Semantic Proximity Search on Heterogeneous Graph by Proximity Embedding

Zemin Liu[1], Vincent W. Zheng[2], Zhou Zhao[1], Fanwei Zhu[3], Kevin Chen-Chuan Chang[4], Minghui Wu[3]*, Jing Ying[1]

[1] Zhejiang University, China; [2] Advanced Digital Sciences Center, Singapore;
[3] Zhejiang University City College, China; [4] University of Illinois at Urbana-Champaign, USA

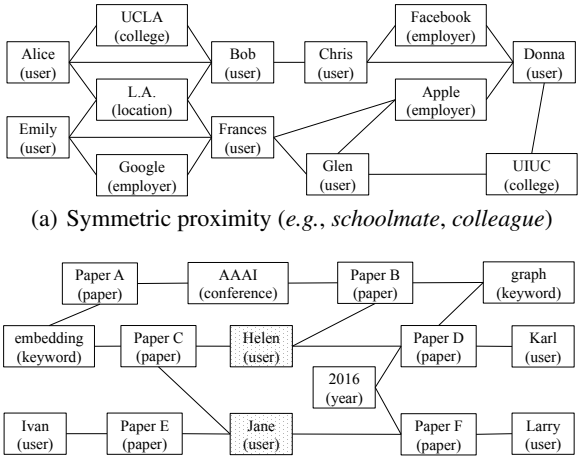
Abstract

Many real-world networks have a rich collection of objects. The semantics of these objects allows us to capture different classes of proximities, thus enabling an important task of semantic proximity search. As the core of semantic proximity search, we have to measure the proximity on a heterogeneous graph, whose nodes are various types of objects. Most of the existing methods rely on engineering features about the graph structure between two nodes to measure their proximity. With recent development on graph embedding, we see a good chance to avoid feature engineering for semantic proximity search. There is very little work on using graph embedding for semantic proximity search. We also observe that graph embedding methods typically focus on embedding nodes, which is an "indirect" approach to learn the proximity. Thus, we introduce a new concept of proximity embedding, which directly embeds the network structure between two possibly distant nodes. We also design our proximity embedding, so as to flexibly support both symmetric and asymmetric proximities. Based on the proximity embedding, we can easily estimate the proximity score between two nodes and enable search on the graph. We evaluate our proximity embedding method on three real-world public data sets, and show it outperforms the state-of-the-art baselines. We release the code for proximity embedding[1].

Introduction

Many real-world networks have a rich collection of objects. For example, in the social networks such as Facebook or LinkedIn, there are "users", "locations", "employers", "colleges". In the academic networks such as DBLP, there are "authors", "papers", "conferences" and "years". Such objects can be linked together to form a heterogeneous graph. For example, in Fig. 1(a) and Fig. 1(b), we show two object graphs based on LinkedIn data and DBLP data, respectively.

The rich semantics of objects allows us to capture multiple classes of proximity on the graph. For example, in Fig. 1(a) Bob and Alice are *schoolmates* as they are attending the same college, whereas Glen and Donna are *colleagues* as they are working for the same company. Being able to differentiate multiple classes of proximity is very useful. For example, we can do circle-based friend suggestion (*e.g.*, who is my schoolmate / colleague?) in the social network, and author role prediction (*e.g.*, who is my advisor / advisee?) in the bibliography network.



(a) Symmetric proximity (*e.g.*, *schoolmate*, *colleague*)

(b) Asymmetric proximity (*e.g.*, *advisor*, *advisee*)

Figure 1: Examples of semantic proximity search.

In this paper, we study an important task, *semantic proximity search* on the graph (Sun et al. 2011; Fang et al. 2016b). Specifically, given a proximity class (*e.g.*, *schoolmate*) and a query node (*e.g.*, Bob) of a particular type (*e.g.*, "user") on the graph, semantic proximity search aims to output a ranking list of other nodes of the same types. It is worth noting that, the proximity can be either symmetric, such as *schoolmate* and *colleague* in Fig. 1(a), or asymmetric, such as *advisor* and *advisee* in Fig. 1(b).

The core problem of semantic proximity search is how to measure the proximity on a heterogeneous graph with various types of objects. Most existing methods try to measure the proximity between a query node q and a target node v by *engineering features* about the graph structure. For example, Metapath (Sun et al. 2011) counts the number of meta-paths, which match some deliberately designed path patterns based on the node types (*e.g.*, "author"–"paper"–"author"), between q and v. Metagraph (Fang et al. 2016b) counts the number of meta-graphs, which match some deliberately designed subgraph patterns based on the node

*Corresponding author

[1]https://bitbucket.org/vwz/aaai2017-proxembed/

types (*e.g.*, "user"–"school" & "major"–"user"). With the recent advance on graph embedding (Perozzi, Al-Rfou, and Skiena 2014; Tang et al. 2015; Grover and Leskovec 2016), we now have the chance to avoid feature engineering in measuring the proximity between nodes. However, most of the existing graph embedding methods focus on a *single node*, instead of *two (possibly distant) nodes*. To be precise, graph embedding typically aims to produce a low-dimensional vector for each single node. Any further graph analytics tasks, such as node classification (Yang, Cohen, and Salakhutdinov 2016), link prediction (Wang, Cui, and Zhu 2016) and clustering (Yang et al. 2016), are performed based on the node embedding. Very few works attempt to learn embedding for more than one node (Luo et al. 2015; Zheng et al. 2016), and they are not for proximity search.

Embedding nodes is an *indirect* approach to learn proximity, because it lacks an explicit representation for the network structure between two possibly distant nodes. Typically, node embedding exploits the network structure by "neighbors", such that two nodes as immediate neighbors (Tang et al. 2015) or sharing common neighbors (Perozzi, Al-Rfou, and Skiena 2014) have similar embeddings. Thus, node embedding implicitly includes the network structure. However, node embedding suffers from two major problems: 1) difficult to handle heterogeneous graphs. All the neighbor nodes are of various types, thus trivially treating them as the same is unlikely to be effective; 2) having a gap between node embedding and proximity score. There are many different ways to compute a proximity score based on two node embedding vectors, *e.g.*, using Euclidean distance, or applying a weight matrix (Fang et al. 2016a) or weight vector (Grover and Leskovec 2016) to convert two vectors into a score. It is not clear what the best way is.

We propose ProxEmbed, a *direct* proximity embedding method for semantic proximity search. In ProxEmbed, we aim to learn a vector as the embedding for the network structure between two possibly distant nodes; then we can straightforwardly compute a proximity score by multiplying the embedding with a weight vector. Motivated by the recent graph embedding methods (Perozzi, Al-Rfou, and Skiena 2014; Grover and Leskovec 2016), we use random walk paths between two nodes q and v as their connecting network structure. ProxEmbed takes these paths as input to learn the proximity embedding for q and v. We differentiate the paths from q to v and those from v to q, such that we can generate different proximity embeddings for (q, v) and (v, q) for asymmetric proximity. Although some recent work starts to consider asymmetric proximity in embedding (Ou et al. 2016), it still tries to embed single nodes. To embed multiple paths with various types of nodes and varying lengths into a vector, we devise a Long Short-Term Memory (Hochreiter and Schmidhuber 1997) recurrent neural network architecture with *discounted path pooling*, where we pool embeddings from all the paths with discounts w.r.t. the path length. To supervise the proximity embedding, we use some training tuples of ranking nodes w.r.t. a query node under a semantic class. Then, for each query node, we compute proximity scores for its target nodes based on their proximity embeddings, and use them to construct the ranking loss against the ground truth. Finally, we optimize the ranking loss together with the proximity embedding.

We summarize our contributions as follows.

- So far as we know, ProxEmbed is the first direct proximity embedding method for semantic proximity search. It is flexible for both symmetric and asymmetric proximities.
- We evaluate ProxEmbed on three real-world data sets with totally six types of proximity. We improve the state-of-the-art baselines by at least 1.0%–18.9% (NDCG), 5.3%–82.7% (MAP) on symmetric proximity, and 9.8%–15.6% (NDCG), 4.7%–14.5% (MAP) on asymmetric proximity, when using 1,000 training samples.

Related Work

There is little work that uses graph embedding for semantic proximity search. Earlier graph proximity search work such as Personalized PageRank (Jeh and Widom 2003) and SimRank (Jeh and Widom 2002) does not differentiate semantic classes. Recent work starts to consider rich network structure (Backstrom and Leskovec 2011; Sun et al. 2011; Fang et al. 2016b), but their network structures are engineered, thus often incomprehensive and time consuming. Graph embedding is an automatic graph feature learning technique. However, most graph embedding work focuses on node classification (Perozzi, Al-Rfou, and Skiena 2014; Yang, Cohen, and Salakhutdinov 2016), link prediction (Wang, Cui, and Zhu 2016; Grover and Leskovec 2016) and clustering (Yang et al. 2016), but rarely in proximity search.

There is little work that considers embedding for more than one node in the graph. Most graph embedding methods focus on single node embedding; *e.g.*, earlier methods, such as MDS (Cox and Cox 2000), LLE (Roweis and Saul 2000), IsoMap (Tenenbaum, de Silva, and Langford 2000) and Laplacian eigenmap (Belkin and Niyogi 2001), typical rely on eigendecomposition of the graph affinity matrices to find the leading eigenvectors as each node's embedding. Similarly, latent eigenmodel (Hoff 2008) also looks for a latent vector for each node in network modeling. More recent methods use neural networks to learn node embedding by either shallow architectures (Yang, Cohen, and Salakhutdinov 2016; Xie et al. 2016; Tang et al. 2015) or deep architectures (Niepert, Ahmed, and Kutzkov 2016; Wang, Cui, and Zhu 2016; Chang et al. 2015). There are some attempts to learn edge embedding (Luo et al. 2015) and community embedding (Zheng et al. 2016), but they do not directly encode the network structure between two nodes.

There is little work that handles both symmetric and asymmetric proximities in graph embedding. Earlier methods (Tenenbaum, de Silva, and Langford 2000; Belkin and Niyogi 2001) focus on undirected graphs and do not consider asymmetric proximity. Recent methods consider directed graphs (Perrault-Joncas and Meila 2011; Ou et al. 2016) for asymmetric proximity, but they focus on node embedding and cannot support symmetric proximity together. We handle asymmetric proximity based on the sequence order modeling power of LSTM (Hochreiter and Schmidhuber 1997). LSTM has shown significant improvement in many sequence modeling tasks (Sutskever, Vinyals, and Le 2014;

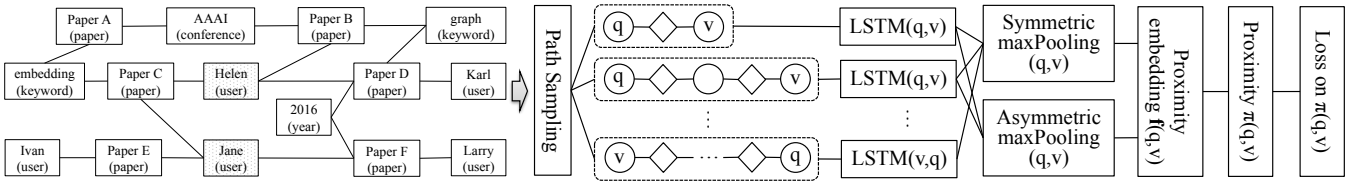


Figure 2: Overall training framework for ProxEmbed.

Le and Mikolov 2014). We novelly use LSTM in the graph setting to model the network structure between two nodes, and we devise an architecture with discounted path pooling to produce a single vector as proximity embedding.

Problem Formulation

As our semantic proximity task is based on a heterogeneous graph where nodes are typed, we first define:

Definition 1 (Typed graph) *A typed graph is* $G = (V, E, C, \tau)$*, where* V *is the set of nodes and* E *is the set of edges.* $C = \{c_1, ..., c_K\}$ *is the set of possible node types, and* $\tau : V \to C$ *is a type mapping function for* G*.*

For example, in Fig. 1(a), we have $C = \{$"user", "college", "location", "employer"$\}$, and τ(Alice) = "user", τ(UCLA) = "college", τ(L.A.) = "location", τ(Apple) = "employer".

As inputs for our proximity embedding problem, we are given a typed graph G and a set of training tuples $\mathcal{D} = \{(q_i, v_i, u_i) : i = 1, ..., m\}$, where for each query node q_i, node v_i is closer to q_i than node u_i. Following the same setting with the existing semantic proximity search work (Fang et al. 2016b), q_i's, v_i's and u_i's are of the same type, *e.g.*, all are "users" in our experiments. As output, we want to generate a vector as the proximity embedding for each (q_i, v_i) and a vector for each (q_i, u_i), $\forall i = 1, ..., m$. Generally, we denote the proximity embedding for a query node $q \in V$ and a target node $v \in V$ as $\mathbf{f}(q, v) \in \mathbb{R}^d$, where d is the proximity embedding dimension. Note that, for symmetric proximity, we have $\mathbf{f}(q, v) = \mathbf{f}(v, q)$; for asymmetric proximity, we have $\mathbf{f}(q, v) \neq \mathbf{f}(v, q)$. Then we define a proximity score between q and v based on the proximity embedding as

$$\pi(q, v) = \boldsymbol{\theta}^T \mathbf{f}(q, v), \tag{1}$$

where $\boldsymbol{\theta} \in \mathbb{R}^d$ is a parameter vector. Finally, based on the embeddings $\mathbf{f}(q_i, v_i)$ and $\mathbf{f}(q_i, u_i)$ for each training tuple (q_i, v_i, u_i), we can evaluate the ranking loss with their corresponding proximity scores $\pi(q_i, v_i)$ and $\pi(q_i, u_i)$.

Proximity Embedding

We summarize the overall training workflow for ProxEmbed in Fig. 2. Given a typed graph G as input, we first sample paths for future extraction of the connecting structure between two nodes. Similar to (Perozzi, Al-Rfou, and Skiena 2014), starting from each node, we randomly sample γ paths, each of length ℓ. In the end, we have a set of paths, denoted as $\mathcal{P}$. We are also given a set of training tuples $\{(q_i, v_i, u_i) : i = 1, ..., m\}$ as input. For each query node $q \in \{q_1, ..., q_m\}$ and a corresponding target node $v \in \{v_1, ..., v_m, u_1, ..., u_m\}$, we extract multiple subpaths from $\mathcal{P}$. We see each path as a sequence with a particular node order. We denote the subpaths starting from q and ending at v in $\mathcal{P}$ as $\mathcal{P}(q, v)$, and those from v to q as $\mathcal{P}(v, q)$. The paths in $\mathcal{P}(q, v)$ and $\mathcal{P}(v, q)$ have varying lengths and different types of nodes. To model each path, we adopt LSTM (Sutskever, Vinyals, and Le 2014) to embed the path into a vector (modeling details to be discussed next). Then, we do *discounted path pooling* over $\mathcal{P}(q, v)$ and $\mathcal{P}(v, q)$ to finally generate a proximity embedding for each pair (q, v). For asymmetric proximity, we use max pooling over only $\mathcal{P}(q, v)$ to get the proximity embedding $\mathbf{f}(q, v)$; whereas for symmetric proximity, we use max pooling over both $\mathcal{P}(q, v)$ and $\mathcal{P}(v, q)$ to get $\mathbf{f}(q, v)$. Because the paths are of varying length, to emphasize more on the shorter paths, we introduce a discount for each path's LSTM output vector based on the path length during max pooling. As a result, the shorter paths contribute more to the final proximity embedding for (q, v). Finally, given the proximity embedding, we compute a proximity score $\pi(q, v)$, and later use it to evaluate the ranking loss against the ground truth training tuples.

In the following, we introduce how to model each path by LSTM and how to enforce discounted path pooling for the final proximity embedding between two nodes.

Path modeling with LSTM. A LSTM is an architecture designed for recurrent neural network to address the vanishing/exploding gradient issue (Hochreiter and Schmidhuber 1997). In general, LSTM takes as input a sequence $(\mathbf{x}_1, ..., \mathbf{x}_T)$, where $\mathbf{x}_t \in \mathbb{R}^n$ is a feature vector at timestep t. As intermediate output, LSTM generates a vector $\mathbf{y}_t \in \mathbb{R}^d$ for each timestep. By furthering pooling all the $\mathbf{y}_t$'s, we can output a vector as the embedding for the sequence.

The structure of LSTM is a *memory cell*, which consists of an input gate, a neuro with a self-recurrent connection, a forget gate and an output gate. Generally, the input gate allows incoming signal to alter the state of the memory cell or block it. The self-current connection balance the signals from the previous timestep and the current timestep. The forget gate modulates the memory cells self-recurrent connection, allowing the cell to remember or forget its previous state, as needed. The output gate allows the state of the memory cell to affect other neurons or prevent it. In the following, we summarize the modeling details for LSTM.

Input gate: denote $W^{(i)} \in \mathbb{R}^{d \times n}$, $U^{(i)} \in \mathbb{R}^{d \times d}$ as weight matrices, $\mathbf{r}^{(i)} \in \mathbb{R}^d$ as a bias vector. LSTM computes the input gate activation vector $\mathbf{g}_t^{(i)} \in \mathbb{R}^d$ as: $\forall j = 1, ..., d$,

$$\mathbf{g}_{t,j}^{(i)} = \sigma(W_{j,*}^{(i)} \mathbf{x}_t + U_{j,*}^{(i)} \mathbf{y}_{t-1} + \mathbf{r}_j^{(i)}). \tag{2}$$

Forget gate: denote $W^{(f)} \in \mathbb{R}^{d\times n}$, $U^{(f)} \in \mathbb{R}^{d\times d}$ as weight matrices, $\mathbf{r}^{(f)} \in \mathbb{R}^d$ as a bias vector. LSTM computes the forget gate activation vector $\mathbf{g}_t^{(f)} \in \mathbb{R}^d$ as: $\forall j = 1, ..., d$,

$$\mathbf{g}_{t,j}^{(f)} = \sigma(W_{j,*}^{(f)}\, \mathbf{x}_t + U_{j,*}^{(f)}\, \mathbf{y}_{t-1} + \mathbf{r}_j^{(f)}). \quad (3)$$

Cell state: denote $W^{(c)} \in \mathbb{R}^{d\times n}$, $U^{(c)} \in \mathbb{R}^{d\times d}$ as weight matrices, $\mathbf{r}^{(c)} \in \mathbb{R}^d$ as a bias vector. LSTM computes the cell state activation vector $\mathbf{g}_t^{(c)} \in \mathbb{R}^d$ as: $\forall j = 1, ..., d$,

$$\mathbf{g}_{t,j}^{(c)} = \mathbf{g}_{t,j}^{(i)} \cdot \tilde{\mathbf{g}}_{t,j}^{(c)} + \mathbf{g}_{t,j}^{(f)} \cdot \mathbf{g}_{t-1,j}^{(c)}, \quad (4)$$

where $\tilde{\mathbf{g}}_{t,j}^{(c)} = tanh(W_{j,*}^{(c)}\, \mathbf{x}_t + U_{j,*}^{(c)}\, \mathbf{y}_{t-1} + \mathbf{r}_j^{(c)})$.
Output gate: denote $W^{(o)} \in \mathbb{R}^{d\times n}$, $U^{(o)} \in \mathbb{R}^{d\times d}$ as weight matrices, $\mathbf{r}^{(o)} \in \mathbb{R}^d$ as a bias vector. LSTM computes the output gate activation vector $\mathbf{g}_t^{(o)} \in \mathbb{R}^d$ as: $\forall j = 1, ..., d$,

$$\mathbf{g}_{t,j}^{(o)} = \sigma(W_{j,*}^{(o)}\, \mathbf{x}_t + U_{j,*}^{(o)}\, \mathbf{y}_{t-1} + \mathbf{r}_j^{(o)}), \quad (5)$$

Finally, the output vector $\mathbf{y}_t \in \mathbb{R}^d$ at time t is: $\forall j = 1, ..., d$,

$$\mathbf{y}_{t,j} = \mathbf{g}_{t,j}^{(o)} \cdot tanh(\mathbf{g}_{t,j}^{(c)}). \quad (6)$$

By doing max pooling over all the $\mathbf{y}_t$'s, $\forall t = 1, ..., T$, we obtain an embedding $\mathbf{h} \in \mathbb{R}^d$ for a sequence:

$$\mathbf{h} = maxPooling(\{\mathbf{y}_t : t = 1, ..., T\}). \quad (7)$$

In our path modeling, we consider each path as a sequence of nodes, and each node v has a feature vector $\mathbf{x} \in \mathbb{R}^n$. In this paper, we consider $\mathbf{x}$ as a concatenation of the following four types of observations: 1) *node type*, which is a K-dimensional vector. In this vector, only the dimension corresponding to v's node type is one, and all the others are zeros.; 2) *node degree*, which is a scalar; 3) *distribution of neighbors' types*, which is also a K-dimensional vector. In this vector, each dimension records the number of v's neighbors having a particular node type. We further take logarithm over this vector (plus 1 in each dimension to avoid ill definition for logarithm); 4) *entropy of neighbors' types*, which is a scalar computed from the distribution of neighbors' types.

Discounted path pooling. There are many paths between a query node q and a target node v. To alleviate the bias of some nodes possessing much more paths than the other nodes, we need to aggregate the paths to uniquely define the proximity embedding for each (q, v). Moreover, the paths are of varying length. Generally, the longer a path is, the smaller proximity it implies. This motivates us to introduce a discount factor w.r.t. the path length when we aggregate the paths. Formally, for each path $s \in \mathcal{P}(q, v)$, we denote $|s|$ as the length of s. Next we need to aggregate the paths. For a unified notation, we introduce $\mathcal{Q}(q, v)$ as the path set between q and v. For asymmetric proximity, we have $\mathcal{Q}(q, v) = \mathcal{P}(q, v)$ as the paths from q to v. For asymmetric proximity, we have $\mathcal{Q}(q, v) = \mathcal{P}(q, v) \cup \mathcal{P}(v, q)$ as the paths from q to v and from v to q. Finally, to generate a proximity embedding for (q, v), we aggregate all the path embeddings $\mathbf{h}_s$'s between q and v in $\mathcal{Q}(q, v)$ by max pooling.

$$\mathbf{f}(q, v) = maxPooling(\{\mathbf{h}_s \cdot e^{-\alpha|s|} : s \in \mathcal{Q}(q, v)\}), \quad (8)$$

Algorithm 1 ProxEmbed

Require: typed graph $G = (V, E, C, \tau)$, training tuples $\mathcal{D} = \{(q_i, v_i, u_i)\}$, number of paths per node γ, walk length ℓ, embedding dimension d, parameters $\{\alpha, \beta, \gamma\}$.
Ensure: proximity embedding model parameters Θ.

```
1: Initialize a path set P = ∅;
2: for all v ∈ V do
3:    for i = 1 : γ do
4:       P ← P ∪ SamplePath(G, v, ℓ);
5:    end for
6: end for
7: B ← GenerateBatches(D);
8: for all batch b ∈ B do
9:    Initialize loss for batch b as L_b = 0;
10:   for all each (q, v, u) ∈ b do
11:      f(q, v) ← GetProxEmbedding(P, q, v, d, α);
12:      f(q, u) ← GetProxEmbedding(P, q, u, d, α);
13:      L_b = L_b + ℓ(π(q, v), π(q, u)), based on Eq.9;
14:   end for
15:   L_b = L_b + μΩ(Θ);
16:   Update Θ based on L_b by gradient descent.
17: end for
```

where $\alpha > 0$ is a parameter controlling the path discount. The bigger α is, the more we favor shorter paths.

After getting the proximity embedding for (q, v), we then compute its proximity score $\pi(q, v)$ by Eq. 1. In training, for each tuple (q_i, v_i, u_i), $\forall i = 1, ..., m$, we define a ranking loss based on the proximity scores $\pi(q_i, v_i)$ and $\pi(q_i, u_i)$. There are different forms of ranking loss; in this paper, we can define it as the logarithm of a logistic function

$$\ell(\pi(q_i, v_i), \pi(q_i, u_i)) = -\log \sigma_\beta(\pi(q_i, v_i) - \pi(q_i, u_i)), \quad (9)$$

where $\sigma_\beta(x) = 1/(1 + e^{-\beta x})$ and $\beta > 0$ is a parameter. The bigger β is, the more $\ell(\pi(q_i, v_i), \pi(q_i, u_i))$ contributes to the proximity embedding.

Denote our parameters as $\Theta = \{\boldsymbol{\theta}, W^{(i)}, U^{(i)}, \mathbf{r}^{(i)}, W^{(f)}, U^{(f)}, \mathbf{r}^{(f)}, W^{(c)}, U^{(c)}, \mathbf{r}^{(c)}, W^{(o)}, U^{(o)}, \mathbf{r}^{(o)}\}$. Finally, our ultimate goal becomes minimizing

$$L(\Theta) = \sum\nolimits_{i=1}^{m} \ell(\pi(q_i, v_i), \pi(q_i, u_i)) + \mu\, \Omega(\Theta), \quad (10)$$

where $\mu > 0$ is a trade-off parameter, $\Omega(\cdot)$ is a regularization function (*e.g.*, the sum of l_2-norm for each parameter in Θ).

Algorithm. We summarize ProxEmbed in Alg. 1. In lines 1–6, we sample paths on the graph. In line 7, we split the training tuples into batches, and then do batch stochastic gradient descent. In lines 10–14, we compute the proximity embedding by Alg. 2 for each (q, v) and (q, u), then compute the ranking loss $\ell(\pi(q, v), \pi(q, u))$. In lines 15–16, we accumulate the loss for batch b and do gradient descent. Note that in Alg. 2, we overload the function "GetSubpaths" to get different subpaths for symmetric and asymmetric proximities.

Experiments

Data sets. We use three real-world public data sets in our evaluation. The LinkedIn data set (Li, Wang, and Chang

Algorithm 2 GetProxEmbedding

Require: a set of paths $\mathcal{P}$, a query node q, a target node v, embedding dimension d, discount parameter α.
Ensure: proximity embedding $\mathbf{f}(q, v)$.

```
Q(q, v) ← GetSubpaths(P, q, v);
for all path s ∈ Q(q, v) do
    h_s ← LSTM(s) by Eq. 7;
end for
f(q, v) ← DiscountedPathPooling({h_s}) by Eq. 8.
```

Table 1: Data sets with symmetric / asymmetric proximities.

	$\|V\|$	$\|E\|$	$\|C\|$	#(queries)
LinkedIn	65,925	220,812	4	172 (*school.*), 173 (*collea.*)
Facebook	5,025	100,356	10	340 (*family*), 904 (*classmate*)
DBLP	165,728	928,513	5	2,439 (*advisor*), 1,204 (*advisee*)

2014) contains two symmetric semantic classes: *schoolmate* and *colleague*. The Facebook data set (McAuley and Leskovec 2012) contains two symmetric semantic classes: *family* and *classmate*. The DBLP data set (Wang et al. 2010) contains two asymmetric semantic classes: *advisor* and *advisee*. We summarize the data set statistics in Table 1.

Set up. We follow the set up in (Fang et al. 2016b). Specifically, we generate the queries and their ground truth ranking tuples w.r.t. each semantic class. Then we randomly split the queries in each data for each semantic class into two subsets: 20% for training and the rest 80% for testing. In training w.r.t. a particular semantic class, for each q_i, we construct a training tuple (q_i, v_i, u_i) by randomly choosing two other nodes v_i and u_i, such that q_i and v_i belong to the same class, whereas q_i and u_i belong to different classes. In testing w.r.t. the particular semantic class, for each query q'_j, we construct an ideal ranking, such that other nodes that are the same class with q_i are ranked higher than those with a different class or unknown label. For evaluation, we thus predict the ranking for each test query q'_j w.r.t. a semantic class, and compare it against the ideal ranking. We then use NDCG and MAP to evaluate the algorithmic ranking performance for the top 10 nodes in each ranking.

In path sampling, we take node type into consideration to alleviate the possible node type imbalance in the data sets (*e.g.*, in the DBLP data set, there are more "users", but fewer "conferences"). In each step of random walk at a node $v \in V$, instead of randomly sampling among all v's neighbors, we first randomly sample one node type, then randomly sample one neighbor of that type as the next step.

Parameters and environment. In the LinkedIn data set, we set $\gamma = 20, \ell = 20$ for both *schoolmate* and *colleague*. In the Facebook data set, as the graph is smaller than the LinkedIn graph, we try to sample longer length; hence, we set $\gamma = 40, \ell = 80$ for *classmate* and $\gamma = 20, \ell = 80$ for *family*. In the DBLP data set, we set $\gamma = 20, \ell = 80$ for *advisor* and $\gamma = 20, \ell = 40$ for *advisee*. In all the data sets and all the semantic classes, we set by default $\alpha = 0.3$, $\beta = 0.5$ and $\mu = 0.0001$ (except in *family*, $\mu = 0.001$). We tune different d's for different data sets. We run experiments on Linux machines with eight 2.27GHz Intel Xeon(R) CPUs and 32GB memory. We use Theano (Team 2016) for LSTM implementation and Java jdk-1.8 for path sampling.

Table 2: Relative improvement of ProxEmbed over the best baselines when using 1000 labels.

	NDCG	MAP
LinkedIn-*schoolmate*	18.9% ($p < 0.01$)	82.7% ($p < 0.01$)
LinkedIn-*colleague*	12.8% ($p < 0.01$)	51.1% ($p < 0.01$)
Facebook-*family*	1.8% ($p < 0.05$)	5.3% ($p < 0.01$)
Facebook-*classmate*	1.0% ($p < 0.10$)	9.8% ($p < 0.01$)
DBLP-*advisor*	15.6% ($p < 0.01$)	14.5% ($p < 0.01$)
DBLP-*advisee*	9.8% ($p < 0.01$)	4.7% ($p < 0.05$)

Baselines. We compare our ProxEmbed with the following state-of-the-art semantic proximity search baselines.

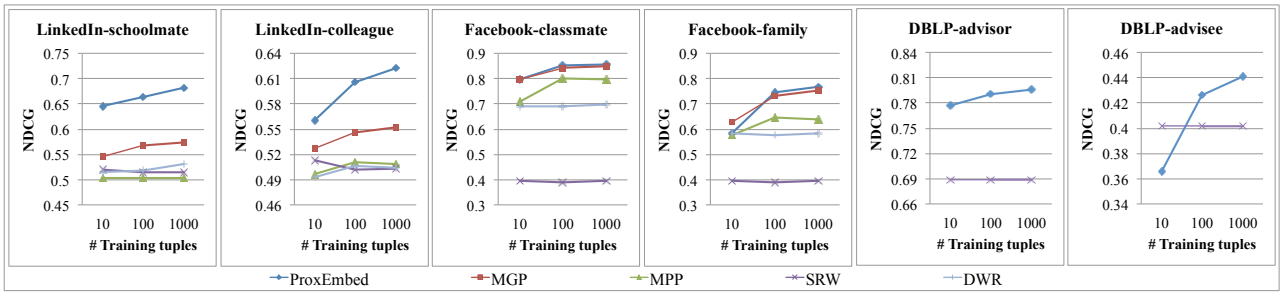
- MGP (Fang et al. 2016b): Meta-Graph Proximity uses meta-graphs as features to measure proximity.
- MPP (Sun et al. 2011): Meta-Path Proximity uses metapaths as features to measure proximity. We follow (Fang et al. 2016b) to generate the metapaths, by restricting the set of metagraphs to paths only.
- SRW (Backstrom and Leskovec 2011): Supervised Random Walk learns the edge weights, so as to make the random walk results consistent with the ground truth ranking. We set edge feature as a 0-or-1 vector indicating the edge type, which is defined by the types of its two nodes.
- DWR: DeepWalk Ranking first learns the node embedding by DeepWalk (Perozzi, Al-Rfou, and Skiena 2014), then it applies Hadamard product on the embeddings of two nodes as the proximity embedding, finally it optimizes a ranking loss like our Eq. 9.

As MGP, MPP and DWR are all designed for symmetric proximity, it is unfair to compare them w.r.t. the asymmetric proximity, thus we exclude them from comparison when the semantic classes are asymmetric. In summary, we compare with all the baselines on the LinkedIn and Facebook data sets, and only compare with SRW on the DBLP data set.

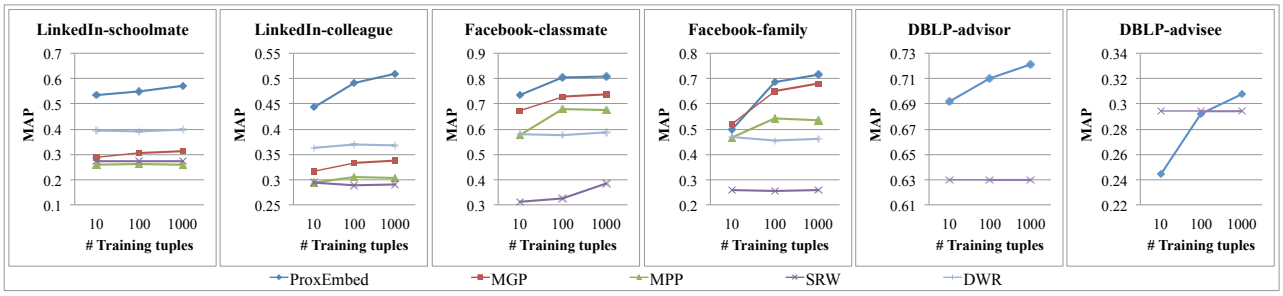
For MGP and MPP, we use the same metapaths and metagraphs as (Fang et al. 2016b); we also set the same parameter values with them. For SRW, we set its regularization parameter $\lambda = 10$, random walk teleportation parameter $\alpha = 0.2$ and loss parameter $b = 0.1$. We set the dimension of DWR as 128, the same as (Perozzi, Al-Rfou, and Skiena 2014). We input the same sampled paths for ProxEmbed to DeepWalk.

Comparison with Baselines. We vary the number of training tuples, and compare ProxEmbed with the baselines on different data sets under different semantic classes. As shown in Fig. 3, ProxEmbed is generally better than the baselines. In Table 2, we summarize our relative improvement over the best baselines under different data sets, different semantic classes and different evaluation metrics. We also report the student t-test one-tailed p-values.

In the following, we analyze the comparison results. Firstly, ProxEmbed is better than MGP and MPP, showing that feature learning is more effective than feature engineering on the graph for proximity learning. As ProxEmbed is



(a) NDCG



(b) MAP

Figure 3: Comparison with baselines for semantic proximity search.

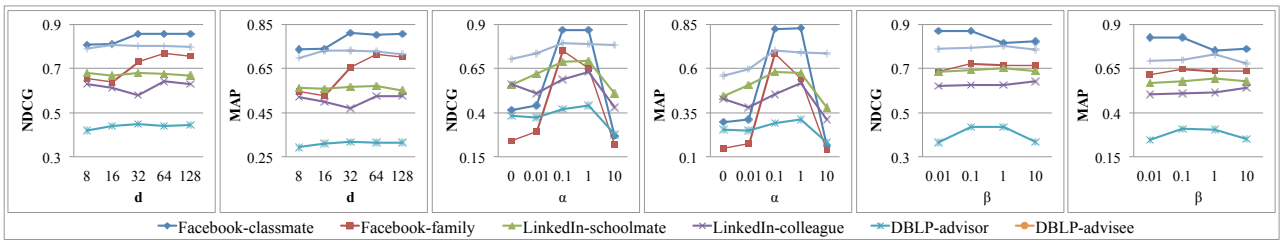


Figure 4: Impact of parameters: number of dimension d, path length discount α, ranking loss discount β.

based on paths, whereas MGP is based on subgraphs, it is interesting to see that ProxEmbed actually outperforms MGP. This suggests our proximity embedding in some sense capture the subgraph information. Secondly, ProxEmbed is better than SRW most of time, except when the number of training tuples is small (*e.g.*, 10) at DBLP's *advisor* and *advisee* semantic classes. This is because ProxEmbed has more parameters to learn than SRW, and it needs more labels. SRW is less sensitive to the number of training tuples, as also observed in (Fang et al. 2016b). Thirdly, ProxEmbed is better than DWR, showing that embedding proximity is more effective than embedding nodes. As motivated in the introduction, embedding nodes is likely to suffer from graph heterogeneity and the gap to turn node embeddings to a proximity score. Besides, DWR is also insensitive to the number of training tuples, as its embedding is unsupervised and only a proximity weight vector with fixed dimensions is learned.

Impact of Parameters. We also study the impact of several parameters that are mostly unique to our model. Here we set the number of training tuples as 100. As we can see, $d = 64$ is mostly the best across different data sets and different semantic classes. $\alpha = 0.1$ is mostly the best, and it is better than $\alpha = 0$ and $\alpha = 10$. This means the path length discount is necessary, but the discount cannot be too big either, otherwise many paths have very little contribution to the proximity embedding. Besides, $\beta = 0.1$ is also mostly the best, suggesting a moderate discount on ranking loss.

Conclusion

In this paper, we study the problem of semantic proximity search on a heterogeneous graph. As most existing methods rely on feature engineering with different graph structures to estimate the proximity, we consider using graph embedding to avoid the feature engineering for proximity learning. We observe that existing node embedding approach is an "indirect" approach to learn the proximity. Thus we introduce a new concept of proximity embedding, and propose ProxEmbed to directly embed the network structure between two nodes. In ProxEmbed, we characterize the proximity between two nodes with a set of paths, then we try to embed and aggregate these paths into a vector as the output. We adopt LSTM and devise an architecture with asymmetric proximity modeling and discounted path pooling to learn the proximity embedding. We further incorporate supervision of

some ground truth proximity ranking tuples to enhance the proximity embedding. Finally, we evaluate ProxEmbed on three real-world data sets with both symmetric and asymmetric proximities. We show that ProxEmbed significantly outperforms all the state-of-the-art baselines.

In future, we plan to extend our method to the weighted graph setting and a deeper neural network architecture.

Acknowledgments

We thank the support from: Zhejiang Provincial Natural Science Foundation (No. LQ14F020002), National Natural Science Foundation of China (No. 61502418 and No. 61602405), Fundamental Research Funds for Central Universities 2016QNA5015 and China Knowledge Centre for Engr. Sci. & Tech., Key Lab of Advanced Info. Sci. & Network Tech. of Beijing (XDXX1603), and the research grant for Human-centered Cyber-physical Systems Programme at Advanced Digital Sciences Center from Singapores Agency for Science, Technology and Research (A*STAR). This work is partially supported by NSF Grant IIS 16-19302. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

Backstrom, L., and Leskovec, J. 2011. Supervised random walks: Predicting and recommending links in social networks. In *WSDM*, 635–644.

Belkin, M., and Niyogi, P. 2001. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *NIPS*, 585–591.

Chang, S.; Han, W.; Tang, J.; Qi, G.; Aggarwal, C. C.; and Huang, T. S. 2015. Heterogeneous network embedding via deep architectures. In *KDD*, 119–128.

Cox, T. F., and Cox, M. 2000. *Multidimensional Scaling, Second Edition*. Chapman and Hall/CRC, 2 edition.

Fang, H.; Wu, F.; Zhao, Z.; Duan, X.; Zhuang, Y.; and Ester, M. 2016a. Community-based question answering via heterogeneous social network learning. In *AAAI*, 122–128.

Fang, Y.; Lin, W.; Zheng, V. W.; Wu, M.; Chang, K. C.; and Li, X. 2016b. Semantic proximity search on graphs with metagraph-based learning. In *ICDE*, 277–288.

Grover, A., and Leskovec, J. 2016. node2vec: Scalable feature learning for networks. In *KDD*.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural Comput.* 9(8):1735–1780.

Hoff, P. 2008. Modeling homophily and stochastic equivalence in symmetric relational data. In *NIPS*. 657–664.

Jeh, G., and Widom, J. 2002. Simrank: A measure of structural-context similarity. In *KDD*.

Jeh, G., and Widom, J. 2003. Scaling personalized web search. In *WWW*, 271–279.

Le, Q. V., and Mikolov, T. 2014. Distributed representations of sentences and documents. In *ICML*, 1188–1196.

Li, R.; Wang, C.; and Chang, K. C. 2014. User profiling in an ego network: co-profiling attributes and relationships. In *WWW*, 819–830.

Luo, Y.; Wang, Q.; Wang, B.; and Guo, L. 2015. Context-dependent knowledge graph embedding. In *EMNLP*, 1656–1661.

McAuley, J. J., and Leskovec, J. 2012. Learning to discover social circles in ego networks. In *NIPS*, 548–556.

Niepert, M.; Ahmed, M.; and Kutzkov, K. 2016. Learning convolutional neural networks for graphs. In *ICML*, 2014–2023.

Ou, M.; Cui, P.; Pei, J.; Zhang, Z.; and Zhu, W. 2016. Asymmetric transitivity preserving graph embedding. In *KDD*, 1105–1114.

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

Perrault-Joncas, D. C., and Meila, M. 2011. Directed graph embedding: an algorithm based on continuous limits of laplacian-type operators. In *NIPS*, 990–998.

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

Sun, Y.; Han, J.; Yan, X.; Yu, P. S.; and Wu, T. 2011. PathSim: Meta path-based top-k similarity search in heterogeneous information networks. *PVLDB* 4(11).

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *NIPS*, 3104–3112.

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

Team, T. D. 2016. Theano: A Python framework for fast computation of mathematical expressions. *CoRR* abs/1605.02688.

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

Wang, C.; Han, J.; Jia, Y.; Tang, J.; Zhang, D.; Yu, Y.; and Guo, J. 2010. Mining advisor-advisee relationships from research publication networks. In *KDD*, 203–212.

Wang, D.; Cui, P.; and Zhu, W. 2016. Structural deep network embedding. In *KDD*, 1225–1234.

Xie, R.; Liu, Z.; Jia, J.; Luan, H.; and Sun, M. 2016. Representation learning of knowledge graphs with entity descriptions. In *AAAI*, 2659–2665.

Yang, L.; Cao, X.; He, D.; Wang, C.; Wang, X.; and Zhang, W. 2016. Modularity based community detection with deep learning. In *IJCAI*, 2252–2258.

Yang, Z.; Cohen, W. W.; and Salakhutdinov, R. 2016. Revisiting semi-supervised learning with graph embeddings. In *ICML*, 40–48.

Zheng, V. W.; Cavallari, S.; Cai, H.; Chang, K. C.; and Cambria, E. 2016. From node embedding to community embedding. *CoRR* abs/1610.09950.