Learning Based Proximity Matrix Factorization for Node Embedding

Xingyi Zhang
Kun Xie
Sibo Wang*
The Chinese University of Hong Kong
Hong Kong SAR, China
xyzhang,xiekun,swang@se.cuhk.edu.hk

Zengfeng Huang
Fudan University
Shanghai, China
huangzf@fudan.edu.cn

ABSTRACT

Node embedding learns a low-dimensional representation for each node in the graph. Recent progress on node embedding shows that proximity matrix factorization methods gain superb performance and scale to large graphs with millions of nodes. Existing approaches first define a proximity matrix and then learn the embeddings that fit the proximity by matrix factorization. Most existing matrix factorization methods adopt the same proximity for different tasks, while it is observed that different tasks and datasets may require different proximity, limiting their representation power.

Motivated by this, we propose *Lemane*, a framework with trainable proximity measures, which can be learned to best suit the datasets and tasks at hand automatically. Our method is end-to-end, which incorporates differentiable SVD in the pipeline so that the parameters can be trained via backpropagation. However, this learning process is still expensive on large graphs. To improve the scalability, we train proximity measures only on carefully subsampled graphs, and then apply standard proximity matrix factorization on the original graph using the learned proximity. Note that, computing the learned proximities for each pair is still expensive for large graphs, and existing techniques for computing proximities are not applicable to the learned proximities. Thus, we present generalized push techniques to make our solution scalable to large graphs with millions of nodes. Extensive experiments show that our proposed solution outperforms existing solutions on both link prediction and node classification tasks on almost all datasets.

CCS CONCEPTS

• **Information systems → Data mining**; • **Computing methodologies → Dimensionality reduction and manifold learning**.

KEYWORDS

Node Embedding; Trainable Proximity; Matrix Factorization

*Sibo Wang is the corresponding author.

KDD '21, August 14–18, 2021, Virtual Event, Singapore
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8332-5/21/08…$15.00
https://doi.org/10.1145/3447548.3467296

ACM Reference Format:
Xingyi Zhang, Kun Xie, Sibo Wang, and Zengfeng Huang. 2021. Learning Based Proximity Matrix Factorization for Node Embedding. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3447548.3467296

1 INTRODUCTION

Node embedding is the task to map nodes in the original graph into low-dimensional representations. For each node, it outputs an embedding vector and the embedding vectors play an important role in preserving not only the structural information but also other underlying properties in the graph. These vectors can be fed into machine learning models, facilitating widespread machine learning tasks, such as node classification [21, 34, 38], link prediction [41, 45], graph reconstruction [46, 48], and recommendation [17, 49].

An important category of node embedding methods with superb performance is the ones using proximity matrix factorization. For such solutions, they first define the proximity matrix $\boldsymbol{S}$ for the nodes in the input graph where $S(i, j)$ is the proximity measure of node j with respect to node i. Different methods may adopt different proximity measures and *personalized PageRank (PPR)* is a popular choice of proximity measure in node embedding. For example, PPR is adopted in NRP [46] as the proximity. In STRAP [48], the authors further propose to adopt $\boldsymbol{\pi}_u(v) + \boldsymbol{\pi}_v^T(u)$ as the proximity where $\boldsymbol{\pi}_u(v)$ is the PPR of v with respect to u and $\boldsymbol{\pi}_v^T(u)$ is the PPR of u with respect to v on the transpose graph G^T by reversing the direction of each edge in G. Given the proximity matrix $\boldsymbol{S}$, two embedding vectors $\boldsymbol{x}_u$ and $\boldsymbol{y}_u$ are derived such that $\boldsymbol{x}_u \cdot \boldsymbol{y}_v \sim S(u, v)$. For existing solutions in this category, the embedding vectors are typically obtained by singular value decomposition (SVD) or eigen-decomposition on $\boldsymbol{S}$ or on a sparse matrix closely related to $\boldsymbol{S}$.

Despite their success, all existing matrix factorization approaches aim to learn an embedding that preserves the chosen proximity without considering if the proximity is suitable for the task on the dataset or not. However, it is observed that different tasks and datasets may require different proximities to achieve high performance, limiting the representation power of such solutions. In addition, it is shown in existing node embedding methods, e.g., STRAP [48] and NetMF [37], that non-linear operations (such as taking logarithm or softmax) on the proximity matrix can help improve the representation power of the embedding. Nevertheless, most latest matrix factorization methods, like HOPE [32], AROPE [53], and NRP [46], do not explicitly derive the proximity matrix, which limits their representation powers. For those methods that explicitly derive the proximity matrix, it indicates that the matrix factorization needs to be taken on the final proximity matrix. Thus, to derive trainable proximity measures, the model needs to be end-to-end and includes the proximity computation as well as the SVD decomposition into the training process. This further imposes challenges especially when the input graph has millions of nodes.

Contribution. Motivated by the limitation of existing solutions, we present an effective framework *Lemane*[1] with trainable proximity measures, which can be learned to best suit the datasets and tasks at hand automatically. Our trainable proximity measure is inspired by personalized PageRank (PPR) [33]. The PPR $\boldsymbol{\pi}_u(v)$ can be defined as the probability that an α-discounted random walk from u stops at node v, where an α-discounted random walk from a source node u has α probability to stop at the current node and has $(1-\alpha)$ probability to randomly jump to an out-neighbor of the current node. For α-discounted random walks, the majority will always be the one-hop random walks, which may not be the most representative one for the task at hand. This motivates us to learn a more representative random walk for our trainable proximity measure. Instead of fixing the stopping probability as α at each step, our trainable proximity is defined on a supervised random walk where the stopping probability at the l-th hop is learned by our defined loss function. Then, our trainable proximity of node v with respect to u, dubbed as the supervised PPR $\boldsymbol{S}(u,v)$, is the probability that the supervised random walk from u stops at v.

To learn the stopping probabilities at each hop for the supervised random walk, we design different loss functions for different tasks in order to learn a more representative proximity for the task at hand. In this paper, we focus on two popular tasks of node embedding: link prediction and node classification. Given the loss functions and trainable parameters, we then design an end-to-end method which incorporates a differentiable SVD in the pipeline so that the parameters can be trained via backpropagation. Our solution is mainly inspired by previous work on learning-based low-rank approximations [24], which includes a differentiable SVD that allows the gradients to flow easily in the framework to solve the low-rank approximation problem. In our framework, with the differentiable SVD, the gradients can easily flow from the loss function and differentiable SVD to our proximity matrix computation, which is determined by the training parameters, i.e., the stopping probabilities at each hop of the supervised random walk.

However, the above training process is too expensive and does not scale to large graphs. To improve the scalability, we train the stopping probabilities for the supervised random walk only on carefully subsampled graphs, and then apply standard proximity matrix factorization on the original graph using the learned proximity. Our main observation is that the stopping probabilities at each hop of the supervised random walk are node-independent. Thus, with a carefully subsampled graph, the learned stopping probability at each hop for the task should still be similar to that on the input graph. Motivated by this, we present an effective subgraph sampling based method to train the parameters on multiple sampled subgraphs, which improves the scalability of our Lemane.

[1]Learning based Proximity Matrix Factorization for Node Embedding

Finally, given learned probabilities, computing learned proximities for each node-pair is still expensive for large graphs, and existing efficient algorithms for computing proximities, like PPR, are not applicable to the learned proximities. Thus, we present generalized push techniques to make our solution scalable to large graphs with millions of nodes. Given a source node s, our generalized push algorithm computes an approximate supervised PPR score for each node with respect to s with $O(\frac{1}{\delta})$ cost where δ is a parameter to control the quality of approximate supervised PPR scores. Then, the proximity matrix can be computed with $O(\frac{n}{\delta})$ cost and takes $O(\frac{n}{\delta})$ space. A sparse SVD algorithm is applied on the proximity matrix $\boldsymbol{S}$ to derive the final embedding with $O(\frac{n}{\delta}+n\cdot d^2)$ running cost, where d is the embedding dimension.

In our experiment, we compare our Lemane against 15 existing node embedding methods on the link prediction and node classification tasks using 6 real datasets with up to 3 million nodes and 117 million edges. Extensive experiments show that our Lemane outperforms existing methods on both link prediction and node classification tasks on almost all datasets.

2 RELATED WORK

There are three basic categories of node embedding methods: skip-gram methods, matrix factorization methods, and neural network methods. Next, We briefly review existing works for each category.

Skip-gram methods. The methods in this category are inspired by the great success of the word2vec model [30] for natural language processing. DeepWalk [34] first proposes to train embedding vectors by feeding truncated random walks to the Skip-gram model. The nodes sampled from the random walks are then treated as the positive samples. Subsequent methods try to explore more representative random walks to feed into the Skip-gram model. LINE [38] adopts one-hop and two-hop random walks while Node2vec [21] proposes to explore higher-order random walks that exploits both DFS and BFS nature of the graph. VERSE [41] and APP [54] adopt α-discounted random walks to obtain positive samples. Recently, InfiniteWalk [9] studies DeepWalk in the limit as the window size goes to infinite, linking DeepWalk to graph Laplacian matrix.

Matrix factorization methods. Another idea in node embedding is to do matrix factorization on a chosen proximity matrix. To explicitly derive the proximity matrix, e.g., the case in NetMF [37], it typically takes $\Theta(n^2)$ cost and is too expensive for large graphs. To avoid the $\Theta(n^2)$ running cost, HOPE [32], AROPE [53], and NRP [46] are proposed to derive the embedding without explicitly computing the proximity matrix. For instance, instead of computing all-pair proximity scores and then decomposing the proximity matrix $\boldsymbol{S}$, NRP turns to do SVD on the adjacency matrix, which is sparse for most real-life graphs, reducing the embedding computational cost. Another solution to avoid the $\Theta(n^2)$ cost is to calculate a sparsified proximity matrix $\boldsymbol{S}$. The representative is STRAP [48], which imposes a threshold δ and returns at most $O(\frac{1}{\delta})$ proximity scores no smaller than δ for each node, making the proximity matrix of $O(\frac{n}{\delta})$ size. An SVD is then applied to the sparsified proximity matrix. Since the second solution explicitly derives the proximity matrix, it allows to take non-linear operations on the proximity matrix, improving the representation powers.

Algorithm 1: Compute-Supervised-PPR($\boldsymbol{P}$, L)

```
1 $S \leftarrow \mathbf{0}, R \leftarrow I_n$
2 for $k = 0$ to $L$ do
3     $S \leftarrow S + \alpha_k \cdot R$
4     $R \leftarrow (1 - \alpha_k) \cdot P \cdot R$
5 return $S$
```

Neural network methods. Deep learning provides an alternative solution to generate node embeddings. SDNE [44] and DNGR [8] employ multi-layer auto-encoders with a target matrix to generate embeddings. DRNE [42] utilizes the layer normalized LSTM [23] to generate node embeddings by aggregating the representations of neighbors of each node recursively. GraphGAN [45] adopts the well-known generative adversarial networks [19] into graph representation learning via an adversarial minimax game. AW [5] proposes a novel attention model on the power series of the transition matrix, which guides the random walk to pay attention to important positions within the random walks by optimizing an upstream objective. The bottleneck of these solutions is the high computational cost, which restricts these methods to small graphs.
Other methods. There are also several methods that do not belong to the above three categories. For example, GraphWave [14] learns node representations by leveraging heat wavelet diffusion patterns in an unsupervised way. NetHiex [29] captures the underlying hierarchical taxonomy of the graph to learn node representations with multiple components. RaRE [22] proposes a node embedding method that considers both social rank and proximity of nodes, and separately learns two representations for a node. AutoNE [43] incorporates AutoML into node embedding, which can automatically optimize the hyperparameters from the subgraphs of the original graph. PBG [26] presents a distributed embedding system that uses the block decomposition of the adjacency matrix as a partition method to scale to arbitrary large graphs. A recent work, GraphZOOM [12], first generates subgraphs based on a fused graph and then applies existing approaches to generate node embeddings. Since these methods do not preserve any node pair proximity, the main concern is their insufficient effectiveness for downstream tasks as we will show in our experiment.

There are also various graph embedding methods designed for specific graphs, like dynamic graphs [20, 40] and heterogeneous networks [13, 18]. In this paper, we focus on the most fundamental case when the network is static and no feature vector is given.

3 LEMANE FRAMEWORK

In this section, we present our Lemane framework. Section 3.1 introduces the trainable proximity matrix and the training parameters. Section 3.2 elaborates on the training process of Lemane and introduces the loss functions used for link prediction and node classification, respectively. Section 3.3 presents how to carefully obtain the subsampled graphs to do training on large graphs.

3.1 Trainable Proximity Measure

Recap from Section 1 that, the personalized PageRank (PPR) $\boldsymbol{\pi}_u(v)$ of node v with respect to u is the probability that an α-discounted random walk from u stops at node v, where an α-discounted random walk has α probability to stop at the current node and $(1 - \alpha)$ probability to randomly jump to one of its out-neighbors. Define the transition matrix $\boldsymbol{P} = \boldsymbol{D}^{-1}\boldsymbol{A}$ where $\boldsymbol{D}$ is the diagonal matrix such that $\boldsymbol{D}(i, i)$ is the out-degree (resp. degree) of node i if G is directed (resp. undirected), and $\boldsymbol{A}$ is the adjacency matrix. Then, the PPR proximity matrix can be expressed as:

$$\boldsymbol{S} = \sum_{l=0}^{\infty} \alpha \cdot (1 - \alpha)^l \cdot \boldsymbol{P}^l.$$

In our trainable proximity measure, instead of fixing the stopping probability at each step to be α, we allow the stopping probability α_l of the random walk at the l-th hop to be trainable. Currently, we assume that the stopping probability α_l at the l-th step is given and will show how to train α_l in Section 3.2. A random walk that follows such learned stopping probability at each step is denoted as a *supervised random walk* and the proximity derived from the supervised random walk is denoted as supervised PPR. The supervised PPR proximity matrix can be defined as:

$$\boldsymbol{S} = \alpha_0 \boldsymbol{I}_n + \sum_{l=1}^{\infty} \alpha_l \cdot \prod_{k=0}^{l-1} (1 - \alpha_k) \cdot \boldsymbol{P}^l. \tag{1}$$

To explain, the probability that a supervised random walk stops at exactly the l-th hop is $\alpha_l \cdot \prod_{k=0}^{l-1}(1 - \alpha_k)$. Thus, by summing up the probability to stop at each hop, we derive the supervised PPR score.

The exact supervised PPR score then can be computed with an iterative manner as shown in Algorithm 1 when $L \to \infty$. However, we observe that when L is sufficiently large, the probability that the supervised random walk stops with a hop number larger than L is close to zero. Hence, we discard all supervised random walks that stop with a hop larger than L. The following theorem shows the quality of calculated supervised PPR scores with Algorithm 1.

Theorem 1. *Let $\boldsymbol{S}_L$ be the supervised PPR proximity matrix derived by Algorithm 1. Let $||\boldsymbol{M}||_\infty$ be the infinity-norm of matrix $\boldsymbol{M}$, we have:*

$$||\boldsymbol{S}_L - \boldsymbol{S}||_\infty \leq \prod_{k=0}^{L} (1 - \alpha_k) ||\boldsymbol{S}||_\infty.$$

All the proofs of our theorems can be found in the appendix. According to our observation, the probability that the supervised random walk stops at a hop greater than L is almost zero when we set $L = 15$. Hence, L is set to 15 in our experiment.

Algorithm 1 makes a tight connection between the supervised PPR proximity matrix and our training parameters α_l $(0 \leq l \leq L)$, which allows the gradients to flow from the derived supervised PPR proximity matrix $\boldsymbol{S}_L$ to the training parameters via backpropagation. Let n and m denote the number of nodes and the number of edges, respectively. The time complexity of the algorithm can be bounded by $O(m \cdot n \cdot L)$ since $\boldsymbol{P}$ is a sparse matrix with m entries.

Remark. Note that the idea to learn the stopping probability for each hop is not a new idea in graph neural networks, e.g., [10, 25]. It is much easier for neural networks as the stopping probabilities can be treated as additional weights to learn. However, it is more challenging to integrate this idea to node embedding, especially for matrix factorization methods. We show an efficient and effective solution that works for large graphs, which is non-trivial.

3.2 Training process of Lemane

Algorithm 2 shows the pseudo-code of the training process of Lemane. Firstly, it initializes the stopping probability α_k for $0 \leq k \leq L$ such that the probability a random walk stops at the l-th hop follows some standard distribution, e.g., uniform, geometric, or Poisson (Line 1). Next, it uses the initial settings of these stopping probabilities to derive the supervised PPR score by invoking Algorithm 1 (Line 3). Given the supervised PPR scores, it eliminates all the proximity scores that are too small. A threshold δ is included and all scores smaller than δ are set to zero (Lines 4-5). Then, a non-linear operation, $\log(\frac{S}{\delta})$, is applied to the proximity matrix $\boldsymbol{S}$. The proximity scores are divided by δ to guarantee that each entry after taking the log will be non-negative (Line 6). Thereafter, a differentiable SVD, e.g., [24], is applied on the new matrix $\boldsymbol{M}$ with input parameter d (Line 7). The SVD obtains two $n \times d$ matrices $\boldsymbol{U}$ and $\boldsymbol{V}$, and a $d \times d$ diagonal matrix Σ such that $\boldsymbol{U}\Sigma\boldsymbol{V}^{\boldsymbol{T}} \approx \boldsymbol{M}$. Given the three matrices, $\boldsymbol{U}\sqrt{\Sigma}$ is returned as the first embedding matrix $\boldsymbol{X}$ and $\boldsymbol{V}\sqrt{\Sigma}$ is returned as the second embedding matrix $\boldsymbol{Y}$.

Subsequently, the two embedding matrices are fed into the loss function for the link prediction or node classification (Line 9). The loss functions will be discussed shortly. Then, the stopping probabilities are updated according to the loss function by backpropagation. The training process terminates until the loss function converges (Line 2). Notice that, due to the extremely long computational chain, it is infeasible to write down the explicit form of the gradients. However, like modern deep neural networks, we can use the autograd feature in PyTorch to numerically compute the gradients with respect to the training parameters. Also, the builtin SVD in PyTorch supports to compute the gradient and hence we directly adopt the PyTorch implementation. In what follows, we elaborate on the details of the loss functions.

Algorithm 2: Lemane-Trainining

```
Input: Matrix $\boldsymbol{P}$, maximum length $L$, threshold $\delta$,
       embedding dimension $d$, learning rate $\eta$
Output: stopping probabilities $\alpha_0, ..., \alpha_L$
1  Initialize $\alpha_k$ for $k = 0, 1, ..., L$
2  while not convergence do
3      $\boldsymbol{S} \leftarrow$ Compute-Supervised-PPR($\boldsymbol{P}, L$)
4      if $S(i,j) < \delta$, for $\forall S(i,j) \in S$ then
5          $S(i,j) \leftarrow 0$
6      Get matrix $\boldsymbol{M} \leftarrow \log(\frac{S}{\delta})$ for non-zero entries
7      $[\boldsymbol{U}, \Sigma, \boldsymbol{V}] \leftarrow$ Differentiable-SVD($\boldsymbol{M}, d$)
8      $\boldsymbol{X} \leftarrow \boldsymbol{U}\sqrt{\Sigma}$, $\boldsymbol{Y} \leftarrow \boldsymbol{V}\sqrt{\Sigma}$
9      Compute link prediction loss $\mathcal{L}_p$ via Eq.4 or node
         classification loss $\mathcal{L}_c$ via Eq.7
10     for $k = 0, ..., L$ do
11         $\alpha_k \leftarrow \alpha_k - \eta\nabla_{\alpha_k}\mathcal{L}$
12 return $\alpha_0, ..., \alpha_L$
```

Loss function for link prediction. For link prediction, there are two components in our loss function. The first component of the loss function aims to ensure that the total information in the supervised random walks started from node u is closed to its out-degree. Let $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \cdots, \alpha_L)$. The first part of the loss function is as follows:

$$\mathcal{L}_1(\boldsymbol{\alpha}; \boldsymbol{A}) = \frac{1}{n^2}\sum_{u} \| \sum_{v \neq u} \boldsymbol{x}_u \cdot \boldsymbol{y}_v - d_{out}(u) \|_2^2, \tag{2}$$

where $\boldsymbol{x}_u$ is the u-th row of $\boldsymbol{X}$, $\boldsymbol{y}_v$ is the v-th row of $\boldsymbol{Y}$, and $d_{out}(u)$ is the out-degree of u. Equation 2 indicates that our learned embedding by the supervised random walk should preserve the out-degree information as much as possible.

The second part of the loss function for link prediction is the average cross-entropy over all existing edges:

$$\mathcal{L}_2(\boldsymbol{\alpha}; \boldsymbol{A}) = -\frac{1}{m}\sum_{u}\sum_{v} \boldsymbol{A}_{u,v} \log(\sigma(\boldsymbol{x}_u \cdot \boldsymbol{y}_v)), \tag{3}$$

where $\sigma(x) = 1/(1+\exp(-x))$ is the sigmoid function and $\boldsymbol{A}_{u,v} = 1$ if edge (u,v) exists in the input graph and $\boldsymbol{A}_{u,v} = 0$ otherwise. The final loss function for link prediction is:

$$\mathcal{L}_{\mathrm{p}} = \beta\mathcal{L}_1 + \gamma\mathcal{L}_2, \tag{4}$$

where β and γ are two balancing hyperparameters.

Loss function for node classification. There are also two parts in the loss function for node classification. We first concatenate two output embedding matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ together to get the unique embedding matrix $\boldsymbol{Z} = \mathrm{concat}(\boldsymbol{X}, \boldsymbol{Y})$. Then following the fact that two randomly selected nodes have different labels with high probability, we randomly sample a small set of negative node-pairs $\mathcal{N} \subset V \times V$ for each iteration. Let $H = (V, \mathcal{N})$ denotes the graph with edge set $\mathcal{N}$ with unnormalized Laplacian matrix $\boldsymbol{L}_H$ and $G_k = (V_k, E_k)$ be a complete graph formed by nodes in G with the same label k. Inspired by the relaxation proposed in [51], the first loss function for node classification is defined as follows:

$$\mathcal{L}'_1(\boldsymbol{\alpha}; \boldsymbol{A}) = \frac{\sum_{k=1}^{n_c}\sum_{u,v \in V_k} \|z_u - z_v\|_2^2}{n_c \cdot \sum_{(u,v) \in \mathcal{N}} \|z_u - z_v\|_2^2} = \frac{\sum_{k=1}^{n_c} \mathrm{Tr}(\boldsymbol{Z}^\top \boldsymbol{L}_k \boldsymbol{Z})}{n_c \mathrm{Tr}(\boldsymbol{Z}^\top \boldsymbol{L}_H \boldsymbol{Z})}, \tag{5}$$

where $\boldsymbol{z}_u$ is the concatenated representation of node u, n_c is the total number of class labels in the graph, and L_k is the unnormalized Laplacian matrix of G_k. The goal of $\mathcal{L}'_1$ is to minimize pair-wise distances between node pairs with the same class label and maximize pair-wise distances between negative samples.

Next, we employ an activation function to normalize the output embedding to a probability distribution over predicted class labels: $p_{uk} = \mathrm{softmax}(\boldsymbol{Z} \cdot \boldsymbol{W} + \boldsymbol{b})_{uk}$, where $\boldsymbol{W}$ is a $2d \times n_c$ fixed mapping matrix generated from uniform distribution, $\boldsymbol{b}$ is the bias term, p_{uk} denotes the probability that node u has class label k, and $\mathrm{softmax}(\boldsymbol{x})_{uk} = \exp(\boldsymbol{x}_{uk})/(\sum_{c=1}^{n_c}\exp(\boldsymbol{x}_{uc}))$. Let $\mathcal{Y}$ denote the $n \times n_c$ label matrix, where $\mathcal{Y}_{u,k} = 1$ if node u has class label k and $\mathcal{Y}_{u,k} = 0$ otherwise. Then, the second part of the loss function for node classification is the average cross-entropy over all nodes:

$$\mathcal{L}'_2(\boldsymbol{\alpha}; \boldsymbol{A}) = -\frac{1}{n}\sum_{u}\sum_{k=1}^{n_c} \mathcal{Y}_{u,k} \log p_{uk}. \tag{6}$$

The final loss function for node classification is defined according to $\mathcal{L}'_1$ and $\mathcal{L}'_2$ as follows:

$$\mathcal{L}_{\mathrm{c}} = \beta'\mathcal{L}'_1 + \gamma'\mathcal{L}'_2, \tag{7}$$

where β' and γ' are two balancing hyperparameters.

Algorithm 3: Lemane-Generalized-Push

```
Input: Graph $G$, source node $s$, threshold $\delta$, stopping
       probabilities $\alpha_0, ..., \alpha_L$
Output: Approximate proximity vector $\hat{\boldsymbol{\pi}}_s$
1 Initialize $\hat{\boldsymbol{\pi}}_s \leftarrow 0, \boldsymbol{r}_s^{(k)} \leftarrow 0$ for $k = 0, 1, ..., L$
2 Initialize $\boldsymbol{r}_s^{(0)}(s) \leftarrow 1$
3 while $\exists v \in V, 0 \le k \le L$ such that $\boldsymbol{r}_s^{(k)}(v) > \delta \cdot d_{out}(v)$ do
4     $\hat{\boldsymbol{\pi}}_s(v) \leftarrow \hat{\boldsymbol{\pi}}_s(v) + \alpha_k \cdot \boldsymbol{r}_s^{(k)}(v)$
5     for $u \in N(v)$ do
6         $\boldsymbol{r}_s^{(k+1)}(u) \leftarrow \boldsymbol{r}_s^{(k+1)}(u) + (1 - \alpha_k) \cdot \frac{\boldsymbol{r}_s^{(k)}(v)}{d_{out}(v)}$
7     $\boldsymbol{r}_s^{(k)}(v) \leftarrow 0$
8 return $\hat{\boldsymbol{\pi}}_s$
```

Remark. Notice that the training algorithm for Lemane on node classification needs additional label information and we randomly sample 5% of the labeled nodes for training. To make a fair comparison with our competitors, to train the classifiers, we will only include $(x - 5)\%$ new training data if we split $x\%$ of the data for training and the remaining $(100 - x)\%$ for testing. This guarantees that our Lemane only accesses the same amount of labeled data compared to other competitors.

3.3 Training Lemane with sub-sampling

The above-mentioned training process requires calculations on a dense matrix of $O(n^2)$ size and requires $O(L \cdot n \cdot m)$ running cost, which makes it non-scalable to large graphs. However, such cost seems unavoidable at first glance since we need to obtain the proximity matrix to do backpropagation. After a careful analysis, we make the following two observations to help us avoid the high running costs. Our first key observation is that the parameters that we need to train are only the stopping probabilities at each hop, which are node-independent. That is to say, if we can find a subgraph of the input graph G such that the learned stopping probabilities on the subgraph are identical to that on G for the same task, we can simply learn the parameters on the subgraph with smaller size and apply the learned parameters to the original graph directly, reducing the computational costs. Another observation is that a subgraph of G with similar connectivity should share similar learning stopping probabilities as the input graph on the same task.

Motivated by this, we present our sub-sampling based training method for Lemane on large graphs. Obviously, a straightforward solution is to sample a number n_s of nodes and then consider the subgraph containing these n_s nodes. However, such a solution severely degrades the connectivity among the nodes. Simple edge sampling strategies will face a similar dilemma which hampers the connectivity among the sampled nodes.

To keep the connectivity among the sampled nodes, we apply a BFS style traversal for subgraph sampling. Our goal is still to sample a subgraph with a constant number n_s of nodes. To sample such a subgraph, firstly a source node u is randomly sampled from the input graph G. Thereafter, a BFS traversal is applied from the source u to explore the local community of node u. If the number of visited nodes by the BFS from u is smaller than n_s, another node v is randomly sampled as the source to do BFS. The BFS sampling stops as soon as in total n_s nodes are visited.

However, the weights trained on a single subgraph might be biased and makes the learned stopping probabilities non-generalizable to the original input graph G. To make the learned parameters generalizable to the input graph, we sample multiple subgraphs by the above strategy to learn the parameters. In particular, in each iteration, we sample a subgraph by the BFS strategy and then update the training parameters, i.e., the stopping probabilities, according to the loss functions defined on this subgraph. The loss functions on the subgraph are modified accordingly where the total number n of nodes in Equations 2 and 6 are replaced by sample size n_s; the total number m of edges in Equation 3 is also replaced by sample size n_s; the set of nodes with label k in Equation 5 is changed to $V_{Sk} = V_k \cap V_S$, where V_S is the node set of the subgraph, V_k is the set of nodes with label k in G; and the negative sample set in Equation 5 is changed to $\mathcal{N}_S \subset V_S \times V_S$.

With such a sampling technique, the time complexity of Algorithm 2 can be bounded by $O(h \cdot L \cdot n_s^3)$ where h is the number of training iterations by Algorithm 2 until it converges. Since n_s is a controllable constant, we set n_s such that the proximity matrix can be fed into the GPU memory for more efficient training.

4 GENERALIZED PUSH

Given the learned stopping probabilities, a straightforward solution is to invoke Algorithm 1 to derive the supervised PPR scores. However, this incurs $O(L \cdot n \cdot m)$ running cost, which is prohibitive for large graphs. To tackle this issue, we present a generalized push algorithm to efficiently compute the supervised PPR proximity matrix with $O(\frac{n}{\delta})$ cost, where δ is a parameter to control the computational cost as well as the sparsity of the proximity matrix.

4.1 Generalized Push Algorithm

Our main idea to compute the supervised PPR proximity matrix is to derive a sparsified proximity matrix such that the supervise PPR scores no larger than δ can be safely discarded. But still, how much cost should we take to derive a sufficiently accurate approximation proximity score? In STRAP [48], the authors propose a solution to derive the PPR estimations such that $|\boldsymbol{\pi}_u(v) - \hat{\boldsymbol{\pi}}_u(v)| < \delta$ with a cost of $O(\frac{m}{\delta})$. But, can we further reduce the running cost without sacrificing the embedding quality? Here, we give an affirmative answer. Our solution is inspired by the local graph clustering algorithm Local-Push [6] which returns approximate PPRs with respect to a source s in $O(\frac{1}{\delta})$ running time and guarantees that

$$|\boldsymbol{\pi}_u(v) - \hat{\boldsymbol{\pi}}_u(v)|/d_{out}(v) < \delta, \text{for any } v \in V,$$

on undirected graphs where $d_{out}(v)$ is the degree of node v. The Local-Push algorithm suggests that if we only want to compute the approximate PPR scores around the local graph cluster with respect to a source, the running time can be reduced. In our case, the node embedding aims to find nodes that are their representatives and the nodes in their local graph cluster stand as perfect representatives. However, the Local-Push algorithm only works for PPR, not for our supervised PPR. Thus, we present a generalized push algorithm that works for arbitrary stopping probabilities.

```
Algorithm 4: Lemane-Embedding
   Input: Graph $G$, dimension $d$, threshold $\delta$, stopping
          probability vector $\boldsymbol{\alpha}$
   Output: Embedding matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$
1  Initialize the proximity matrix $\boldsymbol{S} \leftarrow \boldsymbol{0}$
2  for each node $u \in V$ do
3      $\hat{\boldsymbol{\pi}}_u \leftarrow$ Lemane-Generalized-Push$(G, u, \delta, \boldsymbol{\alpha})$
4      $\hat{\boldsymbol{\pi}}_u^T \leftarrow$ Lemane-Generalized-Push$(G^T, u, \delta, \boldsymbol{\alpha})$
5      for each node $v$ in $V$ do
6          if $\hat{\boldsymbol{\pi}}_u(v) > \delta$ then
7              $S(u,v) += \hat{\boldsymbol{\pi}}_u(v)$
8          if $\hat{\boldsymbol{\pi}}_u^T(v) > \delta$ then
9              $S(v,u) += \hat{\boldsymbol{\pi}}_u^T(v)$
10 $\boldsymbol{M} \leftarrow \log(\frac{\boldsymbol{S}}{\delta})$ for non-zero entries
11 $[\boldsymbol{U}, \Sigma, \boldsymbol{V}] \leftarrow$ SparseSVD$(\boldsymbol{M}, d)$
12 $\boldsymbol{X} \leftarrow \boldsymbol{U}\sqrt{\Sigma}, \boldsymbol{Y} \leftarrow \boldsymbol{V}\sqrt{\Sigma}$
13 return $\boldsymbol{X}, \boldsymbol{Y}$
```

Algorithm 3 shows the pseudo-code for our generalized push algorithm. Given a source node s, a vector $\hat{\boldsymbol{\pi}}_s$ is maintained to store the portion of supervised random walks that has stopped at each node, and is the estimated supervised PPR scores with respect to source s. Besides, for each hop $0 \leq k \leq L$, where L is the maximum length of a truncated supervised random walk introduced in Section 3.1, an additional residue vector $\boldsymbol{r}_s^{(k)}$ is maintained. The vector $\boldsymbol{r}_s^{(k)}$ indicates the portion of supervised random walks from s that currently stay at the k-th hop but have not stopped yet. Thus, if the residue vectors are all zero, it returns the exact supervised PPR scores. Initially, the residue vectors are all zero except for $\boldsymbol{r}_s^{(0)}(s) = 1$ (Lines 1-2), indicating that the supervised random walks initially all stay at s and has not stopped yet. Then, if any entry $\boldsymbol{r}_s^{(k)}(v)$ in the L residue vectors is above $\delta \cdot d_{out}(v)$ (Line 3), a push operation (Lines 4-7) is invoked. In particular, it first converts $\alpha_k \cdot \boldsymbol{r}_s^{(k)}(v)$ to $\hat{\boldsymbol{\pi}}_s(v)$ (Line 4). To explain, α_k portion of the $\boldsymbol{r}_s^{(k)}(v)$ random walks stop at the k-th hop. Next, the remaining $(1-\alpha_k)$ portion of the $\boldsymbol{r}_s^{(k)}(v)$ random walks randomly jump to the out-neighbors of v (Lines 5-6). Thus, for each u that is an out-neighbor of v, the residue $\boldsymbol{r}_s^{(k+1)}(u)$ is incremented by $(1-\alpha_k) \cdot \boldsymbol{r}_s^{(k)}(v)/d_{out}(v)$. After the push operation, the residue $\boldsymbol{r}_s^{(k)}(v)$ is set to zero (Line 7). The algorithm terminates when there exists no residue $\boldsymbol{r}_s^{(k)}(v)$ for any k such that it is larger than $d_{out}(v) \cdot \delta$.

Theorem 2. *Algorithm 3 runs in $O(\frac{1}{\delta})$ time.*

Theorem 3. *Let $\boldsymbol{\pi}_s^L(u)$ be the supervised PPR considering random walks within L hops. Then for undirected graphs, Algorithm 3 returns an estimation $\hat{\boldsymbol{\pi}}_s(u)$ of $\boldsymbol{\pi}_s^L(u)$ for each node u such that:*

$$|\hat{\boldsymbol{\pi}}_s(u) - \boldsymbol{\pi}_s^L(u)|/d_{out}(u) \leq \delta \cdot L.$$

By setting $\delta = \frac{\delta'}{L}$, Algorithm 3 runs in $O(\frac{L}{\delta'})$ time. At the same time, the error bound in Algorithm 3 can be bounded by $L \cdot \frac{\delta'}{L} = \delta'$. Since L can be treated as a constant, the running time with $\delta = \frac{\delta'}{L}$ is still $O(\frac{1}{\delta'})$ and for any node u, we have that:

$$|\hat{\boldsymbol{\pi}}_s(u) - \boldsymbol{\pi}_s^L(u)|/d_{out}(u) \leq \delta'.$$

The above analysis shows that our generalized push algorithm can provide identical result quality as the Local-Push algorithm with the identical asymptotic running cost, thus returning high-quality results for the representative nodes of the source node.

Table 1: Dataset statistics.

Name	Type	n	m	labels
Wikipedia	directed	4.78K	184.81K	40
WikiVote	directed	7.12K	103.69K	-
BlogCatalog	undirected	10.31K	333.98K	39
Slashdot	directed	82.17K	870.16K	-
TWeibo	directed	1.94M	50.66M	100
Orkut	undirected	3.07M	117.19M	100

4.2 Final Embedding

Given the generalized push algorithm, we finally show how to output the embedding for the graph. Following STRAP [48], we compute the supervised PPR on both the input graph G and the transpose graph G^T by reversing the direction of each edge of G and set $S(u,v)$ as $\boldsymbol{\pi}_u(v) + \boldsymbol{\pi}_v^T(u)$, where $\boldsymbol{\pi}_u(v)$ is the supervised PPR of v with respect to u and $\boldsymbol{\pi}_v^T(u)$ is the supervised PPR of u with respect to v on the transpose graph G^T. Note that we do not bring this part into our training phase to reduce the computational costs since we use the same stopping probabilities for both the input graph G and the transpose graph G^T. Algorithm 4 shows how to compute approximate proximity scores. For each node u, we compute the approximate supervised PPR on the input graph G and the transpose graph G^T (Lines 3-4). For any approximate supervised PPR score $\hat{\boldsymbol{\pi}}_u(v)$, it is added to $\boldsymbol{S}(u,v)$ only if it is larger than the threshold δ. Similarly, each approximate supervised PPR score $\hat{\boldsymbol{\pi}}_u^T(v)$ on the transpose graph G^T is added to $\boldsymbol{S}(v,u)$ only if $\hat{\boldsymbol{\pi}}_u^T(v)$ is larger than δ (Lines 5-9). With such a pruning strategy, the proximity matrix $\boldsymbol{S}$ is sparsified to include $O(\frac{n}{\delta})$ non-zero entries. Then, a non-linear operation $\log(\frac{\boldsymbol{S}}{\delta})$, is applied to $\boldsymbol{S}$. Notice that all the entries are divided by δ before taking the logarithm to guarantee that the values will be non-negative (Line 10). The resulting matrix $\boldsymbol{M}$ is then fed to a sparse SVD to derive final embedding matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ (Lines 11-12). We have the following theorem for the running time and decomposition quality with respect to $\boldsymbol{M}$.

Theorem 4. *Algorithm 4 runs in $O(\frac{n}{\delta} + \frac{n \cdot d^2}{\epsilon^4})$ time to guarantee that the embedding preserves the feeding matrix $\boldsymbol{M}$ with $(1+\epsilon)$-approximation to the best rank-d matrix in terms of Frobenius-norm.*

$$||\boldsymbol{M} - \boldsymbol{X}\boldsymbol{Y}^T||_F \leq (1+\epsilon) \min_{rank(\boldsymbol{B}) \leq d} ||\boldsymbol{M} - \boldsymbol{B}||_F.$$

Following previous work [48], we treat ϵ as a constant and use the default setting of builtin SVD implementations. The running time of Algorithm 4 is $O(\frac{n}{\delta} + n \cdot d^2)$.

5 EXPERIMENTS

We compare our Lemane against alternative solutions on link prediction and node classification tasks. All experiments are conducted

Table 2: Link prediction precision (%) on small datasets.

Method	Wikipedia	Wikivote	BlogCatalog
DeepWalk	88.33	68.32	85.35
Node2vec	82.86	78.50	82.07
VERSE	88.09	82.82	88.49
InfiniteWalk	66.86	81.07	84.05
AROPE	84.27	62.08	88.30
RandNE	83.15	77.62	87.09
ProNE	52.06	66.22	57.87
NetSMF	72.01	72.64	47.92
STRAP	86.53	92.58	89.58
NRP	83.56	91.07	90.10
GraphGAN	70.33	71.76	71.83
AW	50.42	56.62	62.56
NetHiex	45.03	73.01	65.58
GraphZoom	84.73	82.10	86.82
Louvain	56.22	58.33	59.97
Lemane-F	87.79	92.78	89.92

on a Linux machine with an Intel Xeon(R) CPU clocked at 2.70GHz, an NVIDIA GeForce RTX 2080 Super 8GB GPU, and 384GB memory.

5.1 Experimental settings

Datasets. We test on six real datasets that are used in recent node embedding studies [21, 26, 34, 41, 46–48]. The statistics of these datasets are shown in Table 1. *BlogCatalog* [39], *Slashdot* [28], *TWeibo* [1] and *Orkut* [31] are four social networks in which links represent a friendship/following relationship between users. *Wikipedia* [2] is a co-occurrence network of words appearing in the Wikipedia dump. *Wikivote* [27] is a who-votes-on-whom network on Wikipedia. All datasets and the node labels (if any) can be downloaded from public sources [1–4].

Competitors. We evaluate Lemane against 15 node embedding methods, including some classic methods and several state-of-the-art methods. We divide these methods into four groups as follows.

- Skip-gram methods: DeepWalk [34], Node2vec [21], VERSE [41], and InfiniteWalk [9];
- Matrix factorization methods: AROPE [53], RandNE [52], ProNE [50], NetSMF [36], STRAP [48], and NRP[2] [46];
- Neural network methods: GraphGAN [45] and AW [5];
- Other methods: NetHiex [29], GraphZOOM[3] [12], Louvain [7].

For our methods, we use Lemane-F to indicate the algorithm trained on the entire graph and Lemane-S to indicate the algorithm trained on subsampled graphs. Notice that Lemane-F is adopted for small datasets Wikipedia, Wikivote, and BlogCatalog. Lemane-S is adopted for large datasets Slashdot, TWeibo, and Orkut.

Parameter settings. We obtain the source code of all competitors from GitHub and perform these methods with default parameter settings suggested by their authors. Following previous studies [34, 41, 48], we set the embedding dimensionality $d = 128$. For our Lemane-S, we set the sample size $n_s = 5000$. For Lemane-F and

[2]There were some implementation issues in the released code of NRP on undirected graphs, which was fixed recently by the inventors.
[3]We use the default embedding method DeepWalk for GraphZoom.

Table 3: Link prediction precision (%) on large datasets.

Method	Slashdot	Tweibo	Orkut
AROPE	82.83	69.46	82.03
RandNE	81.03	70.74	79.45
ProNE	72.80	45.47	80.88
STRAP	83.07	94.58	85.73
NRP	80.98	93.87	86.34
Louvain	55.56	64.25	80.85
Lemane-S	84.13	94.89	89.15

Lemane-S, we use grid search to set $\beta, \beta', \gamma, \gamma'$ from {0.01, 0.1, 0.5, 1, 2, 3}, learning rate from {0.001, 0.005, 0.01, 0.05, 0.1, 0.5}, and threshold δ from $\{10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}\}$; we use JacobiSVD for Lemane-F and frPCA [16] for Lemane-S, to generate final embeddings.

Since the backpropagation in Lemane is complicated, our objective function may easily fall into a local minimum. To tackle this issue, the stopping probabilities are initialized with different distributions. Specifically, we set $\alpha_0, \alpha_1, \cdots, \alpha_L$ such that the probability that the supervised random walk stops at the l-th hop follows a geometric distribution or Poisson distribution with respect to l and we report the best results of Lemane. The parameters of Lemane are optimized by Stochastic Gradient Descent (SGD) optimizer.

5.2 Link Prediction

Link prediction aims to predict which pairs of nodes are likely to form edges. Following previous work [46, 53], we randomly hide 30% of the edges for testing and train the embedding vectors on the rest of the graph. Then, the testing set is generated by including *(i)* the node pairs corresponding to the 30% removed edges, and *(ii)* an equal number of node pairs that are not connected by any edge in the original graph G. Given a node pair (u, v) in E_{test}, we compute a score for (u, v) based on the embedding vectors, and evaluate model performance using precision score.

Following previous work [46, 48], for DeepWalk, Node2vec, VERSE, InfiniteWalk, GraphGAN, and Louvain, we use the edge-feature approach introduced in [29]: *(i)* randomly select 30% existing edges which are not in E_{test} and the same number of non-existing edges as training set E_{train} on each dataset; *(ii)* for each node pair $(u, v) \in E_{train} \cup E_{test}$, we concatenate d-dimension embedding vectors of node u and that of node v; *(iii)* we consider the $2d$-length vectors as the features of node pairs in E_{train} and feed them into a binary logistic regression classifier; *(iv)* then the trained classifier is used to perform link prediction on E_{test}. For AROPE, RandNE, ProNE, AW, NetHiex, and GraphZOOM, the score of a node pair (u, v) is the inner product of the embedding vector of node u and that of node v; for NetSMF, STRAP, NRP, and our Lemane, the score of a node pair (u, v) is the inner product of embedding $\boldsymbol{x}_u$ from $\boldsymbol{X}$ and $\boldsymbol{y}_v$ from $\boldsymbol{Y}$ (Ref. to Section 3 for the definitions of $\boldsymbol{X}$ and $\boldsymbol{Y}$).

Table 2 reports the performance of Lemane-F against the 15 competitors on three small datasets. Table 3 further reports the performance of Lemane-S against 6 methods which scale to large graphs. For the other 9 methods, they either cannot finish training in 24 hours or run out of memory on the large graphs. As we can observe, our Lemane shows the best performance on 4 social networks where link prediction finds extensive applications. On

Table 4: Node Classification Micro-F1 (%) on Wikipedia.

Method	0.1	0.3	0.5	0.7	0.9
DeepWalk	42.02	46.12	48.46	49.39	49.35
Node2vec	44.75	48.08	49.82	50.69	50.41
VERSE	38.76	41.92	43.84	44.92	44.31
InfiniteWalk	38.96	42.64	45.94	47.73	48.24
AROPE	45.82	50.59	52.47	53.36	52.16
RandNE	34.51	32.83	43.25	45.55	45.93
ProNE	44.49	50.45	53.15	54.38	54.43
NetSMF	40.29	42.56	43.68	44.08	44.12
STRAP	46.51	50.77	52.44	52.64	52.37
NRP	48.04	52.71	54.39	55.20	54.30
GraphGAN	32.87	35.43	36.47	37.68	37.50
AW	40.70	40.70	40.44	40.30	39.47
NetHiex	45.58	47.95	49.39	49.77	49.20
GraphZoom	40.76	41.03	41.18	41.07	40.54
Louvain	40.86	40.99	40.72	41.25	40.69
Lemane-F	47.79	52.39	54.34	54.67	54.25

Table 5: Node Classification Micro-F1 (%) on BlogCatalog.

Method	0.1	0.3	0.5	0.7	0.9
DeepWalk	33.01	36.97	38.70	39.87	41.31
Node2vec	35.01	37.16	37.97	38.51	39.13
VERSE	32.76	36.32	38.18	39.09	40.71
InfiniteWalk	34.30	38.00	40.21	41.87	43.14
AROPE	29.12	32.78	34.12	34.95	35.77
RandNE	26.75	31.56	36.20	38.34	39.93
ProNE	36.38	40.33	41.56	42.32	42.28
NetSMF	34.95	37.99	39.30	40.19	40.72
STRAP	38.62	41.80	42.96	43.39	43.97
NRP	38.73	41.65	42.36	43.15	43.34
GraphGAN	14.97	17.23	18.81	20.07	21.16
AW	16.52	16.91	16.98	17.25	17.39
NetHiex	37.46	40.06	40.63	41.43	42.33
GraphZoom	22.02	25.59	27.75	29.37	30.70
Louvain	19.16	19.88	21.12	21.30	21.74
Lemane-F	39.64	42.38	43.34	44.03	44.37

the Wikipedia dataset, a co-occurrence network of words appearing in the Wikipedia, our Lemane still achieves high performance and is the best method among all matrix factorization methods. Compared to two state-of-the-art matrix factorization methods STRAP and NRP, our Lemane takes the lead by more than 1% on Wikipedia and Slashdot, and takes the lead by almost 3% on Orkut. This demonstrates the effectiveness of our learning based method.

5.3 Node Classification

Node classification task aims to predict the label(s) of each node based on the embeddings. As we mentioned in Section 3, we first randomly sample 5% labeled nodes for parameter training. Then the classification task is performed with the following steps: *(i)* following [46], for Lemane and other matrix factorization methods [4], we first normalize the embedding vector[5] $\boldsymbol{x}_v$ from $\boldsymbol{X}$ and embedding vector $\boldsymbol{y}_v$ from $\boldsymbol{Y}$ of each node v, and then concatenate them to get the representation of v; *(ii)* for methods without factorization operation, we use the embedding vector of node v as its representation. Specifically, we randomly split the node and label sets into the training set and testing set and the training ratio varies from 10% to 90%. To make a fair comparison, note that if the training ratio is x%, then for Lemane, it will sample an additional $(x-5)$% and include the 5% labeled nodes to train the classifiers. Following previous work [21, 34], we employ a one-vs-rest logistic regression classifier implemented by LIBLINEAR [15] with default parameters for all methods. Micro-F1 score is used as the evaluation metric for the classification task.

For node classification, we test on four datasets Wikipedia, BlogCatalog, TWeibo, and Orkut, which include label information. Table 4 and Table 5 show the performance of our Lemane-F against all 15 methods on the two small datasets: Wikipedia and BlogCatalog, respectively. Table 6 and Table 7 show the performance of our Lemane-S against 6 methods that scale to TWeibo and Orkut.

We make the following observations. Firstly, our Lemane achieves the best Micro-F1 scores on three datasets BlogCatalog, Tweibo, and Orkut in all of the tested training ratios. Besides, compared to STRAP, which takes PPR without training the stopping probabilities, our Lemane achieves more than 1% lead on Wikipedia datasets and up to 3% on the Orkut dataset. Compared to the second-best matrix factorization method NRP, our Lemane further achieves about 1% lead on the BlogCatalog dataset in all of the tested training ratios.

Table 6: Node classification Micro-F1 (%) on TWeibo.

Method	0.1	0.3	0.5	0.7	0.9
AROPE	33.96	34.11	34.18	34.24	34.29
RandNE	34.64	34.67	34.80	34.92	34.96
ProNE	35.27	35.37	35.43	35.48	35.52
STRAP	35.75	35.97	36.03	36.04	36.04
NRP	35.73	35.97	36.03	36.04	36.04
Louvain	34.14	34.29	34.33	34.38	34.42
Lemane-S	35.77	36.03	36.04	36.05	36.05

Table 7: Node classification Micro-F1 (%) on Orkut.

Method	0.1	0.3	0.5	0.7	0.9
AROPE	49.11	50.86	51.55	51.89	52.22
RandNE	44.94	49.35	50.47	51.11	51.53
ProNE	36.09	37.13	38.32	38.81	39.44
STRAP	70.37	73.09	74.04	74.60	75.08
NRP	72.47	75.58	76.69	77.36	77.98
Louvain	29.16	36.00	36.51	36.85	36.63
Lemane-S	73.32	76.27	77.26	77.89	78.14

[4]For matrix factorization based methods, there was some implementation issues in the evaluation code for node classification on directed graphs. We have rerun the experiment for matrix factorization based methods on directed graphs.

[5]We observe some significant improvement on Orkut dataset when normalization is applied. Thus, we take normalization for the embedding vectors on all datasets.

In summary, experimental studies reveal that our Lemane can learn proximity measures that most suit the task in most scenarios. Compared to other matrix factorization methods, e.g., STRAP [48] and NRP [46], that take personalized PageRank as the proximity measure without learning, our Lemane can train the proximity measure, i.e., the supervised PPR, to gain better performance.

6 CONCLUSION

In this paper, we present Lemane that learns trainable proximity measures to best suit the datasets and tasks at hand automatically. Experimental results reveal that Lemane can learn more representative embeddings compared with state-of-the-art approaches.

ACKNOWLEDGMENTS

Sibo Wang is supported by Hong Kong RGC ECS (No. 24203419), RGC CRF (No. C4158-20G), CUHK Direct Grant (No. 4055114), and NSFC (No. U1936205). Zengfeng Huang is supported by Shanghai Sailing Program Grant No. 18YF1401200, Shanghai Science and Technology Commission Grant No. 17JC1420200.

REFERENCES

[1] Kaggle. https://www.kaggle.com/c/kddcup2012-track1/data.
[2] Large text compression benchmark. http://mattmahoney.net/dc/textdata.
[3] SNAP. http://snap.stanford.edu/data/.
[4] Social-Dimension Approach to Classification in Large-Scale Networks. http://leitang.net/social_dimension.html.
[5] Sami Abu-El-Haija, Bryan Perozzi, Rami Al-Rfou, and Alex Alemi. 2018. Watch Your Step: Learning Node Embeddings via Graph Attention. In *NeurIPS*. 9198–9208.
[6] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local Graph Partitioning using PageRank Vectors. In *FOCS*. 475–486.
[7] Ayan Kumar Bhowmick, Koushik Meneni, Maximilien Danisch, Jean-Loup Guillaume, and Bivas Mitra. 2020. LouvainNE: Hierarchical Louvain Method for High Quality and Scalable Network Embedding. In *WSDM*. 43–51.
[8] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2016. Deep Neural Networks for Learning Graph Representations. In *AAAI*. 1145–1152.
[9] Sudhanshu Chanpuriya and Cameron Musco. 2020. InfiniteWalk: Deep Network Embeddings as Laplacian Embeddings with a Nonlinearity. In *SIGKDD*. 1325–1333.
[10] Eli Chien, Jianhao Peng, Pan Li, and Olgica Milenkovic. 2021. Adaptive Universal Generalized PageRank Graph Neural Network. In *ICLR*.
[11] Kenneth L. Clarkson and David P. Woodruff. 2017. Low-Rank Approximation and Regression in Input Sparsity Time. *J. ACM* 63, 6 (2017).
[12] Chenhui Deng, Zhiqiang Zhao, Yongyu Wang, Zhiru Zhang, and Zhuo Feng. 2020. GraphZoom: A Multi-level Spectral Approach for Accurate and Scalable Graph Embedding. In *ICLR*.
[13] Yuxiao Dong, Nitesh V. Chawla, and Ananthram Swami. 2017. Metapath2vec: Scalable Representation Learning for Heterogeneous Networks. In *SIGKDD*. 135–144.
[14] Claire Donnat, Marinka Zitnik, David Hallac, and Jure Leskovec. 2018. Learning Structural Node Embeddings via Diffusion Wavelets. In *SIGKDD*. 1320–1329.
[15] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A Library for Large Linear Classification. *J. Mach. Learn. Res.* 9 (2008), 1871–1874.
[16] Xu Feng, Yuyang Xie, Mingye Song, Wenjian Yu, and Jie Tang. 2018. Fast Randomized PCA for Sparse Data. In *ACML*. 710–725.
[17] Francois Fouss, Alain Pirotte, Jean-michel Renders, and Marco Saerens. 2007. Random-Walk Computation of Similarities between Nodes of a Graph with Application to Collaborative Recommendation. *TKDE* 19 (2007), 355–369.
[18] Tao-yang Fu, Wang-Chien Lee, and Zhen Lei. 2017. HIN2Vec: Explore Meta-Paths in Heterogeneous Information Networks for Representation Learning. In *CIKM*. 1797–1806.
[19] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *NeurIPS*. 2672–2680.
[20] Palash Goyal, Nitin Kamra, Xinran He, and Yan Liu. 2018. DynGEM: Deep Embedding Method for Dynamic Graphs. *CoRR* abs/1805.11273 (2018).
[21] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. In *SIGKDD*. 855–864.
[22] Yupeng Gu, Yizhou Sun, Yanen Li, and Yang Yang. 2018. RaRE: Social Rank Regulated Large-Scale Network Embedding. In *WWW*. 359–368.
[23] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[24] Piotr Indyk, Ali Vakilian, and Yang Yuan. 2019. Learning-Based Low-Rank Approximations. In *NeurIPS*. 7402–7412.
[25] Johannes Klicpera, Stefan Weißenberger, and Stephan Günnemann. 2019. Diffusion Improves Graph Learning. In *(NeurIPS)*. 13333–13345.
[26] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *SysML*.
[27] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Predicting Positive and Negative Links in Online Social Networks. In *WWW*. 641–650.
[28] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.
[29] Jianxin Ma, Peng Cui, Xiao Wang, and Wenwu Zhu. 2018. Hierarchical Taxonomy Aware Network Embedding. In *SIGKDD*. 1920–1929.
[30] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NeurIPS*. 3111–3119.
[31] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *IMC*. 29–42.
[32] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric Transitivity Preserving Graph Embedding. In *SIGKDD*. 1105–1114.
[33] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web. (1999).
[34] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *SIGKDD*. 701–710.
[35] Pascal Pons and Matthieu Latapy. 2005. Computing communities in large networks using random walks. In *ISCIS*. 284–293.
[36] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Chi Wang, Kuansan Wang, and Jie Tang. 2019. NetSMF: Large-Scale Network Embedding As Sparse Matrix Factorization. In *WWW*. 1509–1520.
[37] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. 2018. Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and Node2vec. In *WSDM*. 459–467.
[38] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-Scale Information Network Embedding. In *WWW*. 1067–1077.
[39] Lei Tang and Huan Liu. 2009. Relational Learning via Latent Social Dimensions. In *SIGKDD*. 817–826.
[40] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. DyRep: Learning Representations over Dynamic Graphs. In *ICLR*.
[41] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. 2018. VERSE: Versatile Graph Embeddings from Similarity Measures. In *WWW*. 539–548.
[42] Ke Tu, Peng Cui, Xiao Wang, Philip S. Yu, and Wenwu Zhu. 2018. Deep Recursive Network Embedding with Regular Equivalence. In *SIGKDD*. 2357–2366.
[43] Ke Tu, Jianxin Ma, Peng Cui, Jian Pei, and Wenwu Zhu. 2019. AutoNE: Hyperparameter Optimization for Massive Network Embedding. In *SIGKDD*. 216–225.
[44] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural Deep Network Embedding. In *SIGKDD*. 1225–1234.
[45] Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. 2018. GraphGAN: Graph Representation Learning With Generative Adversarial Nets. In *AAAI*. 2508–2515.
[46] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, and Sourav S. Bhowmick. 2020. Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank. *PVLDB* 13, 5 (2020), 670–683.
[47] Renchi Yang, Xiaokui Xiao, Zhewei Wei, Sourav S. Bhowmick, Jun Zhao, and Rong-Hua Li. 2019. Efficient Estimation of Heat Kernel PageRank for Local Clustering. In *SIGMOD*. 1339–1356.
[48] Yuan Yin and Zhewei Wei. 2019. Scalable Graph Embeddings via Sparse Transpose Proximities. In *SIGKDD*. 1429–1437.
[49] Xiao Yu, Xiang Ren, Yizhou Sun, Quanquan Gu, Bradley Sturt, Urvashi Khandelwal, Brandon Norick, and Jiawei Han. 2014. Personalized Entity Recommendation: A Heterogeneous Information Network Approach. In *WSDM*. 283–292.
[50] Jie Zhang, Yuxiao Dong, Yan Wang, Jie Tang, and Ming Ding. 2019. ProNE: Fast and Scalable Network Representation Learning. In *IJCAI*. 4278–4284.
[51] Shengzhong Zhang, Zengfeng Huang, Haicang Zhou, and Ziang Zhou. 2020. SCE: Scalable Network Embedding from Sparsest Cut. In *SIGKDD*. 257–265.
[52] Ziwei Zhang, Peng Cui, Haoyang Li, Xiao Wang, and Wenwu Zhu. 2018. Billion-Scale Network Embedding with Iterative Random Projection. In *ICDM*. 787–796.
[53] Ziwei Zhang, Peng Cui, Xiao Wang, Jian Pei, Xuanrong Yao, and Wenwu Zhu. 2018. Arbitrary-Order Proximity Preserved Network Embedding. In *SIGKDD*. 2778–2786.
[54] Chang Zhou, Yuqiong Liu, Xiaofei Liu, Zhongyi Liu, and Jun Gao. 2017. Scalable Graph Embedding for Asymmetric Proximity. In *AAAI*. 2942–2948.

A PROOFS

Proof of Theorem 1. We first prove the following lemma.

Lemma 1. *The probability that a supervised random walk stops at exactly the k-th hop is $\alpha_k \prod_{l=0}^{k-1}(1-\alpha_l)$ and for any k, we have*

$$\alpha_k + \sum_{l=1}^{\infty} \alpha_{k+l} \prod_{j=0}^{k+l-1} (1-\alpha_j) = 1.$$

Proof. Define $Prob_k(L) = \alpha_k + \sum_{l=1}^{L} \alpha_{k+l} \prod_{j=0}^{k+l-1}(1-\alpha_j)$. We claim that $1 - Prob_k(L) = \prod_{j=0}^{L}(1-\alpha_{k+j})$, implying that $(1 - Prob_k(L)) \to 0$ when $L \to \infty$ and thus, $Prob_k(L) \to 1$ when $L \to \infty$. Now we justify $1-Prob_k(L) = \prod_{j=0}^{L}(1-\alpha_{k+j})$ by induction. When $L = 0$, $1 - Prob_k(0) = 1 - \alpha_k$ holds obviously. Suppose $1 - Prob_k(t) = \prod_{j=0}^{t}(1-\alpha_{k+j})$ holds. Then for $Prob_k(t+1) = Prob_k(t) + \alpha_{k+t+1}(1 - Prob_k(t))$, we can derive that:

$$\begin{aligned}1 - Prob_k(t+1) &= 1 - Prob_k(t) - \alpha_{k+t+1}(1 - Prob_k(t)) \\ &= (1-\alpha_{k+t+1})(1 - Prob_k(t)) = \prod_{j=0}^{t+1}(1-\alpha_{k+j}).\end{aligned}$$

Thus $1-Prob_k(L) = \prod_{j=0}^{L}(1-\alpha_{k+j})$ holds for any L. Proof done. □

Note that the transition matrix $\boldsymbol{P}$ is a row-stochastic matrix, *i.e.* $\boldsymbol{P}(i,j) \geq 0$ for any $i, j = 1, 2, \cdots, n$, and $\sum_{j=1}^{n} \boldsymbol{P}(i,j) = 1$ for any $i = 1, 2, \cdots, n$. Following these properties, we can easily derive that $\boldsymbol{P}^l$ is also row-stochastic, where l is the power of the matrix. Now we first look at the r.h.s. of the inequality in Theorem 1. Recall that the infinity-norm of matrix is the maximum absolute row sum, *i.e.*, $||\boldsymbol{M}||_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^{n} |\boldsymbol{M}(i,j)|$. Then, by Lemma 1, for any $i = 1, 2, \cdots, n$, we have

$$\begin{aligned}\sum_{j=1}^{n} \boldsymbol{S}(i,j) &= \alpha_0 \sum_{j=1}^{n} \boldsymbol{P}^0(i,j) + \alpha_1(1-\alpha_0)\sum_{j=1}^{n} \boldsymbol{P}^1(i,j) + \cdots \\ &= \alpha_0 + \sum_{l=1}^{\infty} \alpha_l \prod_{k=0}^{l-1}(1-\alpha_k) = 1.\end{aligned}$$

In terms of the l.h.s. of the inequality in Theorem 1, for any i-th row of matrix $\boldsymbol{S} - \boldsymbol{S}_L$, we have that:

$$\begin{aligned}&\sum_{j=1}^{n}(\boldsymbol{S} - \boldsymbol{S}_L)(i,j) \\ &= \sum_{l=L+1}^{\infty} \alpha_l \prod_{k=0}^{l-1}(1-\alpha_k) \sum_{j=1}^{n} \boldsymbol{P}^{L+1}(i,j) = \sum_{l=L+1}^{\infty} \alpha_l \cdot \prod_{k=0}^{l-1}(1-\alpha_k) \\ &= \prod_{k=0}^{L}(1-\alpha_k)(\alpha_{L+1} + \alpha_{L+2}(1-\alpha_{L+1}) + \cdots) \leq \prod_{k=0}^{L}(1-\alpha_k).\end{aligned}$$

By combining the above derivations,

$$\sum_{j=1}^{n}(\boldsymbol{S} - \boldsymbol{S}_L)(i,j) \leq \prod_{k=0}^{L}(1-\alpha_k) \cdot \sum_{j=1}^{n} \boldsymbol{S}(i,j)$$

holds for any $i = 1, 2, \cdots, n$. Thus, it clearly holds that

$$||\boldsymbol{S} - \boldsymbol{S}_L||_\infty \leq \prod_{k=0}^{L}(1-\alpha_k)||\boldsymbol{S}||_\infty.$$

The theorem is proved.

Proof of Theorem 2. The cost of Algorithm 3 is dominated by the number of push operations. Recall that the push operation is invoked only if there exists an entry $\boldsymbol{r}_s^{(k)}(v)$ in residue vectors such that $\boldsymbol{r}_s^{(k)}(v) \geq \delta \cdot d_{out}(v)$. Let $\boldsymbol{\pi}_s^{(l)}(v)$ denote the l-hop supervised PPR whose value is the probability that the l-hop supervised random walk from s stops at v. Then the total number of push operations caused by the residue value $\boldsymbol{r}_s^{(l)}(v)$ can be bounded by $\frac{\boldsymbol{\pi}_s^{(l)}(v)}{\alpha_l \delta \cdot d_{out}(v)}$. In addition, since in each push operation, the node v will pass its residue to its out-neighbors, the cost for a push operation at v is bounded by $O(d_{out}(v))$. Thus, the total cost of the push operations on entry $\boldsymbol{r}_s^{(l)}(v)$ is bounded by $\frac{\boldsymbol{\pi}_s^{(l)}(v)}{\alpha_l \delta \cdot d_{out}(v)} \cdot d_{out}(v)$. Based on the above analysis, the time cost of Algorithm 3 is:

$$\sum_{l=0}^{L} \sum_{v \in V} \frac{\boldsymbol{\pi}_s^{(l)}(v)}{\alpha_l \delta \cdot d_{out}(v)} \cdot d_{out}(v) \leq \frac{1}{\alpha_{min}\delta} \sum_{l=0}^{L} \sum_{v \in V} \boldsymbol{\pi}_s^{(l)}(v) \leq \frac{1}{\alpha_{min}\delta},$$

where $\alpha_{min} = \min\{\alpha_1, \cdots, \alpha_L\}$ can be treated as a constant. Thus, Algorithm 3 runs in $O(\frac{1}{\alpha_{min}\delta}) = O(\frac{1}{\delta})$ time and the proof is done.

Proof of Theorem 3. The following lemmas are used in the proof.

Lemma 2. [35] *For undirected graph G and any two nodes u and v in G, let $\boldsymbol{P}^k(u,v)$ (resp. $\boldsymbol{P}^k(v,u)$) be the probability of going from u to v (resp. from v to u) through a random walk of fixed length k. Then,*

$$\frac{\boldsymbol{P}^k(u,v)}{d_{out}(v)} = \frac{\boldsymbol{P}^k(v,u)}{d_{out}(u)}.$$

Lemma 3. *Let $\boldsymbol{\pi}_s^L(u)$ denote the supervised PPR within L hops. Then, after the end of every iteration in Algorithm 3, for $\hat{\boldsymbol{\pi}}_s(u)$ and the residue vectors $\boldsymbol{r}_s^{(0)}, \cdots \boldsymbol{r}_s^{(L)}$, we have*

$$\boldsymbol{\pi}_s^L(u) \leq \hat{\boldsymbol{\pi}}_s(u) + \sum_{k=0}^{L} \sum_{v \in V} \boldsymbol{r}_s^{(k)}(v) \cdot h_u^k(v), \tag{8}$$

where $h_u^k(v) = \alpha_k \boldsymbol{e}_u(v) + \sum_{l=1}^{\infty} \alpha_{k+l} \prod_{j=k}^{k+l-1}(1-\alpha_j)\boldsymbol{P}^l(v,u)$, and $\boldsymbol{e}_u$ is a unit vector in which the u-th entry of $\boldsymbol{e}_u$ is 1 and others are all 0.

Proof. We prove Lemma 3 by induction. First, at the begin of Algorithm 3, vector $\hat{\boldsymbol{\pi}}_s$ and all residue vectors are set to $\boldsymbol{0}$ except for $\boldsymbol{r}_s^{(0)}(s) = 1$. Then we have

$$\begin{aligned}&\hat{\boldsymbol{\pi}}_s(u) + \sum_{k=0}^{K} \sum_{v \in V} \boldsymbol{r}_s^{(k)}(v) \cdot h_u^k(v) \\ &= \alpha_0 + \sum_{l=1}^{\infty} \alpha_l \prod_{j=0}^{l-1}(1-\alpha_j)\boldsymbol{P}^l(s,u) = \boldsymbol{\pi}_s(u),\end{aligned}$$

which implies that Equation 8 holds under the initial condition since $\boldsymbol{\pi}_s^L(u) \leq \boldsymbol{\pi}_s(u)$. Now suppose that Equation 8 holds at the end of i-th iteration. Suppose further that during $(i+1)$-th iteration, entry $\boldsymbol{r}_s^{(w)}(q) \geq \delta \cdot d_{out}(q)$ is detected and push operation is invoked here. If the change of l.h.s of Equation 8 after the push operation is zero, then Equation 8 holds at the end of $(l+1)$-th iteration.

Table 8: Hyperparameters of Lemane for link prediction.

Dataset	Hyperparameters
Wikipedia	initialization: $\phi_{hk}(l)$ with $t = 5$, learning rate: 0.001, δ: 10^{-5}, β: 0.01, γ: 1, SVD used for push: JacobiSVD
Wikivote	initialization: $\phi_{hk}(l)$ with $t = 1$, learning rate: 0.5, δ: 10^{-6}, β: 0.5, γ: 1, SVD used for push: frPCA
BlogCatalog	initialization: $\phi_{ge}(l)$ with $\alpha = 0.5$, learning rate: 0.1, δ: 10^{-7}, β: 0.01, γ: 1, SVD used for push: frPCA
Slashdot	initialization: $\phi_{hk}(l)$ with $t = 5$, learning rate: 0.001, δ: 10^{-5}, β: 0.1, γ: 1, SVD used for push: frPCA
Tweibo	initialization: $\phi_{ge}(l)$ with $\alpha = 0.5$, learning rate: 0.01, δ: 10^{-5}, β: 0.1, γ: 1, SVD used for push: frPCA
Orkut	initialization: $\phi_{hk}(l)$ with $t = 1$, learning rate: 0.01, δ: 10^{-4}, β: 1, γ: 1, SVD used for push: frPCA

Table 9: Hyperparameters of Lemane for node classification.

Dataset	Hyperparameters
Wikipedia	initialization: $\phi_{hk}(l)$ with $t = 5$, learning rate: 0.05, δ: 10^{-5}, β': 1, γ': 0.5, SVD used for push: JacobiSVD
BlogCatalog	initialization: $\phi_{hk}(l)$ with $t = 5$, learning rate: 0.01, δ: 10^{-5}, β': 1, γ': 0.5, SVD used for push: JacobiSVD
TWeibo	initialization: $\phi_{hk}(l)$ with $t = 5$, learning rate: 0.05, δ: 10^{-5}, β': 1, γ': 3, SVD used for push: frPCA, X and Y are normalized before concatenation in loss function $\mathcal{L}_1'$
Orkut	initialization: $\phi_{hk}(l)$ with $t = 1$, learning rate: 0.5 , δ: 10^{-5}, β': 1, γ': 2, SVD used for push: frPCA, X and Y are normalized before concatenation in loss function $\mathcal{L}_1'$

For the convenience of analysis of the change, let $\hat{\boldsymbol{\pi}}'_s, \dot{\boldsymbol{r}}_s^{(w)}$ (resp. $\hat{\boldsymbol{\pi}}''_s, \ddot{\boldsymbol{r}}_s^{(w)}$) be the corresponding vectors at the end of i-th iteration(resp. $(i+1)$-th iteration). Then after performing the push operation in the $(i+1)$-th ieration, we have

$$\hat{\boldsymbol{\pi}}''_s(q) - \hat{\boldsymbol{\pi}}'_s(q) = \alpha_w \cdot \dot{\boldsymbol{r}}_s^{(w)}(q),$$
$$\ddot{\boldsymbol{r}}_s^{(w)}(q) - \dot{\boldsymbol{r}}_s^{(w)}(q) = -\dot{\boldsymbol{r}}_s^{(w)}(q).$$

Recap $N(q)$ is the set of q's out-neighbors, for $o \in N(q)$, we have

$$\ddot{\boldsymbol{r}}_s^{(w+1)}(o) - \dot{\boldsymbol{r}}_s^{(w+1)}(o) = (1-\alpha_w)\frac{\dot{\boldsymbol{r}}_s^{(w)}(q)}{d_{out}(q)}.$$

Then the change between i-th iteration and $(i+1)$-th iteration is

$$\hat{\boldsymbol{\pi}}''_s(u) - \hat{\boldsymbol{\pi}}'_s(u) + \ddot{\boldsymbol{r}}_s^{(w)}(q)\cdot h_u^w(q) - \dot{\boldsymbol{r}}_s^{(w)}(q)\cdot h_u^w(q) + \sum_{o\in N(q)} \ddot{\boldsymbol{r}}_s^{(w+1)}(o)\cdot h_u^{w+1}(o) - \sum_{o\in N(q)} \dot{\boldsymbol{r}}_s^{(w+1)}(o)\cdot h_u^{w+1}(o)$$

$$= \hat{\boldsymbol{\pi}}''_s(u) - \hat{\boldsymbol{\pi}}'_s(u) - \dot{\boldsymbol{r}}_s^{(w)}(q)\cdot h_u^w(q) + (1-\alpha_w)\frac{\dot{\boldsymbol{r}}_s^{(w)}(q)}{d_{out}(q)}\sum_{o\in N(q)} h_u^{w+1}(o)$$

$$= \hat{\boldsymbol{\pi}}''_s(u) - \hat{\boldsymbol{\pi}}'_s(u) - \dot{\boldsymbol{r}}_s^{(w)}(q)\cdot h_u^w(q) + \dot{\boldsymbol{r}}_s^{(w)}(q)(h_u^w(q) - \alpha_w \boldsymbol{e}_u(q))$$

$$= \hat{\boldsymbol{\pi}}''_s(u) - \hat{\boldsymbol{\pi}}'_s(u) - \alpha_w \boldsymbol{e}_u(q)\dot{\boldsymbol{r}}_s^{(w)}(q) = 0.$$

Therefore, the change of l.h.s of Equation 8 after $(i+1)$-th iteration is zero, which implies that Equation 8 also holds at the end of $(i+1)$-th iteration. The lemma is proved. □

Next, we prove Theorem 3. By Equation 8 in Lemma 3, we have:

$$|\boldsymbol{\pi}_s^L(u) - \hat{\boldsymbol{\pi}}_s(u)|/d_{out}(u) \le \sum_{k=0}^{L}\sum_{v\in V}\frac{\boldsymbol{r}_s^{(k)}(v)}{d_{out}(u)}\cdot h_u^k(v)$$

$$\le \delta \sum_{k=0}^{L}\sum_{v\in V}(\alpha_k \boldsymbol{e}_u(v) + \sum_{l=1}^{\infty}\alpha_{k+l}\prod_{j=k}^{k+l-1}(1-\alpha_j)\boldsymbol{P}^l(v,u))$$

$$= \delta \sum_{k=0}^{L}(\alpha_k + \sum_{l=1}^{\infty}\alpha_{k+l}\prod_{j=k}^{k+l-1}(1-\alpha_j)) = L\cdot\delta,$$

where the last equation holds by Lemma 1. The theorem is proved.

Proof of Theorem 4. The time complexities of Algorithm 4 depend on two parts: Generalized Push and SparseSVD. From the results in Theorem 2, for each source $s \in V$, the cost of the push operations is $O(\frac{1}{\delta})$. Thus, the total cost of push operations on n nodes is bounded by $O(\frac{n}{\delta})$. Then, we need the following theorem [11] to analyze the cost of SparseSVD.

Theorem 5. *Let $\boldsymbol{A}$ denote an $n\times n$ matrix, there is an algorithm that, with failure probability $1/10$, finds two $n\times d$ matrices $\boldsymbol{U},\boldsymbol{V}$ with orthonormal columns, and a $d\times d$ diagonal matrix Σ, so that $\left\|\boldsymbol{A} - \boldsymbol{U}\Sigma\boldsymbol{V}^T\right\|_F \le (1+\epsilon)\,||\boldsymbol{A} - [\boldsymbol{A}]_d||_F$, where $[\boldsymbol{A}]_d$ denotes the best rank-d approximation to $\boldsymbol{A}$. The algorithm runs in time*

$$O\left(nnz(\boldsymbol{A}) + \tilde{O}\left(nd^2\epsilon^{-4} + d^3\epsilon^{-5}\right)\right).$$

Racall that for any approximate supervised PPR score $\hat{\boldsymbol{\pi}}_u(v)$, it is add to $\boldsymbol{S}$ only if it is larger than the error parameter δ. Thus the total number of non-zero entries in $\boldsymbol{S}$ is $\text{nnz}(\boldsymbol{S}) = O(\frac{n}{\delta})$. The cost of SparseSVD is bounded by $O(\frac{n}{\delta} + \frac{nd^2}{\epsilon^4})$. Finally, combining these two parts, the running cost of Algorithm 4 is bounded by $O(\frac{n}{\delta} + \frac{nd^2}{\epsilon^4})$, which completes our proof.

B HYPER-PARAMETERS SETTINGS

Table 8 and Table 9 summarize the hyperparameter settings of Lemane on each dataset. The searching hyperparameters include initialized distribution $\phi(l)$ to indicate the probability that the supervised random walk stops at the l-th hop, the loss function coefficients $\beta, \gamma, \beta', \gamma'$, error parameter δ, learning rate, and buildin SVDs used in generalized push. For the initialization of distribution $\phi(l)$, two standard distribution functions are used, namely the geometric distribution $\phi_{ge}(l) = \alpha(1-\alpha)^l$ and the Poisson distribution $\phi_{hk}(l) = \frac{e^{-t}t^l}{l!}$.