 14 of the most prolific ambiguous authors were collected, together with their direct co-authors' complete profiles. More information on this dataset can be seen in Section 4.4.

Table 4.1 presents details about the first six of these datasets, used in the link prediction and node classification tasks. It presents the number of nodes, edges and classes in each of these datasets, also showing the number of edges per node in them. We can see that Blog is the graph with the largest branching factor, 32.38, and DBLP is the one with the smallest, 3.31. At the same time, PPI is the smallest graph, with only 3,890 nodes, while DBLP, the largest, contains 317,080 nodes and 1,049,866 edges. Further analysis on these graphs' assortativity properties is presented in Section 5.3.1.

Table 4.1: Statistics on the first six graph datasets

	Nodes	Edges	Edges/Node	# Classes
Facebook[1]	4,039	88,234	21.84	-
Astro[1]	18,772	198,110	10.55	-
PPI[1,2]	3,890	38,739	9.95	49
Wikipedia[1,2]	4,777	92,517	19.36	39
Blog[1,2]	10,312	333,983	32.38	39
DBLP[1]	317,080	1,049,866	3.31	-

[1] used in Link Prediction
[2] used in Node Classification

Figure 4.1 shows the distribution of classes in the three datasets used for node classification. While Wikipedia has a long tailed distribution, with one class being present in almost 50% of its nodes, PPI's probabilities are well distributed along the 49 different possible classes.

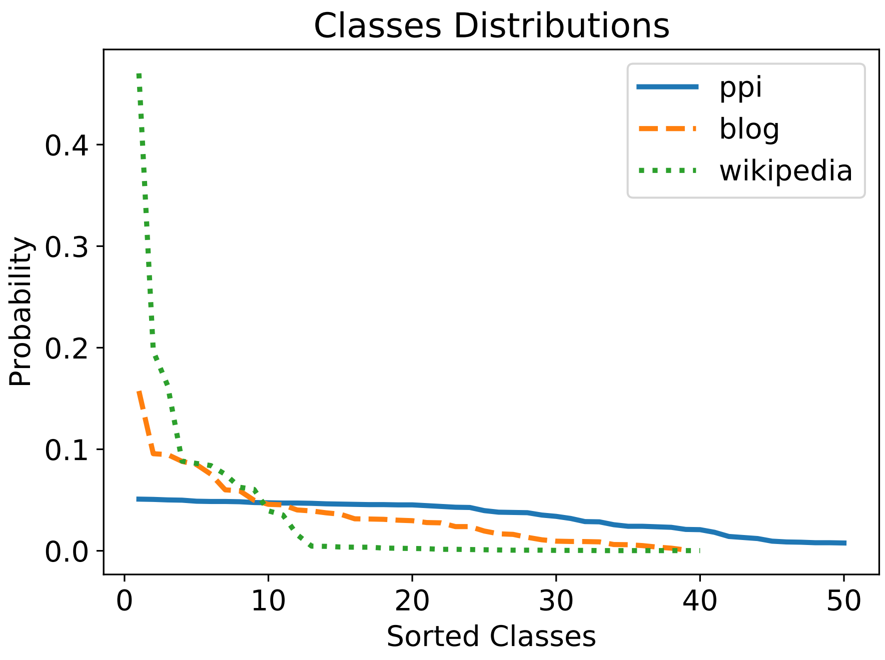


Figure 4.1: (Color online) Distribution of percentage of nodes per class in label classification datasets with classes sorted by their frequency.

4.2 Link Prediction

Link prediction attempts to estimate the likelihood of the existence of a link between two nodes, based on observed links and the nodes' attributes [Lü and Zhou, 2011]. Typical approaches to this task are based on similarity metrics, such as Common Neighbors or Adamic-Adar [Adamic and Adar, 2003]. Sarkar et al. [2011] presents theoretical justifications for the performance of these similarity metrics, while formalizing the link prediction problem as one of estimating distances between nodes in latent spaces. Instead of these heuristic-based similarity metrics, we propose to train a logistic classifier based on the concatenation of the embeddings from both nodes that possibly form an edge and predict the existence or not of the edge.

4.2.1 Setup

To train NBNE on this task, we first obtained a sub-graph with 90% of the edges from each dataset uniformly select at random, and obtained the node embeddings by training NBNE on this sub-graph. Then, we separated a small part of these sub-graph edges as a validation set, using the rest to train a logistic regression with the learned embeddings as features.

After the training was completed, the unused 10% of the edges were used as a test set to predict new links. 10-fold cross-validation was used on the entire training process to access the statistical significance of the results, analyzing statistical difference between the baselines and NBNE.[2] To evaluate the results on this task, we used as metrics: AUC (area under the ROC curve) [Baeza-Yates and Ribeiro-Neto, 2011], and training time.[3] The logistic regressions were all trained and tested using all available edges (respectively in the training or test set), and an equal sized sample of negative samples, which, during training, included part of the 10% removed edges.

4.2.2 Results

Table 4.2 presents results for this task. Considering AUC scores on the Link Prediction task, NBNE was statistically better than both DeepWalk and Node2Vec on all datasets except Facebook, in which there was no statistically significant difference. NBNE was better than the baselines on the Astro and PPI datasets, with more than 7% improvement, also showing a 4.67% performance gain in Wikipedia and a small, but

[2]In all experiments we performed Welch's t-tests with $p = 0.01$. The symbol * marks results which are statistically different from NBNE.

[3]Training times were all obtained using 16 core processors, running NBNE, Node2Vec or DeepWalk on 12 threads, with all algorithms having been implemented using gensim [Řehůřek and Sojka, 2010].

statistically significant, gain on Blog. Only losing by a small percentage on Facebook, with a difference that was not statistically significant.

In DBLP, NBNE again presents the best AUC score, although this difference was small and its statistical significance could not be verified due to the large training times of the baselines. This dataset contains the largest graph analyzed in this work (317,080 nodes and 1,049,866 edges) and in it, to train a single fold, Node2Vec took 3,285m59s (more than 54 hours) and DeepWalk took 164m34s (approximately 2 hours and 44 minutes), while NBNE took only 14m30s, which represents a 226/11 times improvement over Node2Vec and DeepWalk, respectively.

Considering training time for this task, NBNE presents the biggest improvements on sparser networks of medium size, like Astro, PPI and Wikipedia datasets. On these graphs, the best results are for $n = 1$, resulting in more than 50x faster training than DeepWalk and more than 1,500 times faster than Node2Vec, achieving a 6,049 times faster training than Node2Vec on Wikipedia. For the Blog and Facebook datasets the best results are for $n = 5$, resulting in larger training times, but still more than one order of magnitude faster than DeepWalk and more than 350 times faster than Node2Vec. For the DBLP dataset, the best results were achieved with $n = 10$, still much faster than the baselines.

Table 4.2: Link prediction results

	Facebook		Astro		PPI	
	AUC	Training Time	AUC	Training Time	AUC	Training Time
NBNE	0.9688	**0m11s**	**0.8328**	**0m07s**	**0.8462**	**0m02s**
DeepWalk	0.9730	2m26s	0.7548*	6m55s	0.7741*	2m30s
Node2vec	**0.9762**	69m33s	0.7738*	182m16s	0.7841*	66m37s
Gain	-0.76%	12.96x 369.85x	7.62%	59.06x 1555.80x	7.91%	77.43x 2061.67x

	Wikipedia		Blog		DBLP	
	AUC	Training Time	AUC	Training Time	AUC	Training Time
NBNE	**0.6853**	**0m02s**	**0.9375**	**1m11s**	**0.9335**†	**14m30s**
DeepWalk	0.6534*	7m38s	0.9098*	28m13s	0.9242‡	164m34s
Node2Vec	0.6547*	236m60s	0.9202*	838m41s	0.9322‡	3,285m59s
Gain	4.67%	194.86x 6049.77x	1.88%	23.86x 709.24x	0.13%	11.34x 226.52x

† average of 10 fold results
‡ no statistical tests were run, due to the time necessary to run a single fold

4.3 Node Classification

Given a partially labeled graph, node classification is the task of inferring the classification of the unknown nodes, using the structure of the graph and/or the properties of the nodes. In this section, we again compare our algorithm's performance to DeepWalk and Node2Vec, now analyzing node classification tasks on the Blog, PPI and Wikipedia datasets.

4.3.1 Setup

In this task, the node embeddings were trained using NBNE on the complete graph. After obtaining the node embeddings, 80% of the labeled nodes in the graph were used to train a logistic classifier that predicted the class of each node, while 5% of the nodes were used for validation and the remaining 15% nodes were used as a test set. This entire process was repeated for 10 different random seed initializations to access the statistical relevance of the results.

4.3.2 Results

Results on the Blog, PPI and Wikipedia datasets are shown in Table 4.3 and are presented in terms of Macro F1 scores and training times. NBNE produces statistically similar results to its baselines, in terms of Macro F1, on both PPI and Wikipedia, while showing a statistically significant 22.45% gain in the Blog dataset, indicating that NBNE's embeddings did not only get a better accuracy on Blog, but also that correct answers were better distributed across the 39 possible classes.

Considering training times, NBNE is more than 10 times faster than DeepWalk on these three datasets and is [300 to 600] times faster than Node2Vec. NBNE did not show statistically worse results in any dataset analyzed here, while having an order of magnitude faster training time than DeepWalk and more than two orders of magnitude faster training time than Node2Vec.

4.4 Author Name Disambiguation

One of the hardest problems faced by current scholarly digital libraries is author name ambiguity [Ferreira et al., 2012]. This problem occurs when an author publishes works under distinct names or distinct authors publish works under similar names [Ferreira et al., 2015]. Automatic solutions, which are effective, efficient and practical in most

Table 4.3: Node classification results

	Blog		PPI		Wikipedia	
	Macro F1	Training Time	Macro F1	Training Time	Macro F1	Training Time
NBNE	**0.2004**	**1m57s**	0.0978	**0m16s**	**0.0727**	**0m41s**
DeepWalk	0.1451*	31m31s	**0.0991**	3m04s	0.0679	13m04s
Node2vec	0.1637*	959m12s	0.0971	83m02s	0.0689	408m00s
Gain	22.45%	16.18x 492.57x	-1.35%	11.82x 319.78x	5.56%	19.04x 594.62x

situations, are still in need [Santana et al., 2014]. In this section, we test our algorithm against the case where distinct authors publish works under similar names.

4.4.1 Setup

For this problem, the DBLP repository was crawled and the profiles of fourteen of the most prolific ambiguous authors were obtained, together with their direct co-authors' complete profiles.[4] With this, we created a new dataset, called here DBLP-ambiguous, consisting of fourteen separate co-authorship networks (14 separate graphs), each with all the connections of one of these homonymous authors and their co-authors' connections. Details on these graphs can be seen in Table 4.4.

Table 4.4: DBLP-ambiguous (DBLP-amb) Dataset Details

Name	# Authors	Nodes	Edges
JingLi	4	105,746	589,367
JingWang	16	108,913	581,457
JunLiu	2	106,533	590,032
JunWang	21	121,511	691,705
JunZhang	16	116,497	631,738
LeiZhang	38	118,798	664,898
LiZhang	14	122,403	693,916
WeiLi	57	157,427	887,727
WeiWang	85	183,962	1,103,702
WeiZhang	52	131,200	722,272
XiaodongWang	3	50,854	284,733
XinWang	14	107,920	578,084
YangLiu	33	130,319	740,501
YuZhang	9	131,683	734,214

[4]The code used to crawl the DBLP is available at https://github.com/tiagopms/dblp-crawler.

Using these co-authorship networks, embeddings were obtained by training on the graphs with 20% of the papers from each ambiguous author removed. After the embeddings had already been learned for each author, the probability of each possible author-coauthors "group" was calculated as:

$$s_{possible\_author} = \left[v_{possible\_author},\ v_{coauthor\_1},\ \ldots,\ v_{coauthor\_j}\right]$$

This probability is given by:

$$p(author) = \frac{1}{T}\sum_{t=1}^{T}\left(\sum_{-k\leq j\leq k, j\neq 0}\left(\log\left(p\left(v_{t+j}|v_t\right)\right)\right)\right) \tag{4.1}$$

where $v_1 = author$, which comes from the NBNE model itself.

As a baseline, we used the typical solution that classifies the closest of the possible ambiguous authors as co-author for each of the test papers. If no path on the graph existed to any of the possible ambiguous authors, or if there was a tie between the distances to two or more of them, a random one was chosen between the possible ones. DeepWalk and Node2Vec were not used as baselines for this task due to the size of the 14 graphs analyzed here, most with more than 100,000 nodes and 500,000 edges, which would result in a prohibitive training time.

4.4.2 Results

Table 4.5 presents the results for the author name disambiguation task for each chosen author. This experiment was run using NBNE as an unsupervised algorithm with a fixed number of permutations $n = 10$, having no validation set. We also used groups of size $k = 5$ and node embeddings of size $d = 128$.

After the embeddings had already been learned for each author, which can be done off-line, the NBNE algorithm was faster in assigning the authors than its baseline. This occurred because it only required computing the probability of each possible author-coauthors "group" ($p(s)$), while the baseline had to dynamically get the distance between the papers' co-authors and the possible authors.

It can be seen in Table 4.5 that for all but two authors the precision was higher when using the NBNE embeddings instead of the graph baseline, while for the other two precision score remained the same.

Table 4.5: Author name disambiguation results

Name	# Authors	Algorithm	Precision
Jing Li	4	NBNE	0.9415
		Baseline	0.9415
JingWang	16	NBNE	0.8791
		Baseline	0.8512
JunLiu	2	NBNE	0.9709
		Baseline	0.9651
JunWang	21	NBNE	0.8357
		Baseline	0.7821
JunZhang	16	NBNE	0.8206
		Baseline	0.8130
LeiZhang	38	NBNE	0.8843
		Baseline	0.8309
LiZhang	14	NBNE	0.8661
		Baseline	0.8201
WeiLi	57	NBNE	0.8221
		Baseline	0.7822
WeiWang	85	NBNE	0.8143
		Baseline	0.8070
WeiZhang	52	NBNE	0.8408
		Baseline	0.8184
XiaodongWang	3	NBNE	0.9697
		Baseline	0.9576
XinWang	14	NBNE	0.8639
		Baseline	0.8639
YangLiu	33	NBNE	0.7955
		Baseline	0.7540
YuZhang	9	NBNE	0.9268
		Baseline	0.9024
Average		NBNE	0.8737
		Baseline	0.8492

4.5 Comparison with SDNE

Structural Deep Network Embedding [Wang et al., 2016] is another algorithm used for learning node embeddings in graphs. As described in Chapter 2, SDNE is based on first and second order proximities, using autoencoders to learn compact representations for nodes based on their adjacency vector (second-order proximity), while forcing representations of connected nodes to be similar (first-order proximity) by using an hybrid cost function.

This algorithm has a time complexity of $O(|V|^2)$, but its main computation, which is calculating the gradients of its cost function and updating model parameters, can be highly parallelized by using modern GPUs and Deep Learning frameworks. In this section, we compare NBNE and SDNE in terms of both efficacy and efficiency, analysing both AUC/Macro F1 scores and training time. For this purpose, we trained SDNE embeddings using both a dedicated K40 GPU with CUDA 8.0 and a dedicated 16 core linux server.[5]

In the work where it was originally proposed, SDNE was run in a semi-supervised setting, finding the best value of α, β and ν by tuning them on a small validation set. In this work we fix $\alpha = 0.2$ and $\beta = 10$, since in their work they state that these values commonly give the best results, while only choosing ν in a semi-supervised manner. We use SDNE's architecture with [10,300; 1,000; 128] nodes on each layer and test it on both Link Prediction and Node Classification tasks, using the same steps described in Sections 4.2 and 4.3. We train these embeddings using $\nu \in \{0.1, 0.01, 0.001\}$ and choose the best value on the same validation sets used to tune n for NBNE and p and q for Node2vec.

Table 4.6 shows results obtained when using either NBNE or SDNE embeddings on Link Prediction tasks. In this table we can see that both algorithms produce similar results in terms of AUC scores, with each having a statistically significant better result on two datasets, and NBNE having a non statistically significant, but slightly better result on the fifth. It is clear that even when training SDNE using a K40 GPU, NBNE still has more than an order of magnitude faster training time on all datasets, being more than two orders of magnitude faster on most. When comparing to SDNE trained on a CPU, NBNE has more than three orders of magnitude faster training time. On Astro, the dataset with the largest number of nodes analyzed here, NBNE had a 2,009 times faster training time compared to SDNE on a GPU and 44,896 times faster compared to SDNE on CPU.[6]

Table 4.7 shows the results of running NBNE and SDNE on the Node Classification task. On this task NBNE gave statistically better results on two datasets, with an impressive gain of 29.27% on PPI and 46.94% on Blog, only losing on Wikipedia also by a large margin of -20.20%. We can again see that NBNE has more than an order of magnitude faster training time than SDNE on a GPU in this dataset, being more than two orders of magnitude faster when SDNE is trained on a CPU.

Analyzing both these tables we can also see that the largest gains in training time occur when using NBNE on a large but sparse network, such as Astro. This agrees

[5]SDNE code was implemented using Tensorflow [Abadi et al., 2015]

[6]We tried running SDNE with the DBLP dataset, but after five days it hadn't reached half of the training, so we stopped it.

Table 4.6: Link prediction results with SDNE

	Facebook		Astro		PPI	
	AUC	Training Time	AUC	Training Time	AUC	Training Time
NBNE	**0.9688**	**0m11s**‡	**0.8328**	**0m07s**‡	0.8462	**0m02s**‡
SDNE	0.9510*	20m34s† 242m10s‡	0.8157*	234m24s† 5,237m59s‡	**0.8751***	16m10s† 232m01s‡
Gain	1.87%	112.21x 1,320.91x	2.10%	2,009.17x 44,896.96x	-3.30%	485.10x 6,960.34x

	Wikipedia		Blog		DBLP	
	AUC	Training Time	AUC	Training Time	AUC	Training Time
NBNE	**0.6853**	**0m02s**‡	0.9375	**1m11s**‡	**0.9335**	**14m30s**‡
SDNE	0.6781	22m23s† 337m47s‡	**0.9462***	81m33s† 1,492m47s‡	-	- -
Gain	1.06%	671.59x 10,133.46x	-0.92%	68.92x 1,261.51x	-	- -

‡ Training time on CPU
† Training time on GPU

Table 4.7: Node classifications results with SDNE

	Blog		PPI		Wikipedia	
	Macro F1	Training Time	Macro F1	Training Time	Macro F1	Training Time
NBNE	**0.2005**	**1m57s**‡	**0.0978**	**0m16s**‡	0.0727	**0m41s**‡
SDNE	0.1364*	96m48s† 1,476m33s‡	0.0757*	16m52s† 231m04s‡	**0.0911***	19m60s† 338m40s‡
Gain	46.94%	49.64x 757.20x	29.27%	63.24x 866.48x	-20.20%	29.26x 495.60x

‡ Training time on CPU
† Training time on GPU

with our theoretical analysis, since SDNE's time complexity grows quadratically with the number of nodes $O(|V^2|)$ and NBNE's grows with $O(|V| \cdot \log(|V|) \cdot b)$, which is close to linear on the number of nodes for large graphs.

Chapter 5

Further Analysis

5.1 Number of Permutations (n)

The quality of NBNE's embeddings depends on both the size of the embeddings (d) and the number of permutations (n). For highly connected graphs, larger numbers of permutations should be chosen ($n \in [10, 1000]$) to better represent distributions, while for sparser graphs, smaller values can be used ($n \in [1, 10]$).

Figure 5.1 shows AUC scores versus embedding sizes for several values of n on the Facebook link prediction task. Quadratic functions approximating $\log(auc\_score)$ were plotted to allow for a better understanding of the results. It can be seen that for larger numbers of permutations ($n > 100$) results improve with embedding size, while for small ones ($n = 1$) they decrease. The plot also shows that $n = 10$ gives fairly robust values for all tested embedding sizes.

A further analysis can be seen in Table 5.1, which indicates that graphs with more edges per node tend to get better results with larger values of n, while graphs with a smaller branching factor have better results with only one permutation ($n = 1$). Other factors also enter into account when choosing n, like graph size. For example, link prediction on the DBLP graph had its best results for $n = 10$, although its branching size was only 3.31. Further experiments on this parameter can be seen in Sections 5.3.2 and 5.4.1.

5.2 Time Complexity

SkipGram's time complexity is linear on the number of sentences (S) and embedding size (d) and logarithmic on the size of the vocabulary ($|V|$) ($O\left(|S| \cdot d \cdot \log(|V|)\right)$)

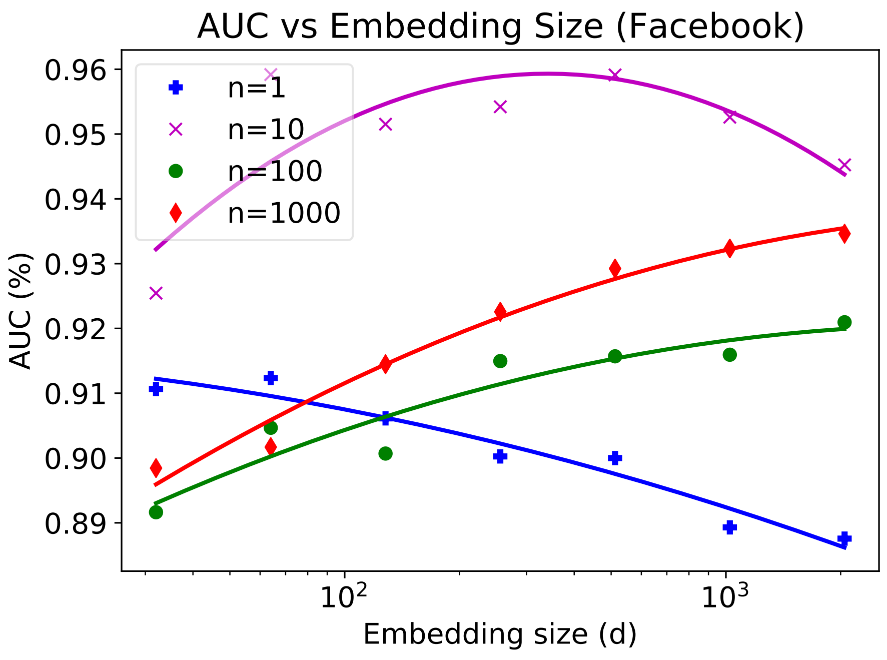


Figure 5.1: (Color online) NBNE AUC scores vs embedding sizes on Facebook dataset with 50% edges removes

Table 5.1: Link Prediction results for varying n with NBNE

n	PPI (9.95[†])		Facebook (21.84[†])		Blog (32.38[†])	
	Precision Test	AUC	Precision Test	AUC	Precision Test	AUC
10	0.7108	0.7795	0.9061	0.9642	0.8627	**0.9348**[‡]
5	0.7305	0.8071	**0.9070**	**0.9688**	**0.8681**	**0.9375**[‡]
1	**0.7751**	**0.8462**	0.8410	0.9150	0.8374	0.9146

[†] Edges per node [‡] No statistical difference

[Mikolov et al., 2013a]. Since the number of sentences is the number of groups, which is linear on the number of permutations (n), branching factor of the graph (b) and on the number of nodes, which is the size of the vocabulary ($|V|$):

$$|S| \propto n \cdot b \cdot |V|$$

NBNE's algorithm will take a time bounded by:

$$O\left(d \cdot n \cdot b \cdot |V| \cdot \log(|V|)\right)$$

Figure 5.2 shows training time is indeed linear on both embedding size and number of permutations. This graph is plotted in a log-log scale, so the initial "flatness" implies the algorithm has a large constant in its computational time, and linearity is implied by the line inclination. This figure also shows that Node2Vec is considerably slower than DeepWalk, and only has a similar training time to running NBNE with at least $n = 1000$. NBNE with $n < 10$ was by far the fastest algorithm.

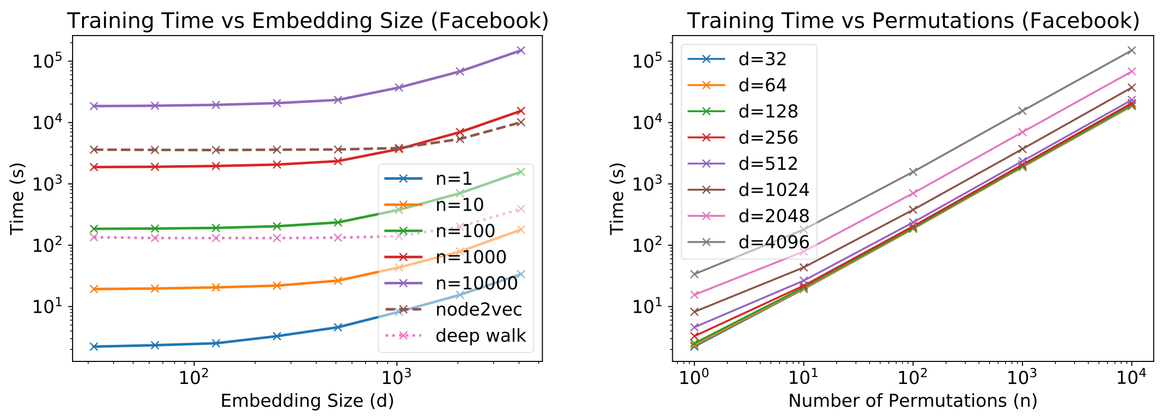


Figure 5.2: (Color online) Facebook dataset with 30% edges removed: Training times vs **(Left)** embedding size **(Right)** number of permutations.

NBNE, Node2Vec and DeepWalk run in a time close to linear in $|V|$ ($O(|V| \log |V|)$), as can be seen in Figure 5.3 (Left). Figure 5.3 (Right) shows that NBNE's time complexity is linear in the branching factor b, while Node2Vec's is quadratic. DeepWalk's running time is roughly constant for this parameter, but for a graph with a larger branching factor, a larger number of walks per node should be used to train this algorithm, which would make its training time increase indirectly with b.

5.3 Assortativity

Assortativity, also referred to as homophily in social network analysis, is a preference of nodes to attach themselves to others which are similar in some sense. In this section, we investigate assortativity properties of the representations generated by our algorithm and of the graphs themselves. In Section 5.3.1, we do a quantitative analysis on the homophily inherent to the datasets considered in this work. In Section 5.3.2, we make a qualitative analysis of how assortativity varies depending on the number of permutations n. In Section 5.3.3, we make a qualitative analysis on the trade-off between first and second order proximities based on the choice of k.

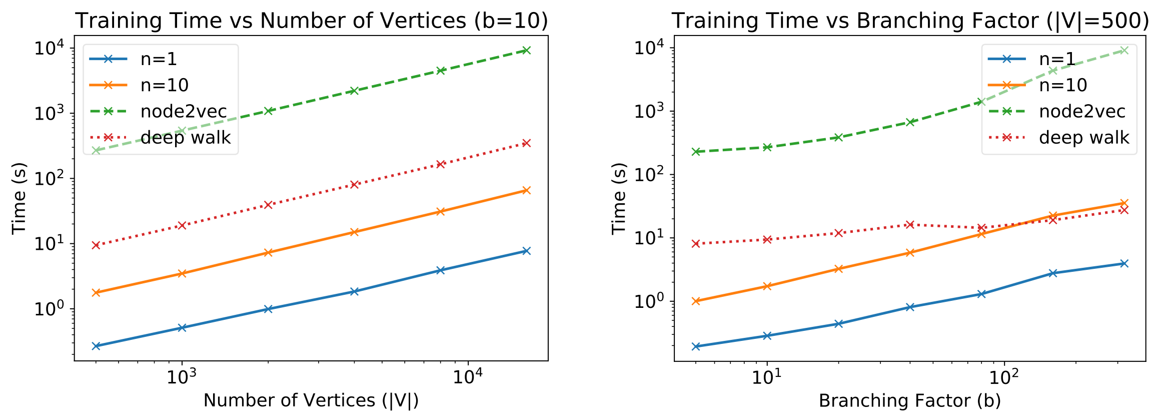


Figure 5.3: (Color online) Training times vs **(Left)** number of vertices on randomly generated graphs with $b = 10$ **(Right)** branching factor on randomly generated graphs with $|V| = 500$.

5.3.1 Datasets' Homophily

There are several ways to quantitatively capture the homophily present in a graph. Jensen and Neville describe relational auto-correlation, which is Pearson's contingency coefficient on the characteristics of nodes which share edges [Jensen and Neville, 2002; Neville and Jensen, 2007]. Park and Barabási [2007] define dyadicity and heterophilicity, which respectively measure how a graph's nodes share common/different characteristics in edges, compared to a random model.

Table 5.2 presents both degree and label assortativity properties of the six graphs analyzed here, calculated using the assortativity coefficient, as defined in Newman [2003]. We can see in this table that the datasets analyzed in this work cover a broad spectrum of assortativity properties. PPI, Wikipedia and Blog graphs present negative degree assortativity, which means nodes in these graphs are more likely to connect with nodes of different connectivity degrees. At the same time, Facebook, Astro and DBLP present positive degree assortativity, which indicates that their nodes tend to connect to others with similar degrees.

We also analyze graphs with both positive and negative label assortativity in our label classification task. While PPI and Blog datasets present positive label assortativity, with connected nodes more frequently sharing classes, Wikipedia has a negative assortativity, with its connected nodes being more likely to have different classes.

Table 5.2: Datasets homophily information

	Assortativity	
	Degree[1]	Label[1]
Facebook	0.0635	-
Astro	0.2051	-
PPI	-0.0930	0.0533
Wikipedia	-0.2372	-0.0252
Blog	-0.2541	0.0515
DBLP	0.2665	-

[1] Calculated as in [Newman, 2003]

5.3.2 More on the Number of Permutations (n)

In this section, we analyze how the number of permutations (n) influences both homophily and overfitting in our learned representations. We qualitatively measure homophily by comparing either cosine or euclidean distances between adjacent nodes to the distances of non-adjacent nodes.

The cosine distances for the PPI dataset, shown by the box plots in Figure 5.4 (top-left), clearly show for larger values of n how the embeddings overfit to the specific graph structure, with the learned similarity on edges not generalizing to the links which were previously removed. In this graph, for larger numbers of permutations the removed edges have a distribution more similar to the non-edges than to the edges used during training, which is a tendency that can be observed in the other graphs, although in a smaller scale.

The box plots in Figure 5.4 (top-right) show the cosine distance for Facebook nodes. We can see that for $n = 5$ there is a larger separation between removed edges and non edges, which justifies the algorithm's choice of this value. For larger values of n we can again see an overlap between the distributions, caused by the embeddings overfitting. On the other hand, the cosine distances for the DBLP in Figure 5.4 (bottom-left) show the largest separation for $n = 10$.

Finally, the box plots in Figure 5.4 (bottom-right) show cosine distances for the Blog dataset. We can see that for $n = 1$ and $n = 5$ there is actually a larger cosine distance between nodes in removed edges than in non-edges, with this situation only inverting for $n \geq 10$. This happens due to this graph's negative degree homophily. This is also observed for similar graphs in the PPI and Wikipedia datasets, though with a smaller intensity in the PPI graph, which has a smaller absolute value of degree assortativity and where only embeddings for $n = 1$ present this property.

The box plots from Figure 5.4 further support our intuition that graphs with larger branching factors should have larger values of n. At the same time, this choice also depends on the graph size and structure, as shown by the algorithms choice of $n = 10$ for the DBLP dataset, which contains the largest degree assortativity. The best choice of n depends on the task in hand, but we believe that, at least for link prediction, this choice is both directly proportional to a graph's branching size and degree assortativity. Nonetheless, the difficulty in analyzing these graphs supports our choice for a semi-supervised approach, i.e. automatically choosing n on a per graph instance.

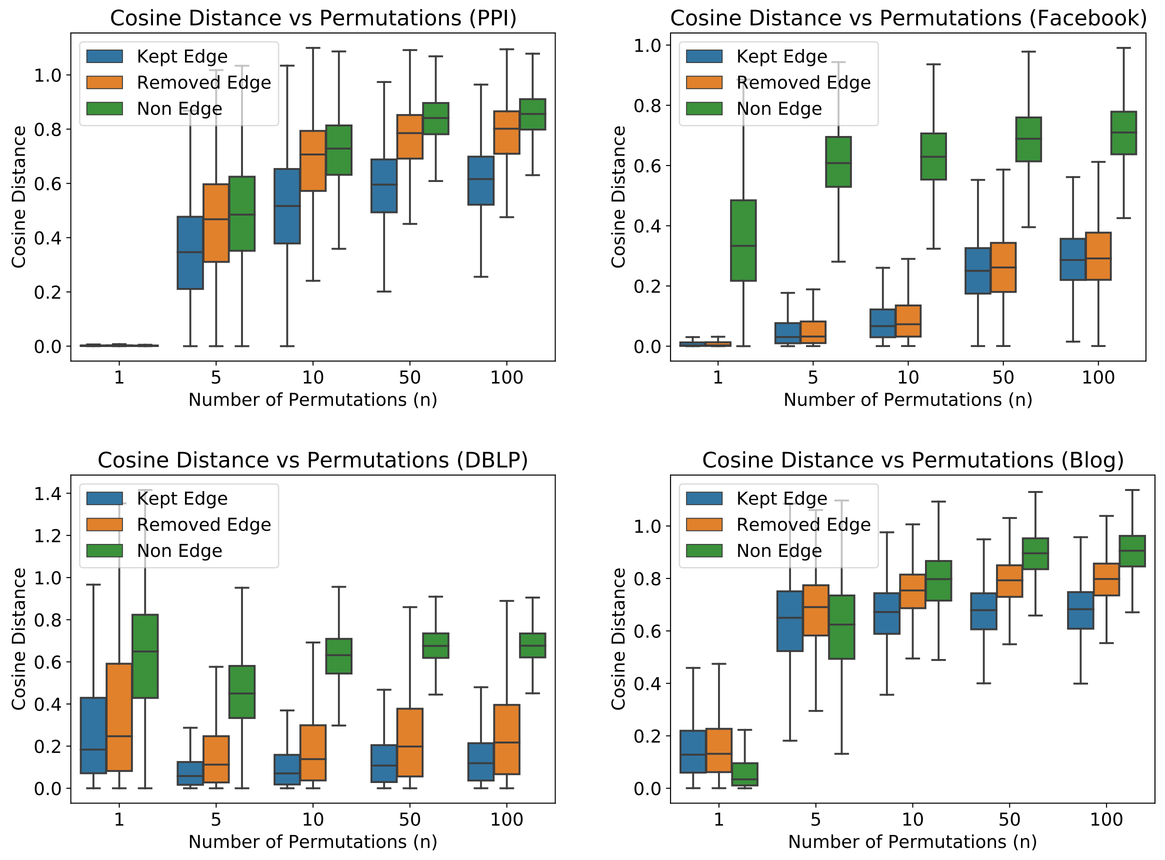


Figure 5.4: (Color online) Cosine distances on the: PPI dataset (top-left); Facebook dataset (top-right); DBLP dataset (bottom-left); Blog dataset (bottom-right). All graphs had 10% of edges removes.

To better understand te results of the experiment on the PPI dataset with $n = 1$, shown in Figure 5.4 (top-left), we present in Figure 5.5 a detail of the euclidean distances between nodes that share or not an edge for this number of permutations. We can see that the distribution of removed edges is a lot closer to the edges used for training than to the non edges.

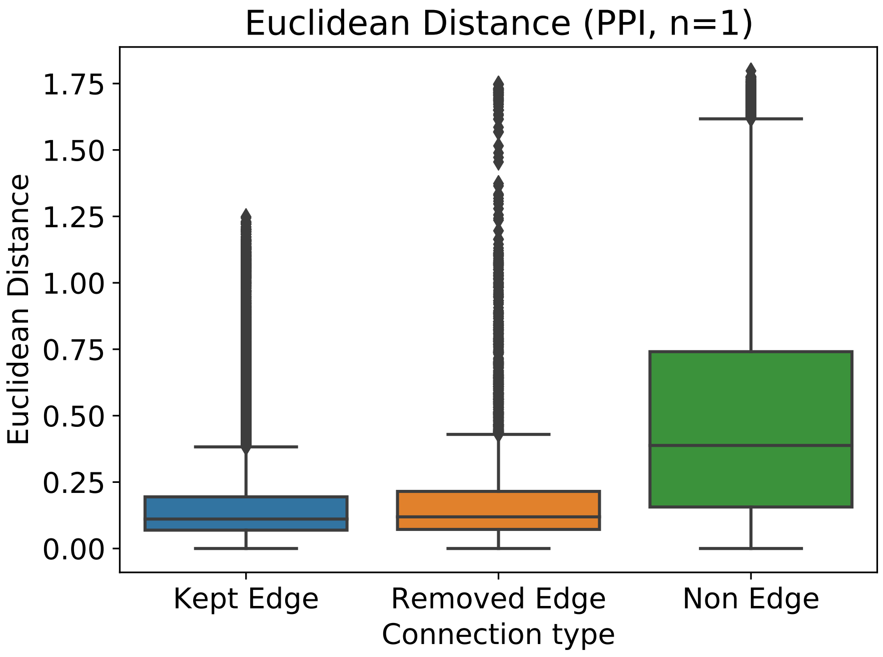


Figure 5.5: (Color online) Euclidean distances on PPI dataset for $n = 1$.

5.3.3 Trade-off between first and second order proximity

The group size, k, controls a trade-off between first and second order proximities in the node embeddings. This can be explained intuitively by analyzing both the group sampling method in Algorithm 2 and Equations 3.1, 3.2 and 3.3, in Section 3.2.

When a smaller k is chosen, $k = 1$ for example, each node's embedding r_{v_i} will only predict its own direct neighbors. This causes nodes with shared neighbors to have closer representations (second order proximity). When larger values of k are chosen, multiple nodes will appear in a group, and will predict its two hop neighbors. This will result in connected nodes having more similar embeddings, increasing first order similarity.

We further analyze this by examining the distribution of cosine distances between nodes at different graph distances. For this analysis, we use synthetic graphs generated from three different network models: Barabási-Albert [Barabási and Albert, 1999]; Erdős-Rényi [Erdos and Rényi, 1960]; Watts-Strogatz [Watts and Strogatz, 1998]. We choose these models because of their structural differences, believing they cover an ample spectrum of different graphs' properties. These graphs were created with $|V| = 2000$ and $b = 20$, and Watts-Strogatz graphs had a probability $\beta = 0.2$ of generating non-lattice edges. To train our representations we used $n = 10$ and $d = 128$.

Figure 5.6 shows box plots of the cosine distances of nodes' representations grouped according to their graph distance on these different synthetic random graphs. In this figure, we can see that, for both Barabàsi-Albert and Erdős-Rényi graphs, when using a group size (k) equal to 1, the cosine similarity is larger for nodes which are two steps away than for nodes which share an edge (it favors second order proximity), while for larger values of k, nodes which share an edge have larger similarity (it favors first order proximity).

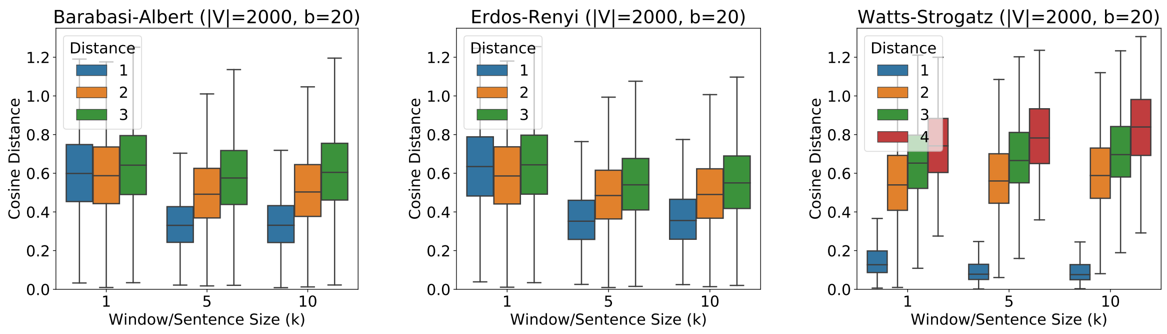


Figure 5.6: (Color online) NBNE features cosine similarities between nodes versus Graph Distance for different values of k for the graphs Barabási-Albert (left), Erdős-Rényi (middle) and Watts-Strogatz (right).

The box plots in Figure 5.6 also show that the difference in similarity increases with the value of k. The larger the value of k, the larger the difference between similarities of nodes which share an edge and nodes with larger distances, as can be seen in detail in Figure 5.7 for the Barabási-Albert model.

5.4 Graph Size and Sparseness Analysis

In this section, we analyze how a graph's sparseness (represented here by its branching factor) and size (represented here by its number of vertices) affect the choice of the number of permutations (n) and of the group size (k). With this purpose we ran several link prediction experiments on synthetic graphs generated from two different network models: Watts-Stogratz and Barabási-Albert.[1] These graphs were generated for different sizes ($|V|$) and sparseness (b), and we ran experiments with the same setup as in Section 4.2, once again setting $\beta = 0.2$ for Watts-Stogratz model.[2] Section 5.4.1

[1] Erdős-Rényi graphs weren't analyzed in this section because, since they have a completely random structure, its removed edges would be unpredictable.

[2]Results presented in this section are all averages of ten cross-validation executions in a single instance of each graph size.

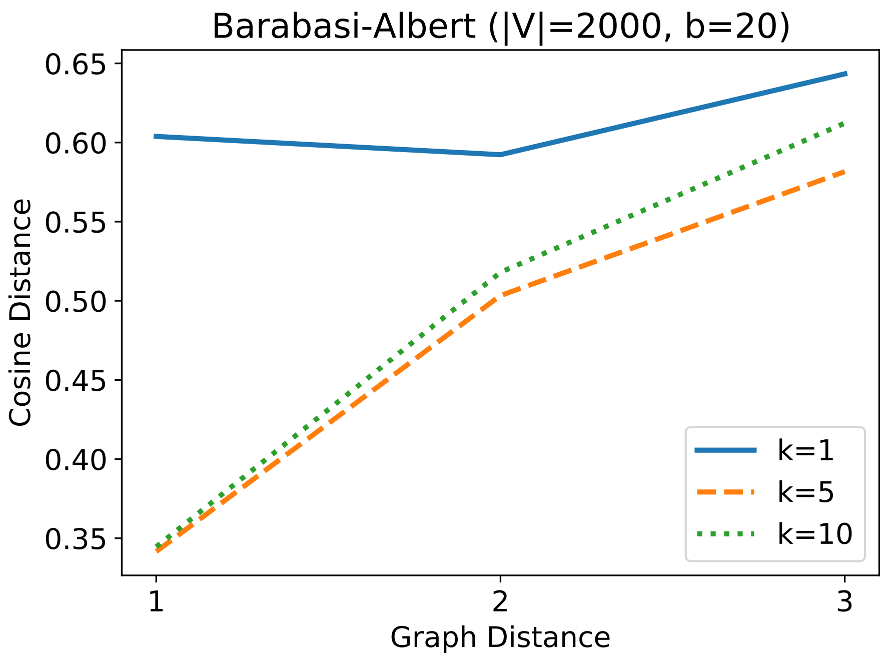


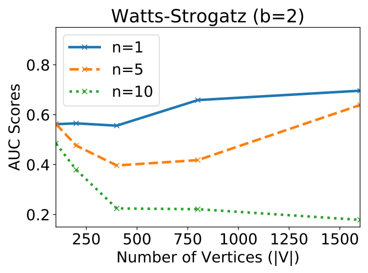
Figure 5.7: (Color online) NBNE cosine vs graph distances in the Barabási-Albert graph.

presents this analysis for the number of permutations (n) and Section 5.4.2 contains the analysis for different group sizes (k).

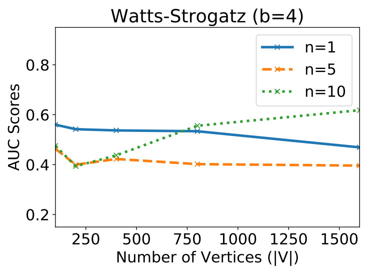
5.4.1 Number of Permutations (n)

In this section, we analyze how the graph's size and sparseness affect the choice of the number of permutations (n), for both Watts-Stogratz and Barabási-Albert graphs. Analyzing the graphs in Figure 5.8, we see a correlation between the best choice of n and a graph's number of vertices ($|V|$) and branching factor (b). In Figure 5.8a, which depicts the experiments in the sparsest graphs ($b = 2$), results for $n = 1$ are better for all graph sizes. A random algorithm would return an AUC score of 0.5, so results below this value clearly flag a problem in the learning algorithm. This is the case for both $n = 10$ and $n = 5$ in these graphs, which overfit its representations.

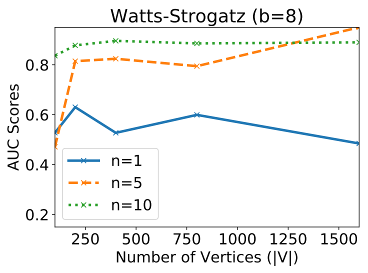
In Figure 5.8b we can see that, when considering a graph with a branching size of $b = 4$, for smaller graphs a smaller value of n is preferable, while for larger graphs a larger number of permutations gives better results ($n = 10$). In Figure5.8c we can see that, for a branching size of $b = 8$, results for larger values of n are always better than for $n = 1$. Notice also that, while results for $b = 2$ and $b = 4$ were around $0.55 \sim 0.7$,



(a) $b = 2$



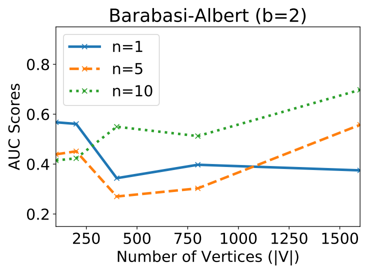
(b) $b = 4$



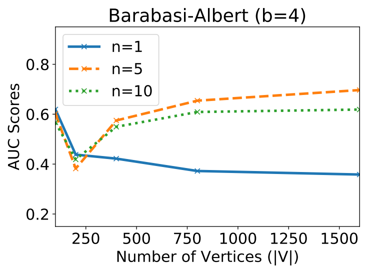
(c) $b = 8$

Figure 5.8: (Color online) AUC Score vs Number of Vertices on a link prediction task on synthetic Watts-Strogatz graphs for different values of n.

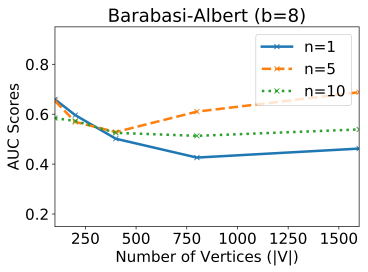
results for $b = 8$ are closer to 0.9, showing that this algorithm is better at learning with more information.



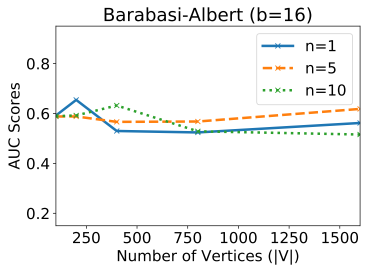
(a) $b = 2$



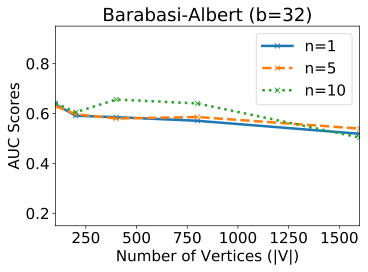
(b) $b = 4$



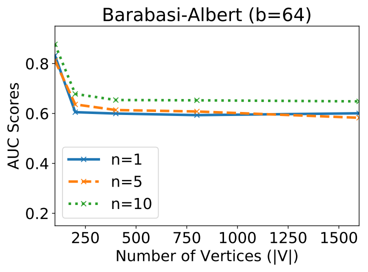
(c) $b = 8$



(d) $b = 16$



(e) $b = 32$



(f) $b = 64$

Figure 5.9: (Color online) AUC Score vs Number of Vertices on a link prediction task for synthetic Barabási-Albert graphs for different values of n.

Our experiments in link prediction using the Barabási-Albert models present slightly more complex results. Figure 5.9 shows that for smaller branching factors ($b \leq 8$), $n = 1$ indeed generate better results for small graphs, but for larger graphs, a larger number of permutations is necessary. For intermediary branch sizes the best value of n is harder to determine, and only for $b = 64$ we start to see a tendency of larger number of permutations consistently yielding better results.

We can also see from Figure 5.9 that edges in Barabási-Albert graphs are considerably more difficult to predict, specially for smaller branching sizes. Most of our results are around 60% and our best AUC scores in these graphs are around 70%.

Again, n's dependency on these graph properties ($|V|$ and b) depends highly on the graph's structure, further supporting our choice of a semi-supervised approach, choosing n on a per graph basis by validating results on a small validation set. This can be considered as a form of early stopping when training these node embeddings.

5.4.2 Groups Sizes (k)

In this section, we use the Watts-Stogratz and Barabási-Albert models once again, this time to analyze how the graph's size and sparseness affect results for different groups' sizes (k) in our model. For these experiments we keep $n = 5$ fixed.

Figure 5.10a shows that, for a small branching factor ($b = 2$), all choices of k clearly overfit for Watts-Strogatz graphs, but $k = 5$ overfits less than larger choices of k. For $b = 8$, $k = 5$ produces slightly better results in these graphs, while larger values of k produce better results for a larger branching size ($b = 32$).

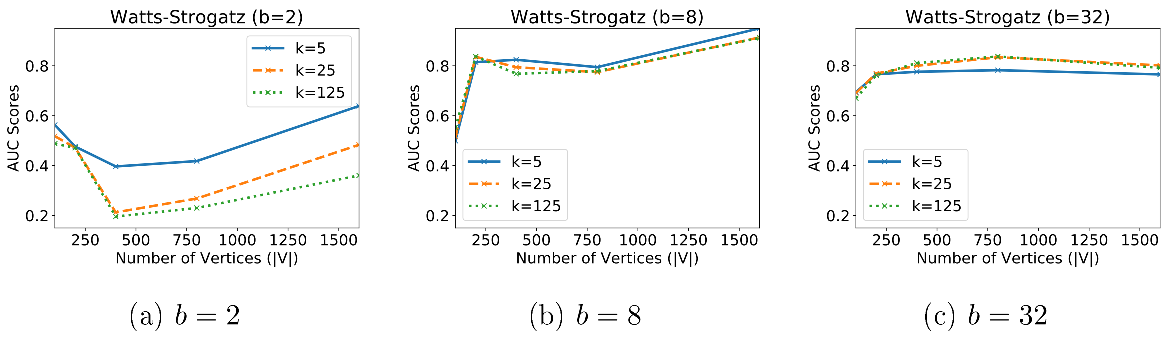
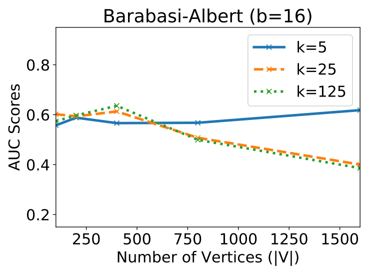


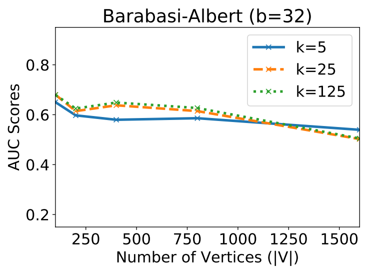
Figure 5.10: (Color online) AUC Score vs Number of Vertices on a link prediction task for synthetic Watts-Strogatz graphs for different values of k.

Barabási-Albert graphs' edges are considerably harder for our algorithm to predict, as shown in the previous section, so we only report results for larger values of b (the algorithm, with our choice of hyper-parameters, overfits for smaller values). We can see from Figure 5.11 that larger values of k usually produce better results for this graph, but are more prone to overfit, especially when being applied to larger sparse graphs ($|V| \geq 800$ and $b = 16$).

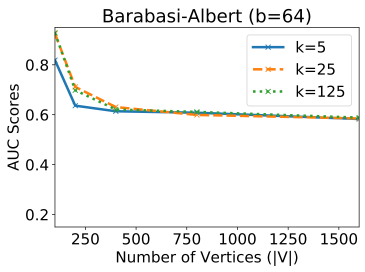
Further analysis on the representations' properties for different values of k could provide better motivation on its choice, but we leave this to future studies, keeping our choice of $k = 5$ constant in this work. Studying if algebraic operations between



(a) $b = 16$



(b) $b = 32$



(c) $b = 64$

Figure 5.11: (Color online) AUC Score vs Number of Vertices on a link prediction task for synthetic Barabási-Albert graphs for different values of k.

representations have comprehensible meanings would also be interesting, such as was done for Word2Vec algorithms, but this is also left as future work.

Chapter 6

Conclusions

The proposed node embedding method NBNE shows results similar or better than the state-of-the-art algorithms Node2Vec and DeepWalk on several different datasets. It shows promising results in two application scenarios: link prediction and node classification, while being efficient and easy to compute for large graphs, differently from other node embedding algorithms, such as LINE [Tang et al., 2015] or SDNE [Wang et al., 2016].

NBNE focuses learning on node's immediate neighbors, creating more ego-centric representations, which we suspect makes them more stable and faster to learn. Empirical results show that, although it has a similar time complexity, NBNE can be trained in a fraction of the time taken by DeepWalk (10 to 190 times faster) or Node2Vec (200 to 6,000 times faster), giving fairly robust results. Since embeddings are learned using only a node's immediate neighbors, we suspect it is also easier to implement more stable asynchronous distributed algorithms to train them, and we leave this as future work.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems.

Adamic, L. A. and Adar, E. (2003). Friends and neighbors on the web. *Social Networks*, 25(3):211--230.

Al Hasan, M., Chaoji, V., Salem, S., and Zaki, M. (2006). Link prediction using supervised learning. In *SDM'06: Workshop on Link Analysis, Counter-terrorism and Security*.

Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. (2016). Learning to compose neural networks for question answering. *NAACL-HLT*, pages 1545--1554.

Baeza-Yates, R. A. and Ribeiro-Neto, B. A. (2011). *Modern Information Retrieval - the concepts and technology behind search*. Pearson Education Ltd., Harlow, England.

Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286(5439):509--512.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3(2):1137--1155.

Breitkreutz, B., Stark, C., Reguly, T., Boucher, L., Breitkreutz, A., Livstone, M. S., Oughtred, R., Lackner, D. H., Bähler, J., Wood, V., Dolinski, K., and Tyers, M. (2008). The biogrid interaction database: 2008 update. *Nucleic Acids Research*, 36(Database-Issue):637--640.

Coppersmith, D. and Winograd, S. (1987). Matrix multiplication via arithmetic progressions. In *STOC*, pages 1--6.

Dong, Y., Chawla, N. V., and Swami, A. (2017). metapath2vec: Scalable representation learning for heterogeneous networks. In *KDD*, pages 135--144.

Edunov, S., Diuk, C., Filiz, I. O., Bhagat, S., and Burke, M. (2016). Three and a half degrees of separation. *Research at Facebook*.

Erdos, P. and Rényi, A. (1960). On the evolution of random graphs. *Publicationes Mathematicae*, 5(1):17--60.

Ferreira, A. A., Gonçalves, M. A., and Laender, A. H. (2015). Automatic methods for disambiguating author names in bibliographic data repositories. In *JCDL*, pages 297--298. ACM.

Ferreira, A. A., Silva, R. M., Gonçalves, M. A., Veloso, A., and Laender, A. H. F. (2012). Active associative sampling for author name disambiguation. In *JCDL*, pages 175--184.

Firat, O., Cho, K., and Bengio, Y. (2016). Multi-way, multilingual neural machine translation with a shared attention mechanism. *NAACL-HLT*, pages 866--875.

Firth, J. R. (1957). A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*.

Grover, A. and Leskovec, J. (2016). Node2vec: Scalable feature learning for networks. In *KDD*, pages 855--864.

Hamilton, W., Ying, Z., and Leskovec, J. (2017). Inductive representation learning on large graphs. In *NIPS*, pages 1025--1035.

Jensen, D. and Neville, J. (2002). Linkage and autocorrelation cause feature selection bias in relational learning. In *ICML*, volume 2, pages 259--266.

Leskovec, J., Kleinberg, J. M., and Faloutsos, C. (2007). Graph evolution: Densification and shrinking diameters. *Transactions on Knowledge Discovery from Data*, 1(1):2.

Liberzon, A., Subramanian, A., Pinchback, R., Thorvaldsdóttir, H., Tamayo, P., and Mesirov, J. P. (2011). Molecular signatures database 3.0. *Bioinformatics*, 27(12):1739--1740.

Lin, Y., Liu, Z., and Sun, M. (2016). Knowledge representation learning with entities, attributes and relations. In *IJCAI*, pages 2866--2872.

Lü, L. and Zhou, T. (2011). Link prediction in complex networks: A survey. *Physica A: Statistical Mechanics and its Applications*, 390(6):1150--1170.

Luong, M.-T. and Manning, C. D. (2016). Achieving open vocabulary neural machine translation with hybrid word-character models. *ACL*, pages 1054--1063.

Mahoney, M. (2011). Large text compression benchmark.

McAuley, J. J. and Leskovec, J. (2012). Learning to discover social circles in ego networks. In *NIPS*, volume 2012, pages 548--56.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781.*

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111--3119.

Neville, J. and Jensen, D. (2007). Relational dependency networks. *Journal of Machine Learning Research*, 8(Mar):653--692.

Newman, M. E. (2003). Mixing patterns in networks. *Physical Review E*, 67(2):026126.

Pan, S., Wu, J., Zhu, X., Zhang, C., and Wang, Y. (2016). Tri-party deep network representation. In *IJCAI*, pages 1895--1901.

Park, J. and Barabási, A.-L. (2007). Distribution of node characteristics in complex networks. *National Academy of Sciences*, 104(46):17916--17920.

Perozzi, B., Al-Rfou, R., and Skiena, S. (2014). Deepwalk: Online learning of social representations. In *KDD*, pages 701--710.

Radivojac, P., Clark, W. T., Oron, T. R., Schnoes, A. M., Wittkop, T., Sokolov, A., Graim, K., Funk, C., Verspoor, K., Ben-Hur, A., et al. (2013). A large-scale evaluation of computational protein function prediction. *Nature methods*, 10(3):221--227.

Řehůřek, R. and Sojka, P. (2010). Software Framework for Topic Modelling with Large Corpora. In *LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45--50.

Roweis, S. T. and Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323--2326.

Santana, A. F., Gonçalves, M. A., Laender, A. H. F., and Ferreira, A. A. (2014). Combining domain-specific heuristics for author name disambiguation. In *JCDL*, pages 173--182.

Sarkar, P., Chakrabarti, D., and Moore, A. W. (2011). Theoretical justification of popular link prediction heuristics. In *IJCAI*, volume 22, page 2722.

Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., and Mei, Q. (2015). Line: Large-scale information network embedding. In *WWW*, pages 1067--1077.

Tenenbaum, J. B., De Silva, V., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319--2323.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. (2018). Graph attention networks. *ICLR*.

Wang, D., Cui, P., and Zhu, W. (2016). Structural deep network embedding. In *KDD*, pages 1225--1234.

Wang, Z. and Li, J. (2016). Text-enhanced representation learning for knowledge graph. In *IJCAI*, pages 1293--1299.

Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440--442.

Wen, T.-H., Gasic, M., Mrksic, N., Su, P.-H., Vandyke, D., and Young, S. (2015). Semantically conditioned lstm-based natural language generation for spoken dialogue systems. *EMNLP*, pages 1711--1721.

Yang, J. and Leskovec, J. (2012). Defining and evaluating network communities based on ground-truth. In *ICDM*, pages 745--754.

Yang, S.-H., Long, B., Smola, A., Sadagopan, N., Zheng, Z., and Zha, H. (2011). Like like alike: joint friendship and interest propagation in social networks. In *Proceedings of the 20th international conference on World wide web*, pages 537--546.

Zafarani, R. and Liu, H. (2009). Social computing data repository at ASU.