FastEMB: Faster Succinct and Accurate Node Embedding of Large Graphs

ABSTRACT

We consider the problem of embedding all nodes of a very large network in a low-dimensional space. Such low-dimensional representations of graphs are very useful for node classification, link prediction, recommendation, and community detection. Existing node embedding algorithms usually involve solving an optimization problem to "learn" an optimal embedding and suffer from scalability worries when given as input massively sized graphs. In this work, we propose a novel embedding algorithm named FastEMB that generates random low-dimensional binary vectors as the embedding of nodes. Our algorithm relies on the recently proposed BinSketch algorithm [22] for sparse binary data. We also present an algorithm to estimate important properties of the graph required for common machine learning tasks from these vectors.

We compare the performance of FastEMB on the tasks of node classification and link prediction with several state-of-the-art algorithms for the said purpose, such as node2vec, DeepWalk, and LINE on eight real-life datasets. We observed significant speedup in terms of embedding computation time which is the key strength of FastEMB. At the same time, we obtained more accurate results compared to the state-of-the-art algorithms. Our proposed binary embedding is highly efficient in terms of space usage as compared to real-valued vectors obtained by other candidate algorithms. For the link prediction task on the Gowalla dataset, the speedup was 731 times as compared to node2vec, 661 times as compared to DeepWalk, and 6751 times as compared to LINE, while simultaneously offering about 8% improvement in accuracy and taking 3.72 times lesser space. Our technique is adaptable for streaming and dynamic graphs with addition of edges. We provide extensive theoretical analysis, giving bounds on accuracy and error wherever possible, to explain the behaviour of FastEMB. Given the simplicity of our method, we hope that it can be easily adopted in practice.

KEYWORDS

Dimensionality Reduction, Sketching, Node Embedding, Social Network, Link Prediction.

ACM Reference Format:

. 2019. FastEMB: Faster Succinct and Accurate Node Embedding of Large Graphs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Graph is a fundamental and ubiquitous data structure that is used extensively in several applications within computer science and related domains. Specifically, graphs can be used in situations where the interactions need to be captured between individual units – units are modeled as nodes, and interactions between them are modeled as edges. Graphs are used to model social networks, biological protein-protein networks, molecular graph structures, recommender systems, etc. Besides modeling, they are also crucially used for the task of inferencing, prediction, and for discovering new patterns using techniques from machine learning. For example, to recommend new friends in a social network [1], classify the role of a protein in an interaction graph [10], classify posts in a social network [1], forecast the future traffic speed in a road network [18].

A fundamental challenge in the above applications is to come up with a way to incorporate the information about the structure of a graph to a machine learning model. This is done *via* computing low-dimensional vector embedding of the nodes of a graph such that they maintain the structural or geometric relationship of the graph in low-dimensional space. This node embedding is then further fed into downstream machine learning algorithms for tasks such as node classification, clustering, link prediction. In this work, we consider the problem of computing a low-dimensional embedding of graphs such that the embedding inherits neighborhood property of the original graph.

A number of challenges have been uncovered in the last few decades that have seen a lot of activity on efficient and effective node embedding [11].

It is desirable of a node embedding technique to have a sound theoretical bedrock that is independent of the end goal of classification or link prediction. The theoretical discussions found in most articles are often limited to a particular optimization problem whose semantics is unclear beyond a specific machine learning task. Furthermore, the embedding is often obtained by solving an optimization problem (say, using gradient descent) and thus lacks any manner of worst-case guarantees.

An embedding is fed as input to a machine learning algorithm, and is therefore, required to be extremely fast so as to not become the bottleneck. As the applications move towards trillion sized graphs, it would be better to have approaches that favour distributed computing; however, learning-based embedding approaches do not typically fall into this league. Recently a few GPU-based techniques are proposed to cater these requirements, but, we are interested in vanilla CPU-only approaches.

The space complexity of generating and storing embedding of all nodes (to give as input to a machine learning task) could be significant for massive graphs since most existing approaches generate the embedding of an entire graph at once. The need of the hour are approaches that enable distributed storage of embedding and are

able to handle evolving graphs in which embedding of nodes with new edges can be updated and stored in a cheaper manner.

Semantics of the embedded representation is a big challenge since many proposed techniques are evaluated using statistical methods on standard datasets which fails to bring out the rich structural invariants that these techniques might be capturing. A few approaches, like deepwalk that capture short random walks [21] and GraRep that capture powers of an adjacency matrix [5], explicitly consider certain structural aspects but it is unclear if and how their embeddings fare on other structural properties of a graph.

Finally, randomization has proved to be highly effective time and again; yet, barring a few approaches based on random walks [9, 21], randomized approaches are missing in this domain.

1.1 Our contribution

In this work, we present FastEMB which takes a large graph as input and outputs binary embedding corresponding to each node. It uses a randomized hashing idea named BinSketch that was recently proposed to compress large vectors [22]. FastEMB takes an $n \times n$ adjacency matrix or edge list of an n-node graph as input and outputs a d-dimensional binary embedding for each node where d is decided based upon the *sparsity* of each row of the adjacency matrix. We define sparsity of a vector as the number of ones; it is common for real-life graphs, like the ones we considered for experiments, to have a sparsity much lower compared to n. We also present an EstCN algorithm to estimate the number of common neighbors between any two nodes with worst-case bounds.

FastEMB incorporates some of the features we outlined earlier and has a few other appealing properties that gives it an edge over the existing techniques, all of which we briefly touch upon now.

Theoretical bounds and semantics of embedding: We show that the embedding, even though compressed, retains information about the graph structure like the number of common neighbors, the number of even length paths, etc. More generally, we discuss how any even power of the adjacency matrix can be approximately computed from the embedding. Not surprisingly, our embedding allowed us to perform link prediction and node classification better than others; we conjecture that the FastEMB embedding would be highly suitable for machine learning tasks that rely on structural graph properties. For all the approximation results mentioned above we bound the additive inaccuracy and the probability of error. We empirically validate this on the task of *link prediction* and *node classification* and noticed that FastEMB generally outperforms other candidate algorithms by a margin of $5-7\%$.

Fast embedding: FastEMB takes an adjacency matrix or an edge list of a graph as input and outputs binary embeddings of each node using just one pass over the graph and doing bit manipulations. Due to its simplicity it computes the embedding in almost real time. For example, on the Flickr [28] dataset which has 80513 nodes and 5899882 edges, FastEMB computes its embedding within 12 secs (for $d = 100$) to 23 secs (for $d = 4000$) on a standard server (see Section 4 for configuration); in contrast, node2vec did not finish within 24 hours and LINE took 100 mins for a single epoch.

Succinct representation: FastEMB generates binary embeddings consuming d bits for every node. On the other hand, several state-of-the-art embedding algorithms output real-valued embedding – generally 128 or 256 dimensional real valued vectors. For example, storing 128 dimensional real valued vector requires $128 \times 64 = 8,192$ bits which is nearly 4 times more compared to a FastEMB embedding with $d = 2200$.

Require less training data: We empirically verified that FastEMB offers comparable performance even on less training data. We ran FastEMB on various splits of training and test partition and notice that its performance remains comparable whereas the performances of other candidate algorithms deteriorate due to the decrease in training data.

Handling streaming and evolving graphs: An advantage of FastEMB is that it can be easily adapted for scenarios where edges are getting added to the graph dynamically, e.g., for streaming applications and evolving graphs. On the arrival of a new edge, say between nodes i and j, FastEMB has to only update the embeddings of i and j and this can be done independently of the rest of the graph. To the best of our understanding, most other embedding algorithms require solving an optimization problem afresh and it is unclear if they can do any better than running their entire algorithm on the updated graph. We also noticed empirically that the embedding computation time of FastEMB remains almost constant upon inserting of new edges, whereas, for other candidate algorithms their running times increase linearly with the addition of edges.

Distributed setting: Both our embedding computation algorithm (FastEMB) as well as the similarity estimating algorithm (EstCN) can be easily computed using a distributed system. The embedding of each node can also be split up across a distributed storage system, if necessary. Once again, this is due to the simple and bitwise operations involved in those algorithms.

1.2 Potential applications of FastEMB

In the following, we list three fundamental applications of node embedding algorithms.

Link prediction. Link prediction is one of the fundamental applications of node embedding. In this problem, the aim is to predict the missing edges or the edges that are likely to form in the future. Link prediction is at the core of many machine learning applications such as predicting missing friendship links in social networks [27], predicting links in biological interaction graphs such as proteins-proteins interaction graphs, drugs and disease interaction graphs [19]. Node embedding is also useful in building recommendation systems for *e.g.* by affinities between users and movies [29].

Node classification. In node classification, labels are available for only a small portion of nodes, and the aim is to predict the labels of the remaining nodes from an available small initial seed set of labels. Node classification is arguably the most common application of node embeddings and is used extensively in applications such as classifying documents, videos, web pages into different categories [15, 21, 27] and classifying proteins according to their biological function [9].

Node clustering. In the node clustering problem, the aim is to partition the graph into a set of clusters (sub-graphs) such that nodes in the same cluster are more similar to each other than those from other clusters. In a social network, such clusters are called communities such as groups of users that belong to similar affiliations or having similar interests. As we have the embedding of nodes, it is possible to run generic clustering algorithms such as k-means [2] or DBSCAN [7] and obtain the underlying clusters. This approach could potentially be a simple, accurate and powerful alternative to standard community detection techniques. This is due to the fact that node embeddings correctly captures the underlying geometric structure between nodes, and clustering groups similar nodes together.

We empirically verify the applicability of our node embedding algorithm FastEMB on the task of *link prediction* and *node classification* on several real-world datasets. We obtain significant speedup along with better performance while simultaneously achieving succinct embedding as compared to the state-of-the-art algorithms for the purpose such as LINE [27], node2vec [9], and deepwalk [21]. We discuss this in detail Section 4.

2 BACKGROUND

Problem formulation: Let $G = \langle V, E \rangle$ be an unweighted undirected graph with n nodes and e edges. The graph can also be represented by the adjacency matrix $A \in \{0,1\}^{n \times n}$, where $A_{i,j} = 1$ implies that there is an edge between node i and node j, and is set to 0 otherwise. We use A_i to denote the i-th row of an adjacency matrix A; this $|V|$-dimensional binary vector represents the adjacency vector of node i. The aim is to compute the binary embedding of the nodes of the graph $U \in \{0,1\}^{n \times d}$,where $d(d \ll n)$ is the dimension of the embedding.

It is common for embedding algorithms to try to preserve local graph structures.

First-order proximity. The first-order proximity between nodes i and j, denoted $s_{ij}^{(1)}$, captures presence of an edge between them and is set to A_{ij} for unweighted graphs.

Second and higher order proximities. We will give an inductive definition. Let $s_i^{(k)} = [s_{i1}^{(k)}, s_{i2}^{(k)}, \ldots, s_{in}^{(k)}]$ denote the k-th order proximity between nodes i and the other nodes. Then the $k+1$-th order proximity $s_{ij}^{(k+1)}$ between nodes i and j is a similarity between $s_i^{(k)}$ and $s_j^{(k)}$. A common practice is to use the inner product for computing similarity which intuitively captures the number of common nodes as the similarity.

It is usually not the case that ensuring k-th order proximity automatically leads to $k+1$-th order proximity; hence, different techniques target proximities of specific orders.

2.1 Related Work

Node embedding techniques can be broadly classified into the following three categories: (1)Factorization based, (2) Random Walk based, and (3) Deep Learning based. We discuss them briefly as follows. For details of these techniques, we refer the reader to the survey papers [8, 11].

Factorization based methods. Matrix factorization based graph embedding represent graph property (e.g., node pairwise similarity) in the form of a matrix and factorize this matrix to obtain node embedding. The problem of graph embedding is treated as a structure-preserving dimensionality reduction problem which assumes that the input data lies in a low dimensional manifold. The underlying matrix that is used to represent the connections include adjacency matrix, Laplacian matrix, node transition probability matrix etc, [8].

Random walk based methods. In this case, the graph is explored using different exploration and then random walks obtained are fed as contextual information into the skip-gram model [20]. deepwalk preserves higher-order proximity between nodes and was the first paper to initiate this idea. It uses DFS like search strategy to generate random walks. Similar to deepwalk, node2vec preserves higher-order proximity between nodes. An important difference between these two is that node2vec employs biased random walk that provide a trade off between BFS and DFS exploration strategies. Choosing the right balance between these two enables node2vec to preserve *community structure* and *structural equivalence* between nodes.

Deep learning based methods. Deep learning based methods employed in graphs to find low-dimensional embedding of nodes. Factorization based methods and Random walk based methods have been also attempted using deep learning by formulating the task into an optimization problem and using gradient based methods to get a solution. In another line of research, the problem of learning node embedding is done using various deep learning models such as autoencoder and CNN. Deep auto-encoders have been widely used for dimensionality reduction [3] due to their ability to model non-linear structure in the data. SDNE [30], and DNGR [6] exploits this ability of deep autoencoder and learn embedding of the graph that captures non-linear structure in graphs. Recently, several convolution neural network architectures for learning over graphs have been proposed to learn the embedding of nodes. They formulate convolution-like operation on in the spectral domain [4, 13] and spatial domain [14, 24].

There is some recent body of research which generates binary embedding of nodes. We compare and contrast our results with a few notable results in this direction follows. DNE (Discrete Network Embedding) [26]learns binary node representations to speed up node classification. They propose to jointly learn the discrete embedding and classifier within a unified framework to improve the compactness as well as discrimination property of network embedding. It is a supervised binary network embedding algorithm that requires node labels to be provided. In contrast, FastEMB is unsupervised and is not task-specific and generates embedding which can be used in other evaluation tasks such as Link prediction, community detection, etc.

BANE [32] suggests binary embedding of attribute network by joint representation learning of node links and attributes. They formulate the problem into *mixed-integer optimization* and using *cyclic coordinate descent* (CCD) learn the embedding of nodes. In contrast, FastEMB computes embedding of graph considering only edges, and our technique is algorithmic, directly computes the binary embedding of nodes.

Our approach follows a quite different trajectory compared to the above approaches. Those approaches try to *learn* the optimal embedding that minimizes a difference function; the difference function captures the desideratum of accurately obtaining some graph property (e.g., neighborhood of a node) or the effectiveness of the embedding for a certain machine learning task. FastEMB deviates from the norm and generates a *random* embedding upfront. It leaves the heavy duty of using it effectively to estimate graph properties to the EstCN estimation algorithm.

3 FASTEMB: FAST EMBEDDING OF NODES

In this section we explain our embedding technique "FastEMB", discuss the properties of a graph it seeks to preserve and explain a few attractive features that make it suitable for embedding graphs, especially for link prediction and node classification.

The idea behind FastEMB has its seed in the BinSketch hashing technique and an algorithm to estimate inner products of binary vectors from their hashes [22]. However, we heavily modify it to obtain important theoretical guarantees.

*Definition 1 (*FastEMB *(proposed as* BinSketch *earlier [22])).* Let π denote a random mapping from $\{1, 2, 3, \ldots, n\}$ to $\{1, 2, 3, \ldots, d\}$. The embedding of a node i is a d-dimensional binary vector which is denoted σ_i and is defined as

$$j\text{-th coordinate of } \sigma_i \colon = (\sigma_i)_j = \bigvee_{k:\pi(k)=j} A_{ik}$$

Algorithm 1 FastEMB embedding all nodes of a graph

Input: undirected unweighted raph $G = \langle V, E\rangle$

Parameter: embedding dimension $d = \psi^2\sqrt{\frac{\psi}{2}\ln\frac{2}{\rho}}$

1: $\pi \leftarrow$ a random mapping from $\{1, \ldots, |V|\}$ to $\{1, \ldots, d\}$

2: For all $i = 1 \ldots |V|$, initialize σ_i as the vector $\overbrace{00\ldots0}^{d}$

3: Set $L \leftarrow$ list of edges E

4: **for all** edge $(i, k) \in L$ **do**

5: Determine $j = \pi(k)$

6: Set the j-th bit of σ_i to 1

7: **end for**

It is immensely easy to generate the embedding of all nodes of G by simply processing all its edges one by one and following Algorithm 1.

FastEMB falls under the "direct encoding approaches" in a recent categorization of graph representation methods [11]. The "encoding" of a node i, represented by its adjacency vector A_i, is given by the matrix-vector product $P \cdot A_i^T$ where P is a $d \times n$ matrix and $\cdot$ is a matrix product. For FastEMB we chose P as a random sparse binary matrix with n ones such that there is exactly one 1 in any column; for the matrix product we used a Boolean-logic equivalent of standard matrix product, i.e., the i-th entry of the product of a binary matrix M and a binary vector is computed as $\bigvee_{k=1}^{n}(M_{ik} \wedge v_k)$. An example illustrating this notion is given in Figure 1.

First modification to BinSketch that we make is to use a larger embedding dimension; we set $d = \psi^2\sqrt{\frac{\psi}{2}\ln\frac{2}{\rho}}$ where ψ is an upper bound on the degree of any node.

$$\pi : \begin{bmatrix} 1 \to 1 & 2 \to 3 \\ 3 \to 2 & 4 \to 2 \\ 5 \to 3 & 6 \to 2 \end{bmatrix} \qquad \begin{bmatrix} 100000 \\ 001101 \\ 010010 \end{bmatrix} \cdot [010110]^T = [011]^T$$

Figure 1: FastEMB embedding of a node with edges to $2, 4, 5$ **is 011. The mapping** π **is shown on the left.**

3.1 Computational complexity

The random mapping can be generated once and stored as a fast-lookup table or can be implemented using an efficient hashing algorithm. Suppose S denotes the space required and T denotes the time-complexity to compute $\pi(\cdot)$; for a lookup table stored in RAM, $S = O(n \log d)$ and $T = O(1)$. Apart from the overhead of π, FastEMB only involves bit manipulations which makes it extremely efficient. For the lemma below we are assuming that G is stored as an adjacency list (which has asymptotically same space complexity as that of an edge-list representation).

Lemma 2. *Algorithm 1 runs in time* $\Theta(T \cdot |E|)$*, returns an embedding using* $d \cdot |V|$ *bits and uses an additional space* S *for generating the embedding.*

We are assuming that the nodes embedding too are stored in RAM; observe that π need not be stored once the embedding of a graph is generated. Another alternative is to not store the embedding, store only π and generate σ_i on demand — this approach is also efficient given the lightweight embedding algorithm.

Lemma 3. *The embedding of a single node, say* i*, uses only* d *bits and can be computed in time* $O(T \cdot n(i))$ *where* $n(i)$ *denotes the number of edges of* i*. The internal space required is only that coming from storing the random mapping* π*.*

The above lemma ensures that our embedding algorithm is update friendly. That is, suppose we have a dynamic graph on n nodes whose embedding $\{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ is already computed, and now, an edge (i, j) is added to the graph. The embedding of the updated graph can be efficiently computed by using $L = \{(i, j),\ (j, i)\}$ in Line 3 of Algorithm 1.

Matrix operations are naturally adaptable to distributed settings and so is the case for FastEMB; embedding of each node can be computed indepedently of the others in a completely parallel and distributed manner. In fact, tabulation hashing techniques can be used to distribute the mapping function as well and one can even distribute the computation on the basis of dimensions.

3.2 Bounding loss function

Now we present a few results that show that the node embedding produced by FastEMB "preserve" similarity of two nodes; we define the *similarity* of two nodes, say i and j, by the number of their common neighbors, denoted $n_{i,j}$. Note that $n_{i,j} \leq \psi$.

Our results follow from the EstCN algorithm, (described in Algorithm 2) that has an important modification in Line 5 to the BinSketch algorithm proposed earlier [22, Algorithm 1]. The latter algorithm was designed to estimate the inner product of two binary vectors from their sketches, and, we adapt it for estimating the number of common neighbors of any two distinct vertices from their

Algorithm 2 EstCN to estimate number of common neighbors $n_{i,j}$

Input: Embeddings σ_i of i and σ_j of j
Parameter: embedding dimension d

1: Compute $\hat{n}_i^s = |\sigma_i|$ ▷ $|v|$: number of ones in vector v
2: Compute $\hat{n}_j^s = |\sigma_j|$
3: Compute $\hat{n}_{i,j}^s = \langle \sigma_i, \sigma_j \rangle$ ▷ $\langle u, v \rangle$: inner prod. of u and v
4: **if** $\hat{n}_{i,j}^s = 0$ **then**
5: **return** $\hat{n}_{i,j} = 0$
6: **end if**
7: Set $\hat{n}_i = \ln(1 - \frac{\hat{n}_i^s}{d})/\ln(D)$ ▷ D denotes $1 - \frac{1}{d}$
8: Set $\hat{n}_j = \ln(1 - \frac{\hat{n}_j^s}{d})/\ln(D)$
9: **return** estimated number of common neighbors

$$\hat{n}_{i,j} = \hat{n}_i + \hat{n}_j - \tfrac{1}{\ln D} \ln\left(D^{\hat{n}_i} + D^{\hat{n}_j} + \frac{\hat{n}_{i,j}^s}{d} - 1\right)$$

embeddings. Adapting a result from BinSketch [22, Theorem 1] for our scenario, we show that the embedding produced by FastEMB can be used, via the EstCN algorithm, to produce a tight estimate of the number of common neighbors.

Theorem 4. *Let ψ be an upper bound on the degree of G and ρ be the desired probability of error. If the embedding dimension d is chosen as $\psi^2\sqrt{\frac{\psi}{2}\ln\frac{2}{\rho}}$ then* EstCN *ensures that its output satisfies the following with probability at least $1-\rho$.*

$$n_{i,j} - 14\sqrt{\frac{\psi}{2}\ln\frac{6}{\rho}} \quad < \hat{n}_{i,j} < \quad n_{i,j} + 14\sqrt{\frac{\psi}{2}\ln\frac{6}{\rho}}$$

Furthermore,

(1) *if $n_{i,j} > 0$ then $\hat{n}_{i,j} > 0$, and*
(2) *if $n_{i,j} = 0$ then $\hat{n}_{i,j} = 0$ with probability at least $1-\sqrt{2/(\psi\ln\frac{2}{\rho})}$.*

The first part about the accuracy of EstCN is known as a feature of the BinSketch algorithm [22, Theorem 1]; however, in that algorithm the embedding dimension was set to $\psi\sqrt{\frac{\psi}{2}\ln\frac{2}{\rho}}$ which has a lower value. Observe that as the embedding dimension is increased, there are even lesser chances of collision in the computation of $\pi(\cdot)$ and higher chances of a particular bit of an encoding being set by only one bit of a source vector. Intuitively speaking, this will only result in better accuracy and lower probability of error in estimation compared to the BinSketch algorithm.

Now we focus on the proof that, with high probability, $n_{i,j} = 0$ iff $\hat{n}_{i,j} = 0$. This is important, since otherwise, the *estimated* number of common neighbors of i and j could be as large as $14\sqrt{\frac{\psi}{2}\ln\frac{6}{\rho}}$ even when i and j share no common neighbor. The enhancement in Line 5 of Algorithm 2 is crucial to prove the second part.

Proof of second part. For the first claim observe that if $n_{i,j} > 0$, then there must be some k such that $A_{ik} = A_{jk} = 1$. Let t denote $\pi(k)$; then both $(\sigma_i)_t = (\sigma_j)_t$ will be set to 1. Thus, $\hat{n}_{i,j}^s = \langle \sigma_i, \sigma_j \rangle > 0$. By way of contradiction, assume that $\hat{n}_{i,j} = 0$. But then we would see

$$\text{(from Algo. 2)} \quad \hat{n}_i + \hat{n}_j = \tfrac{1}{\ln D}\ln\left(D^{\hat{n}_i} + D^{\hat{n}_j} + \frac{\hat{n}_{i,j}^s}{d} - 1\right)$$

$$\equiv \quad D^{\hat{n}_i} \cdot D^{\hat{n}_j} = D^{\hat{n}_i} + D^{\hat{n}_j} + \frac{\hat{n}_{i,j}^s}{d} - 1$$

$$\equiv \quad \left(1 - \frac{\hat{n}_i^s}{d}\right)\left(1 - \frac{\hat{n}_j^s}{d}\right) = \left(1 - \frac{\hat{n}_i^s}{d}\right) + \left(1 - \frac{\hat{n}_j^s}{d}\right) + \frac{\hat{n}_{i,j}^s}{d} - 1$$

$$\equiv \quad \frac{\hat{n}_i^s \cdot \hat{n}_j^s}{d^2} = \frac{\hat{n}_{i,j}^s}{d}$$

The last identity is not true in general. For example, consider a scenario in which $\sigma_i \sim \sigma_j$ with the number of ones in each much less than d; in this case $|\sigma_i| \approx |\sigma_j| \approx \langle \sigma_i, \sigma_j \rangle$ and $|\sigma_i| \ll d$, thus contradicting the identity.

For proving the second claim, take any i and j such that $n_{i,j} = 0$, i.e., there is no k such that both $A_{i,k} = A_{j,k} = 1$. For the sake of contradiction, assume that $\hat{n}_{i,j} > 0$, i.e, there is some t such that $(\sigma_i)_t = (\sigma_j)_t = 1$; this means that there must be some k_1 and k_2 such that $A_{i,k_1} = A_{j,k_2} = 1$ and $\pi(k_1) = \pi(k_2) = t$. The probability for the latter event, denoted E, is same as the probability that in a group of n men and n women, there exists at least one pair with a common birthday, which can be shown to be

$$1 - \sum_{k=1}^{M} \binom{D}{k} k!\, S_{M,k}(D-k)^F / D^{F+M}$$

in which $S_{M,k}$ denotes Stirling's number of the second kind [16]. Getting a closed form of this is difficult, so we show a different technique of bounding the probability of E.

Let E_x be the event that $(\sigma_i)_x = (\sigma_j)_x = 1$. It can be shown that $\mathbb{E}(E_x) = \left(1 - D^{|A_i|}\right) \cdot \left(1 - D^{|A_j|}\right)$ [22, Lemma 5]. Since $|A_i|$ and $|A_j|$ are both at most ψ, and, $D < 1$, therefore,

$$\mathbb{E}(E_x) \le (1 - D^{\psi})^2 = \left(1 - \left(1 - \frac{1}{d}\right)^{\psi}\right)^2 \le \left(\frac{\psi}{d}\right)^2$$

which implies that $\mathbb{E}[\sum_x E_x] = d\left(\frac{\psi}{d}\right)^2 = \frac{\sqrt{2}}{\sqrt{\psi\ln\frac{2}{\rho}}} \ll 1$

Let E_{all} denote $\sum_x E_x$; E_{all} is a random variable that denotes the number of positions x such that $(\sigma_i)_x = (\sigma_j)_x = 1$ and whose expectation we computed above. Since E is equivalent to "$E_{all} \ge 1$", we apply Markov's inequality to derive an upper bound:

$$\Pr[\hat{n}_{i,j} > 0] = \Pr[E] = \Pr[E_{all} \ge 1] \le \mathbb{E}[E_{all}] = \frac{\sqrt{2}}{\sqrt{\psi\ln\frac{2}{\rho}}}.$$

□

For the rest of this section we will use the fact that EstCN estimates non-zero $n_{i,j}$ values with a small additive error and accurately identifies $n_{i,j} = 0$ values (both of these happen with non-negligible probability which we would not explicitly state for the sake of brevity).

The above theorem allows us to accurately quantify the loss function (*aka.* objective function) that is used by most node embedding

algorithm to *learn* the embedding [12]. Using the mean-squared-error (MSE) to compute the loss function, we can represent it as

$$\mathcal{L} = \frac{1}{T} \sum_{(i,j) \in T \subseteq V \times V} |n_{i,j} - \texttt{EstCN}(i,j)|^2.$$

Here T represents a "training set of edges" that is traditionally used to learn an embedding. Our proposed technique does not involve any learning; nevertheless, for the sake of comparison we present an explicit upper bound on $\mathcal{L}$ — it follows directly from Theorem 4.

Lemma 5. *Using* FastEMB *for embedding and* EstCN *for estimating node similarity, $\mathcal{L}$ is upper bounded by $98 \frac{\psi}{2} \ln \frac{6}{\rho}$.*

3.3 Preserving higher-order similarities

Many node embedding algorithm operate on the paths, often up to a certain length, on a graph. For example, both the random-walk based approaches deepwalk and node2vec consider two node to be similar if their presence in short random walks on the graph are highly correlated. Here we show that FastEMB also "preserves" path information in a certain manner. The key observation here is that the square of the adjacency matrix exactly contains the n_{ij} values.

Observation 6. *For any i, j, $A^2_{i,j} = n_{i,j}$.*

This is since $\sum_{k=1}^{n} A_{ik} A_{kj} = |\{k \in V \ : \ (i,k) \in E, \ (k,j) \in E\}|$.

This observation along with Theorem 4 implies that EstCN can approximately compute A^2.

Lemma 7. *$\texttt{EstCN}(\sigma_i, \sigma_j)$ approximately computes $A^2_{i,j}$ with a small additive error. If $A^2_{i,j} = 0$ $\texttt{EstCN}(\sigma_i, \sigma_j)$ outputs 0.*

Next we show how to translate the above lemma to higher *even* powers of A. We first show the result for the 4-th power, A^4, each of whose entry, say $A^4_{i,j}$, denotes the number of paths between i and j using 4 nodes (and 3 edges).

We use ϵ_2 to denote the small additive error mentioned in Lemma 7. To maintain consistency of notations, we will use $\hat{n}2_{i,j}$ to denote $\hat{n}_{i,j}$; by construction, $\hat{n}2$ is symmetric.

Theorem 8. *Let $\hat{n}4_{i,j}$ denote the expression $\sum_{k=1}^{n} \hat{n}2_{i,k} \cdot \hat{n}2_{k,j}$ for all $i, j = 1 \ldots n$. Then $\hat{n}4_{i,j}$ approximately computes $A^4_{i,j}$ with a small additive error. If $A^4_{i,j} = 0$ then $\hat{n}4_{i,j}$ outputs 0.*

Proof. Recall that $A^4_{i,j} = \sum_{k=0}^{n} A^2_{i,k} A^2_{k,j} = \sum_{k=0}^{n} A^2_{i,k} A^2_{j,k}$, and further more, each term in the summation is non-negative. Therefore, $A^4_{i,j} = 0$ implies that $A^2_{i,k} = 0$ and $A^2_{j,k} = 0$ for every $k = 1 \ldots n$. We know from Lemma 7 that, in this case, $\hat{n}2_{i,k} = 0$ and $\hat{n}2_{j,k} = 0$ for all k. Clearly, $\hat{n}4^{i,j} = 0$ too — this proves the second part of the theorem.

For the first part, take any i, j such that $A^4_{i,j} > 0$. That is, $\sum_{k=1}^{n} A^2_{i,k} A^2_{j,k} > 0$. From Lemma 7 we know that, for any y and $x = i, j$, $|\hat{n}2_{x,y} - A^2_{x,y}| \le \epsilon_2$; using z to denote the left-hand side, we can write $\hat{n}2_{x,y} = A^2_{x,y} + z$ where $-\epsilon_2 \le z \le \epsilon_2$ and

$$\hat{n}4_{x,y} = \sum_{u=1}^{n} \hat{n}2_{x,u} \cdot \hat{n}2_{y,u}$$

We will use a technical result about A.

Claim 9. $\sum_{u=1}^{n} A^2_{x,u} \le \psi^2$.

Proof. ψ being an upper bound on the degree of any node, the total number of length-2 paths from x is at most ψ^2. $A^2_{x,u}$ is the total number of length-2 paths from x to u and $\sum_u A^2_{x,u}$ is the total number of length-2 paths from x to any node, which is, therefore, upper bounded by ψ^2. ■

The next observation is applicable to sparse graphs in general but immensely beneficial to graphs where $\psi^2 \ll n$.

Observation 10. *Since the entries of $A^2_{x,u}$ are non-negative, Claim 9 implies that at most ψ^2 entries are non-zero in the x-th row of A^2, i.e, among $A^2_x = \{A^2_{x,u} \ : \ u \in \{1, 2, \ldots, n\}\}$. That is, at least $n - \psi^2$ entries of A^2_x are zero.*

This observation, along with Lemma 7, implies that for any x, at most ψ^2 values in the set $\{\hat{n}2_{x,u} \ : \ u \in \{1, 2, \ldots n\}\}$ are non-zero. We use $(\sum_{u=1}^{n})^{\le \psi^2}$ to denote the fact that in a summation with n summands, at most ψ^2 terms are non-zero. Getting back to proving the theorem,

$$\begin{aligned}
\hat{n}4_{x,y} =& (\sum_{u=1}^{n})^{\le \psi^2} (A^2_{x,u} + z) \cdot (A^2_{y,u} + z) \\
=& \sum_{u=1}^{n} A^2_{x,u} \cdot A^2_{y,u} + z \cdot \sum_{u=1}^{n} A^2_{x,u} \\
&+ z \cdot \sum_{u=1}^{n} A^2_{y,u} + (\sum_{u=1}^{n})^{\le \psi^2} z^2 \\
=& A^4_{x,y} + 2z\psi^2 + z^2\psi^2 \qquad \text{(Using Claim 9)}
\end{aligned}$$

Thus we get

$$\left|\hat{n}4_{x,y} - A^4_{x,y}\right| = \left|2z + z^2\right| \psi^2 = \begin{cases} 3|z|\psi^2 & \text{if } |z| < 2 \\ 2z^2\psi^2 & \text{if } |z| \ge 2 \end{cases}$$

The additive error ϵ_2 for $\hat{n}^2_{x,y}$ is $\tilde{O}(\sqrt{\psi})$ from Theorem 4 (here $\tilde{O}()$ hides $\log(1/\rho)$ factors); therefore, we get the additive error for $\hat{n}4_{x,y}$ as ψ^3. □

Theorem 8 can be generalized to show that entries of A^{2^t}, for $t \ge 1$, can be approximated with additive error $\tilde{O}(poly(\psi))$. Since any even power of A can be written as a product of A raised to a power of 2, we have established that our FastEMB embedding effectively preserves all even power of the adjacency matrix A, and therefore, can approximate information about paths of even lengths in G.

4 EXPERIMENTS

Hardware description. We performed most of our experiments on a laptop with the following configuration: CPU: Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz x 8; Memory: 7.5 GB; OS: Ubuntu 18.04; OS type 64-bits. We performed our experiments on the Gowalla dataset[17] on a server with the following configuration: CPU: product: Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz; size: 1200MHz; capacity: 3GHz; width: 64 bits; memory: 94GiB; width: 64 bits.

Candidate algorithms: We used the source code of BinSketch [22] to implement out embedding and estimating algorithms; we obtained the code from the authors of BinSketch but we did not modify the code to add Line 5 of Algorithm 2 — not adding the line only makes our implementation slower and less accurate.

We evaluated our approach against several state-of-the-art algorithms such as node2vec [9], deepwalk [21], and LINE [27]. For node2vec, we used the implementation provided by its authors [1]. We did a grid search over its parameters p and $q \in [0.25, 0.5, 1, 2, 4]$. This is equivalent to running 25 different experiments. Here we only report the result for the optimum choice of p and q. deepwalk is a special case of node2vec when $p = q = 1$ [9], so the earlier node2vec implementation was used here with the specified parameters. For LINE, we used a standard implementation available online [2] and performed experiments considering both first and second order proximity — the best one is reported here. For all these three approaches, we compute the embedding in both 128 and 256 dimensions, and report the best result here.

We evaluated the effectiveness of the embeddings obtained from these algorithms for two fundamental social network applications, node classification and link prediction.

4.1 Link Prediction

In the link prediction problem we are given a network with a certain fraction of missing edges and the task is to predict these missing edges. It can be recasted as a classification problem where the goal is to train a classifier that, given a pair of nodes, outputs if there is an edge between them or not.

Datasets: We ran experiments on the following four datasets.

Gowalla [17]: Gowalla is a location-based social networking website where users share their locations by checking-in. The friendship network is undirected and was collected using their public API. The dataset consists of 196, 591 nodes and 950, 327 edges. The dataset consist of a total of 6, 442, 890 check-ins of these users over the period of Feb. 2009 - Oct. 2010.

Enron Emails Network: [17]: Enron email communication network covers all the email communication within a dataset of around half million emails. In this dataset, nodes of the network are email addresses and if an address i sent at least one email to address j, the graph contains an undirected edge from i to j. The datasets contains 36, 692 nodes and 1, 83, 831 edges.

BlogCatalog [33]: BlogCatalog is the social blog directory which manages the bloggers and their blogs. The dataset contains the friendship network crawled and group memberships. Here, nodes represent users, and edges represent a friendship relation between any two users. The network has 10, 312 nodes and 333, 983 edges.

Facebook [17]: In the Facebook network, nodes represent users, and edges represent a friendship relation between any two users. The network has 4, 039 nodes and 88, 234 edges.

Methodology: For the purpose of classification, we generate a labeled dataset of edges by splitting a dataset into training and testing partition in 70 − 30 ratio. These will serve as our positive training and testing samples. To this purpose, we randomly sample 30% of edges and remove them. During the removal of edges, we make sure that the residual graph obtained after edge removal remains connected. If the sampled edges don't ensure this, we choose a different edge. We learn the embedding of the graph using all the candidate algorithms on our positive training samples. We now generate missing edges equal in number to the original dataset. Missing edges are edges that are absent in the graph. We split the generated missing edges into training and testing partition in 70−30 ratio. These will serve as our negative training and testing samples. We compute inner product for all the edges and label them 0 or 1 depending on whether it is a missing edge or an actual edge.

Table 1: Datasets for Link Prediction

Datasets	Nodes	Edges	Sparsity
Gowalla	196,591	9,50,327	1,4730
Enron Emails Network	36,692	1,83,831	1,383
BlogCatalog	10,312	333,983	3992
Facebook	4,039	88,234	1045

We combine positive training samples and negative training samples to form our final training data. We train a logistic regression on the final training data. For testing, we combine positive testing samples and negative testing samples to form final testing data. In order to test the classifier model, we calculate inner product on the final testing data using the embeddings created earlier. We consider auc-roc score as our evaluation metric.

Empirical Results and Insights: We record the performances of all the candidate algorithms on the datasets mentioned above and summarise the comparison in Tables 2, 3, 4, 5. We observed significant speed up in embedding computation time as expected, along with better accuracy as compared to the other candidate algorithms.

For example, on the Gowalla dataset, we achieved 731 times speedup (using 2200 dimensions for FastEMB) as compared to node2vec, 661 times speedup as compared to deepwalk, and 6751 times speedup as compared to LINE, while simultaneously offering about 8% improvement in AUC-ROC and 3.72 times lesser space as compared to these algorithms. The space advantage arrives primarily due to the use of binary vectors instead of real-valued vectors each of whose dimensions require a lot many bits, 64 on our system, compared to binary vectors. We obtain similar results for the other datasets.

4.2 Node Classification

For node classification, every node is assigned one or more labels from a given set. During the training period, we observe a certain fraction of nodes and their labels. The aim is to predict the labels for the remaining nodes. We split the dataset into training and testing partition in a 70 − 30 ratio and use logistic regression as classifier.

Datasets: We used three citation datasets — Cora, Citeseer and Pubmed [25] for the experiments. In these datasets, citation relationships are viewed as directed edges. Attributes associated with nodes are extracted from the title and the abstract of the each article and are presented as sparse bag-of-word vectors, after removing the stop words and low-frequency words. Each article in these datasets has only one label representing the class it belong to.

[1] https://github.com/aditya-grover/node2vec
[2] https://github.com/shenweichen/GraphEmbedding

Table 2: Link Prediction Result on Gowalla network

Algorithm	Dimension	AUC ROC Score	Embedding Computation time(s)
FastEMB	2200	83.53	10.7
	2400	83.89	11.07
	2600	83.04	11.58
	2800	84.38	12.53
	3000	84.7	15.48
	3200	84.88	24.07
	3400	84.44	30.53
	3600	84.92	32.87
	3800	85.08	33.76
	4000	85.21	35.78
node2vec ($p = 4, q = 4$)	128	75.48	7827
deepwalk	128	74.79	7075
LINE (First Order)	128	75.33	72242

Table 3: Link Prediction Result on Enron email network

Algorithm	Dimension	AUC ROC Score	Embedding Computation time(s)
FastEMB	400	86.96	0.66
	600	91.25	0.78
	800	92.48	0.87
	1000	92.88	0.99
	1200	93.57	2.7
	1400	93.79	3.46
	1600	94.02	3.81
node2vec ($p = 4, q = 4$)	128	67.13	560
deepwalk	128	66.7	515.36
LINE (First Order)	128	75.87	4216

Table 4: Link Prediction Result on BlogCatalog dataset

Algorithm	Dimension	AUC ROC Score	Embedding Computation time(s)
FastEMB	800	83.71	0.62
	1000	84.38	0.67
	1200	84.5	0.71
	1400	84.97	0.74
	1600	85.02	0.77
	1800	85.3	0.8
	2000	85.36	0.83
	2200	85.39	0.87
	2400	85.48	1.29
node2vec ($p = 4, q = 4$)	128	63.12	979.02
deepwalk	128	61	774.31
LINE (First Order)	128	66	2379

We also considered IMDB-BINARY [23, 31] for our experiments. It is a movie collaboration dataset where actor/actress and genre information of different movies on IMDB are collected. Nodes represent actors/actresses, and an edge between them signifies a joint appearance in some movie. Collaboration graphs is generated on the "action" and the "romance" *genres* and derived *ego-networks* for each actor/actress. A movie can belong to both genres at the same time, therefore movies from romance genre are discarded if they are already included in the action genre. Each ego-network is labeled with the genre graph it belongs to. The task is then simply to identify which genre an ego-network graph belongs to.

Statistics of these datasets, including number of nodes, number of edges, number of classes (categories), and the dimension of attributes are summarised in Table 6.

Empirical Results and Insights: We run the benchmarks of the all the candidate algorithms on the datasets mentioned above and summarise the comparison in Tables 7, 8, 9, 10. Here again, we obtained significant speed up in embedding computation time while simultaneously offering comparable accuracy with respect to other candidate algorithms. Moreover, our embedding is space efficient and requires significantly less number of bits.

Table 5: Link Prediction Result on Facebook network

Algorithm	Dimension	AUC ROC Score	Embedding Computation time(s)
FastEMB	400	92.29	0.15
	600	93.34	0.16
	800	93.66	0.18
	1000	93.85	0.20
	1200	93.81	0.20
	1400	93.89	0.23
	1600	93.96	0.22
	1800	94.06	0.23
node2vec ($p = 0.25, q = 4$)	128	93.64	63.98
deepwalk	128	93.10	54.39
LINE (First Order)	128	83.41	250.25

Table 6: Datasets for Node Classification

Datasets	Nodes	Edges	Classes	Features	Sparsity
Cora	2,708	5,429	7	1,433	5
Citeseer	3.327	4,732	6	3,703	26
Pubmed	19,717	44,338	3	500	171
IMDB	19,773	3,86,124	1000	1024	540

4.3 Performance of FastEMB on varying sparsity and training sample size

We now discuss the performance of FastEMB in comparison with other candidate algorithms by varying sparsity of a dataset and varying training sample size. We discuss our experimental setup as follows. We split the datasets into various ratios of training and test partitions. We start with 50% training and 50% test partition and increased it upto 90% training and 10% test partition. We increase the training partition at the interval of 10%. We ran the Link Prediction experiment on these partitions and observed the AUC-ROC score and the embedding computation time. We ran LINE, node2vec, deepwalk on 128 dimensional embedding, and FastEMB on {400, 600, 1000, 1200, 1400, 1600, 1800, 2000, 2400} dimension. Results of our experiments on the Enron email and BlogCatalog datasets are summarised in Figures 2 and 3.

Table 7: Node Classification Result on Citeseer dataset

Algorithm	Dimension	Micro F_1 Score	Macro F_1 Score	Embedding Computation time(s)
FastEMB	10	28.67	21.25	0.018
	30	27.36	20.29	0.019
	70	31.18	24.15	0.023
	100	34.30	28.42	0.024
	150	37.62	31.14	0.026
	200	36.51	28.98	0.035
	300	34.7	28.57	0.037
	400	41.04	35.54	0.043
	600	41.85	36.23	0.049
	800	43.96	37.7	0.068
	1000	44.16	38.79	0.076
	1200	47.78	41.67	0.087
	1400	45.47	40.97	0.086
	1600	45.57	40.77	0.105
	1800	46.78	40.77	0.106
	2000	48.28	43.16	0.112
node2vec ($p = 0.25, q = 4$)	128	29.77	21.9	1.75
deepwalk	128	25.05	17.06	1.77
LINE (First Order)	128	36.72	32.72	17.14

Table 8: Node Classification Result on Cora dataset

Algorithm	Dimension	Micro F_1 Score	Macro F_1 Score	Embedding Computation time(s)
FastEMB	10	31.85	17.79	0.017
	30	37.76	28.43	0.017
	70	41.82	35.53	0.019
	100	43.91	39.47	0.02
	150	49.44	44.75	0.023
	200	51.78	47.87	0.024
	300	56.7	54.49	0.029
	350	54.98	52.44	0.031
	400	56.21	53.26	0.039
	600	60.14	58.56	0.042
	800	62.11	60.23	0.047
	1000	60.76	58.41	0.055
	2000	63.59	61.64	0.094
	2500	65.55	64.67	0.10
	3000	64.45	63.18	0.124
node2vec ($p = 0.25, q = 4$)	128	52.76	41.31	1.72
deepwalk	128	44.15	28.64	1.49
LINE(First Order)	128	56.33	53.51	15.72

Insight. We notice that the AUC-ROC score of FastEMB always remains higher than the other candidate algorithms. Furthermore, there is slight increase in the AUC-ROC score of FastEMB with the increase of training data. This implies that even on less training data FastEMB offers significantly better performance. We now comment on the behaviour of all the candidate algorithms on embedding computation time with the increase of density of the graph. It is obvious that with the increase of training partition size the dataset becomes more denser. We observe that the embedding computation time of FastEMB always remains constant, and is significantly less than the other candidate algorithms. Whereas for as candidate algorithms it become slower with the increase of density of the graphs.

Table 9: Node Classification Result on PubMed dataset

Algorithm	Dimension	Micro F_1 Score	Macro F_1 Score	Embedding Computation time(s)
FastEMB	10	43.89	32.65	0.15
	50	45.16	34.69	0.29
	100	47.71	39.27	0.20
	200	50.62	43.12	0.23
	400	54.95	50.47	0.29
	600	57.606	54.26	0.42
	800	59.77	56.14	0.83
	1000	63.28	59.63	1.23
	1200	62.69	59.69	1.91
	1400	64.4	61.52	1.68
	1600	67.25	65.07	2.20
	1800	66.9	64.41	2.51
	2000	68.27	65.59	2.5
node2vec ($p = 0.25, q = 4$)	128	43.2	31.34	5.88
deepwalk	128	42.46	35.15	5.78
LINE (First Order)	128	58.19	53.81	840.25

Table 10: Node Classification Result on IMDB dataset

Algorithm	Dimension	Micro F_1 Score	Macro F_1 Score	Embedding Computation time(s)
FastEMB	100	40.37	39.78	0.72
	200	46.91	46.48	0.76
	400	55.2	55	0.86
	600	61.48	61.34	0.91
	800	67.37	67.3	0.98
	1000	70.7	70.58	1.04
	1200	74.63	74.63	1.12
	1400	78	77.97	1.26
	1600	80	79.97	2.94
	1800	82.47	82.49	3.30
	2000	83.88	83.85	3.89
	2200	85.71	85.74	4.10
	2400	86.86	86.89	4.02
	2600	88.06	88.07	4.88
	2800	88.64	88.6	5.56
	3000	89.76	89.81	6.32
node2vec ($p = 4, q = 4$)	128	51.91	51.6	117
deepwalk	128	50.68	50.34	118
LINE (First Order)	128	41.25	40.56	1850

Comment on Space Efficiency of FastEMB. In all our experiments, we ran FastEMB on various values of dimensions, and other candidate algorithms on 128, 256 dimensions. Our embedding are binary vectors while for others candidates it is real-valued vectors. We would like to comment that our embedding is much more space efficient than others. A 128 dimensional real valued vectors requires 8, 192 bits while for FastEMB the embedding dimension is varies from 10 to 3000 dimensions and offers about 81× to 2.7× space saving. Across all the experiments, we ran FastEMB on various values of embedding dimensions and in most of the cases we noticed even

Table 11: Comparing speedup and space overhead

Datasets	Algorithm	Dimension	Speedup in embedding	Space saving
Gowalla	node2vec	2200	731 ×	3.72 ×
	deepwalk	2200	661 ×	3.72 ×
	LINE	2200	6754 ×	3.72 ×
ENRON	node2vec	400	848 ×	20.48×
	deepwalk	400	780 ×	20.48×
	LINE	400	6387 ×	20.48×
BlogCatalog	node2vec	800	1579 ×	10.24×
	deepwalk	800	1248×	10.24×
	LINE	800	3837 ×	10.24×
Facebook	node2vec	400	426.53 ×	20.48 ×
	deepwalk	400	362.6 ×	20.48 ×
	LINE	400	1668.33 ×	20.48×
PubMed	node2vec	10	39.2 ×	819.2×
	deepwalk	10	38.53 ×	819.2 ×
	LINE	10	5601.66 ×	819.2×
IMDB	node2vec	1000	112.5 ×	8.19×
	deepwalk	1000	113.46 ×	8.19 ×
	LINE	1000	1778.84 ×	8.19×

a very small embedding dimension suffices to offer comparable performance while simultaneously offering significant speedup in embedding computation time along with space savings.

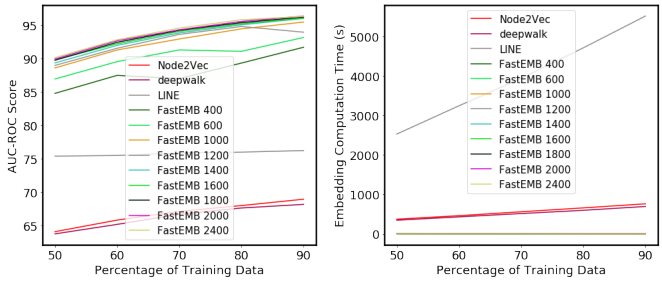

Figure 2: Comparison of AUC-ROC Score and Embedding computation time on Enron dataset.

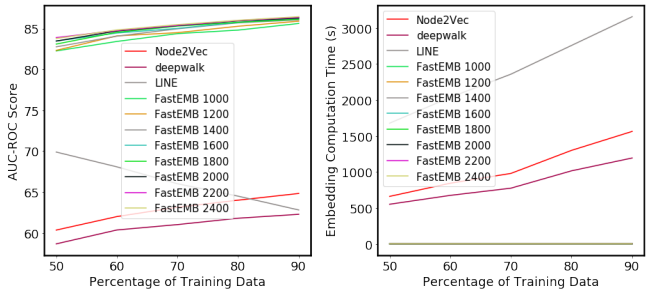

Figure 3: Comparison of AUC-ROC Score and Embedding computation time on BlogCatalog dataset.

We summarize the speed up and space overhead of the other candidate algorithms with FastEMB in Table 11.

5 CONCLUSION AND OPEN QUESTIONS

In this work, we propose FastEMB which takes a large scale graph as input and outputs succinct low-dimensional binary embedding corresponding to each node. A major advantage of FastEMB is that it is extremely fast and computes the embedding of a large scale graph in almost real time. FastEMB does not have strong hardware requirements for embedding computation and can generate embedding of large scale graph having a few million edges within a few seconds on a laptop having 8GB memory. Another advantage of our work is that is can be easily adapted in a streaming framework where edges are getting added/deleted in the graph. We evaluate the performance of FastEMB on the task of *node classification* and *link prediction* and noticed that FastEMB offers significant speed up in embedding computation time while offering comparable performance with respect to the state-of-the-art algorithms of the purpose such as node2vec, deepwalk, LINE. For example: in link prediction task on Gowalla dataset, we obtained 731 times speedup as compared to node2vec, 661 times as compared to DeepWalk, and 6751 times as compared to LINE, while simultaneously offering about 8% improvement in accuracy and taking 3.72 times less space as compared to these algorithms. Our work leaves the possibility of several open questions: a) extending our result for computing embedding of hyper-graphs; b) further improving the accuracy of downstream evaluation tasks potentially incorporating some additional features into FastEMB. Given the simplicity of FastEMB we hope that it can be easily adapted in practice.

REFERENCES

[1] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, February 2008.

[2] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[3] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1798–1828, 2013.

[4] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann Lecun. Spectral networks and locally connected networks on graphs. 12 2013.

[5] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep. pages 891–900, 10 2015.

[6] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Deep neural networks for learning graph representations. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 1145–1152. AAAI Press, 2016.

[7] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[8] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowl.-Based Syst.*, 151:78–94, 2018.

[9] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 855–864, 2016.

[10] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 1025–1035, USA, 2017. Curran Associates Inc.

[11] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Eng. Bull.*, 40(3):52–74, 2017.

[12] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Eng. Bull.*, 40:52–74, 2017.

[13] Mikael Henaff, Joan Bruna, and Yann Lecun. Deep convolutional networks on graph-structured data. 06 2015.

[14] Thomas Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. 09 2016.

[15] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[16] leonbloy. Probability of at least one male and one female sharing the same birthday, Mar 1963.

[17] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[18] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[19] Qing Lu and Lise Getoor. Link-based classification. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, pages 496–503. AAAI Press, 2003.

[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[21] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: online learning of social representations. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 701–710, 2014.

[22] Rameshwar Pratap, Debajyoti Bera, and Karthik Revanuru. Efficient sketching algorithm for sparse binary data. *CoRR*, abs/1910.04658, 2019.

[23] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[24] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *Trans. Neur. Netw.*, 20(1):61–80, January 2009.

[25] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. https://linqs.soe.ucsc.edu/data, 2008.

[26] Xiaobo Shen, Shirui Pan, Weiwei Liu, Yew-Soon Ong, and Quan-Sen Sun. Discrete network embedding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, pages 3549–3555, 2018.

[27] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. LINE: large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 1067–1077, 2015.

[28] Lei Tang and Huan Liu. Relational learning via latent social dimensions. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 817–826, New York, NY, USA, 2009. ACM.

[29] Rianne van den Berg, Thomas N. Kipf, and Max Welling. Graph convolutional matrix completion. *CoRR*, abs/1706.02263, 2017.

[30] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1225–1234, New York, NY, USA, 2016. ACM.

[31] Pinar Yanardag and S. V. N. Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 1365–1374, 2015.

[32] Hong Yang, Shirui Pan, Peng Zhang, Ling Chen, Defu Lian, and Chengqi Zhang. Binarized attributed network embedding. In *IEEE International Conference on Data Mining, ICDM 2018, Singapore, November 17-20, 2018*, pages 1476–1481, 2018.

[33] R. Zafarani and H. Liu. Social computing data repository at ASU. http://www.blogcatalog.com/, 2009.