Dynamic Database Embeddings with FoRWaRD

Jan Toenshoff
RWTH
Aachen, Germany
toenshoff@informatik.rwth-aachen.de

Neta Friedman
Technion
Haifa, Israel
netafr@cs.technion.ac.il

Martin Grohe
RWTH
Aachen, Germany
grohe@informatik.rwth-aachen.de

Benny Kimelfeld
Technion
Haifa, Israel
bennyk@cs.technion.ac.il

ABSTRACT

We study the problem of computing an embedding of the tuples of a relational database in a manner that is extensible to dynamic changes of the database. Importantly, the embedding of existing tuples should not change due to the embedding of newly inserted tuples (as database applications might rely on existing embeddings), while the embedding of all tuples, old and new, should retain high quality. This task is challenging since state-of-the-art embedding techniques for structured data, such as (adaptations of) embeddings on graphs, have inherent inter-dependencies among the embeddings of different entities. We present the FoRWaRD algorithm (Foreign Key Random Walk Embeddings for Relational Databases) that draws from embedding techniques for general graphs and knowledge graphs, and is inherently utilizing the schema and its key and foreign-key constraints. We compare FoRWaRD to an alternative approach that we devise by adapting node embeddings for graphs (Node2Vec) to dynamic databases. We show that FoRWaRD is comparable and sometimes superior to state-of-the-art embeddings in the static (traditional) setting, using a collection of downstream tasks of column prediction over geographical and biological domains. More importantly, in the dynamic setting FoRWaRD outperforms the alternatives consistently and often considerably, and features only a mild reduction of quality even when the database consists of mostly newly inserted tuples.

1 INTRODUCTION

Standard machine learning algorithms assume representations of their input data as numerical vectors. Applying these algorithms for the analysis of non-numerical data requires *embeddings* of these data into a (typically) finite dimensional Euclidean vector space. The embedding needs to be "faithful" to the semantics. In particular, similar entities should be mapped to vectors that are close geometrically, and vice versa. In some modalities, the input comes with a useful embedding to begin with; for example, an image can be represented by the RGB intensities of its pixels. In others, semantic-aware embeddings have to be devised, and indeed have been devised, such as Word2Vec [28] and GLoVe [34] for words of natural language [24], Node2Vec for the nodes of a graph [15], TransE [9] for the entities of a knowledge graph, and Mol2Vec [20] for molecule structures. A possible approach to evaluating the *quality* of a generic embedding technique is via different downstream tasks: solve a collection of machine-learning tasks by utilizing machine-learning models that operate over the embedding (see, e.g., [23, 25]), as illustrated in Figure 1.

Generalizing graph embeddings, generic embeddings have also been devised for relational databases. Such embeddings have enabled the deployment of machine-learning architectures to traditional database tasks such as record similarity [6–8, 16, 17], record linking [13, 29] and other integration tasks such as schema, token and record matching (entity resolution) [10]. The embedded entities are typically either tuples or attribute values. In this work, we focus on tuple embeddings.

Various approaches have been proposed for obtaining embeddings in databases. One is to concatenate predefined embeddings of the attribute values (which are, e.g., words or quantities) [13, 29] or permutations over the list of these values [10]. Another approach views the tuples as text documents and applies word and document embedding [7]. More recently, Cappuzzo, Papotti and Thirumuruganathan [10] studied the approach of transforming the database into a graph (through a variation of the Gaifman graph) and applying a node embedding over this graph. Arguably, an important advantage of the graph approach is that it utilizes information that should be highly relevant to the semantics of the data and is freely available in databases: the *structure* of the data. (See [14] for a recent essay on embeddings for structured data.) For instance, we can infer considerable information about a tuple with apparently meaningless values, such as internal codes, by looking at the tuples that are referenced by this tuple. We are not aware of any embedding technique that directly uses the most common way of referencing tuples, namely foreign-key (to-key) references. Yet, it has been shown useful in the general context of machine learning [37, 39]. As we explain later on, in this work we show how these can be gracefully utilized for the sake of high-quality tuple embedding.

Incorporating the database structure, and particularly references among tuples, means that the embeddings of tuples depend upon each other. This leads to new challenges since the database is often not a static object but rather serves a dynamic organization (e.g., with arrival of new customers and purchases, new patients and patient records, etc.). When new data arrives, we are in a situation where we have an embedding for the old tuples but not the new ones. A straightforward solution to this problem is to reapply the embedding algorithm from scratch over the new database. This approach, however, suffers from two main drawbacks. First, it might be computationally too expensive to compute the embedding over the entire database upon every tuple arrival. More fundamentally,

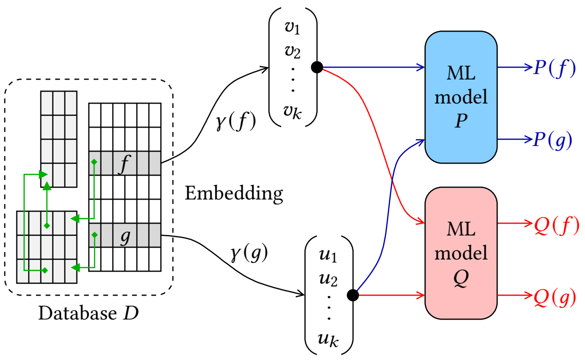


Figure 1: Embedding tuples in a database. Different machine-learning models for various downstream tasks operate over the same vectors that the tuples are mapped to by the embedding. The embedding algorithm can utilize the database structure, particularly the foreign-key references.

reapplying the embedding algorithm is likely to change the embedding of the old tuples due to the inherent randomness in most embedding algorithms [38]. Consequently, it may have serious implications on accompanying database endpoints that have already been using the old embeddings (e.g., for various classification and prediction tasks). Hence, we address the problem of inferring embeddings of new tuples without changing the embedding of old ones. Of course, the challenge is to do so while retaining high quality of the embedding. We attack this problem by designing a database embedding that is inherently extensible to new tuples.

To be more precise, we study the following task that we refer to as *dynamic database embedding*. We need to devise two algorithms. The first algorithm, applied in the *static phase*, takes as input a database D over a schema σ and learns a tuple embedding γ that maps every fact of D (i.e., occurrence of a tuple in a certain relation in D) to the vector space $\mathbb{R}^k$ (for some hyperparameter k). Note that this phase solves the task of *static database embedding*, that is, the embedding problem in its traditional sense. The second algorithm, applied in the *dynamic phase*, has access to D and the tuple embedding γ and takes as input a newly arrived tuple t that is not in D; the goal of this algorithm is to extend γ to $D \cup \{t\}$ by determining the value $\gamma(f)$. Importantly, the schema σ specifies the key and foreign-key constraints that can be used for understanding the actual foreign-key references that exist inside the database D and its future evolution.

Since, for reasons explained above, we want to keep the embedding of the existing tuples stable, deleting tuple from a database is not an issue, we simply delete the tuples and their images under the embedding. This is why we focus on adding tuples in our experiments. Of course, eventually there may be a point where the database has changed so much that a completely new embedding has to be computed.

Our Contributions. The initial solution that we design is based on our adaption of Node2Vec to relational database embeddings. We found empirically that this adaptation performs very well for the static embedding problem. For dynamic database embedding, we devise a dynamic version of Node2Vec that is based on the following idea. When we extend an existing embedding to new nodes, we continue the training of Node2Vec from where it stopped while performing gradient descent *only on the embeddings of new nodes.*

Going forward, the primary solution we devise is a new algorithm, FoRWaRD (Foreign Key Random Walk Embeddings for Relational Databases), that is inherently built to accommodate the structure of relational databases and, importantly, to be extensible to dynamic databases. We demonstrate experimentally that it performs very well for the dynamic database embedding problem, which it was designed for, as well as the static database embedding problem.

The algorithm FoRWaRD draws from node embedding techniques based on random walks [15, 35] as well as the knowledge graph embedding algorithm Rescal [31]. From each tuple in the database we start random walks in the database by repeatedly following foreign-key references. We embed tuples of the database depending on the similarity of the distributions of these walks, where we measure similarity in terms of a predefined similarity measures on the attribute values. Crucially, together with the embedding of tuples, for each type, or *scheme*, of walk we learn a separate similarity measure, encoded as a matrix describing an inner product. Once we have computed, or rather learned, an initial static embedding, we can dynamically extend it to each new tuple essentially by solving a system of linear equations that constrain the distances to the existing tuples (or a random subset of these).

We conducted experiments over several downstream tasks of (binary and multi-label) column prediction from multiple benchmark databases over geographical and biological domains [12, 26, 30, 40]. We showed that FoRWaRD performs very well even for the static database embedding problem. On the majority of the benchmarks, both FoRWaRD and our static adaptation of Node2Vec clearly outperform the state-of-the-art baselines (see Table 3).

Let us now turn to the dynamic embedding problem. Both FoRWaRD and dynamic Node2Vec perform well for the dynamic embedding problem, though overall FoRWaRD shows a clearly superior performance (see Figure 5 and Table 4). Quite remarkably, the performance of FoRWaRD turns out to be rather stable as we add more and more tuples to the database. Even with 50%, sometimes even 80%, newly added tuples the drop in accuracy for column prediction is small (see Figure 5). Combined with the strong performance of FoRWaRD for the static embedding problem, this convincingly shows that FoRWaRD is a viable solution to the dynamic embedding problem.

In summary, our contributions are as follows. First, we define the problem of dynamic database embedding. Second, we devise an adaptation of Node2Vec to dynamic databases. Third (and primarily), we devise the model FoRWaRD algorithm in both its static and dynamic versions. Fourth, we conduct a thorough experimental evaluation of our algorithms, in comparison to other baselines, over a collection of tuple prediction benchmarks, and empirically establish the superiority of FoRWaRD.

Organization. The remainder of the paper is organized as follows. After preliminary definitions and notations in Section 2, we define the problem of dynamic database embedding in Section 3. In Section 4 we describe the FoRWaRD algorithm, and in Section 5 the dynamic Node2Vec solution. Finally, we present our experimental evaluation in Section 6 and conclude in Section 7.

2 PRELIMINARIES

We focus on databases over schemas with key and foreign-key constraints. More precisely, a *database schema* σ consists of a finite collection of *relation schemas* $R(A_1, \dots, A_k)$ where R is a distinct *relation name* and each A_i is a distinct *attribute name*. If A is an attribute of R, then we refer to the term $R.A$ as an *attribute* (*of* σ). Each attribute $R.A$ is associated with a *domain*, denoted $\mathrm{dom}(R.A)$. Each relation schema $R(A_1, \dots, A_k)$ has a unique *key*, denoted $\mathrm{key}(R)$, such that $\mathrm{key}(R) \subseteq \{A_1, \dots, A_k\}$. Note that $\mathrm{key}(R)$ can be empty. By a *foreign-key constraint* (FK) we refer to an inclusion dependency of the form $R[\boldsymbol{B}] \subseteq S[\boldsymbol{C}]$ where R and S are relation names, $\boldsymbol{B} = B_1, \dots, B_\ell$ and $\boldsymbol{C} = C_1, \dots, C_\ell$ are sequences of distinct attributes of R and S, respectively, and $\mathrm{key}(S) = \{C_1, \dots, C_\ell\}$.

A *database* D over the schema σ is a finite set of *facts* $R(a_1, \dots, a_k)$ over the relation schemas $R(A_1, \dots, A_k)$ of σ, so that $a_i \in \mathrm{dom}(R.A_i)$ for all $i = 1, \dots, k$. In addition, such an a_i can be missing, in which case we assume that it is a distinguished *null* value (that belongs to none of the attributes domains) that we denote by $\perp$. The fact $R(a_1, \dots, a_k)$ is also called an *R-fact* and a *σ-fact*. We denote by $R(D)$ the restriction of D to its R-facts. For a fact $f = R(a_1, \dots, a_k)$ over $R(A_1, \dots, A_k)$, we denote by $f[A_i]$ the value a_i, and by $f[B_1, \dots, B_\ell]$ the series $f[B_1], \dots, f[B_\ell]$. For a database D, the *active domain* of an attribute $R.A$ (*w.r.t. to* D), denoted $\mathrm{adom}_D(R.A)$ or just $\mathrm{adom}(R.A)$ if D is clear from the context, is the set of values that occur in D for the attribute $R.A$, that is, $\mathrm{adom}(R.A) = \{f[A] \mid f \in R(D)\}$.

We require that the database D over σ satisfies the constraints of σ. In particular, for the key constraints we require every two distinct R-facts f_1 and f_2 must satisfy $f_1[C] \neq f_2[C]$ for at least one attribute $C \in \mathrm{key}(R)$, and both $f_1[C]$ and $f_2[C]$ are nonnull. Moreover, for every FK φ of the form $R[\boldsymbol{B}] \subseteq S[\boldsymbol{C}]$ and R-fact $f \in D$, if $f[\boldsymbol{B}]$ has no nulls[1] then there exists an S-fact $g \in D$ such that $f[\boldsymbol{B}] = g[\boldsymbol{C}]$; in this case, we say that f *references* g *via* φ (note that f references precisely one fact via φ).

Example 2.1. Figure 2 depicts an example of a movie database over a schema. The leftmost column of each relation includes tuple names for later reference, and is not considered part of the database itself. As conventional, keys are marked by underlying the key attributes. The FKs of each relation schema are given under the corresponding relation. For example, The Movies relation has the key constraint key(Movies) = {mid} and the FK Movies[studio] ⊆ Studio[sid]. The reader can verify that the FK is indeed satisfied by the database of the figure; for example, s03 is indeed the sid attribute of a fact of Studio, namely s_3. Finally, observe that the genre attribute of m_3 is missing, that is, $m_3[\mathrm{genre}] = \perp$. ◆

3 PROBLEM DEFINITION

As explained in the Introduction, the computational problem that we study in this paper is twofold.

- *Static database embedding*: The goal is to derive an embedding of the tuples in the traditional sense. Formally, we are given an input a database D over a schema σ, and we wish to compute an embedding function $\gamma : D \to \mathbb{R}^k$ for some hyperparameter $k > 0$.
- *Dynamic database embedding*: Here the goal is to extend the embedding γ to a new fact f. Formally, we are given an input database D over a schema σ, a precomputed embedding $\gamma : D \to \mathbb{R}^k$ and a new fact $f \notin D$. Our goal is to compute a new embedding $\gamma' : D \cup \{f\} \to \mathbb{R}^k$ such that $\gamma'(f') = \gamma(f')$ for all $f' \in D$. Hence, we only need to compute the result of the embedding on f.

We generalize the problem of dynamic database embedding to a sequence $f_1, \dots, f_\ell$ of new facts arrive (rather than just a single fact f) in the obvious manner. That is, we need to solve ℓ instances of the problem, where each time γ' becomes the γ of the next instance. Hence, in the ith instance we have $\gamma : D \cup \{f_1, \dots, f_{i-1}\} \to \mathbb{R}^k$.

Therefore, we need to devise two algorithms: one for the static database embedding and one for the dynamic database embedding. Note that these algorithms might (and actually should) depend on each other: the algorithm for the static phase needs not only to perform well, but also allow and enhance the effectiveness of the algorithm for the dynamic phase.

Example 3.1. Let D' be the database of Figure 2 and $D = D' \setminus c_4$. Hence, D' is obtained from D by inserting the fact

$$c_4 = \textsc{Collaborations}(\texttt{a01}, \texttt{a04}, \texttt{m06})\,.$$

In static database embedding, for the input D we need to compute a mapping $g\gamma : D \to \mathbb{R}^k$. In the dynamic database embedding, when inserting c_4 into D we wish to extend γ to c_4 by computing $\gamma(c_4)$ without changing it on the facts of D; for example, $\gamma(c_1)$ and $\gamma(m_1)$ should remain intact. To determine $\gamma(c_4)$ we can utilize the semantic knowledge that the new c_4 references the existing a_1, a_4 and m_6. ◆

When we empirically evaluate the quality of γ, we typically consider a downstream learning task (e.g., record classification) and some underlying learning algorithm for the task (e.g., an artificial neural network), where every fact f is represented by its image $\gamma(f)$. See Figure 1 for illustration. We then evaluate the embedding by evaluating the effectiveness of solving the downstream task. Moreover, we evaluate the effectiveness not only over D, but also on databases D' that result from inserting into D new facts.

4 THE FORWARD ALGORITHM

We now present our method for learning an embedding for a schema σ from a database D. Throughout this section we assume underlying σ and D and refrain from mentioning them explicitly.

[1]Note that we adopt the convention that an FK is ignored in a fact that includes nulls in one or more of the referencing attributes.

Movies

	<u>mid</u>	studio	title	genre	budget
m_1	m01	s03	Titanic	Drama	200M
m_2	m02	s01	Inception	SciFi	160M
m_3	m03	s01	Godzilla	⊥	150M
m_4	m04	s03	Interstellar	SciFi	160M
m_5	m05	s02	Tropic Thunder	Action	90M
m_6	m06	s01	Wolf of Wall St.	Bio	100M

Movies[studio] ⊆ Studios[sid]

Studios

	<u>sid</u>	name	loc
s_1	s01	Warner Bros.	LA
s_2	s02	Universal	LA
s_3	s03	Paramount	LA

Actors

	<u>aid</u>	name	worth
a_1	a01	DiCaprio	230M
a_2	a02	Watanabe	40M
a_3	a03	Cruise	600M
a_4	a04	McConaughey	140M
a_5	a05	Damon	170M

Collaborations

	<u>actor1</u>	<u>actor2</u>	<u>movie</u>
c_1	a01	a02	m03
c_2	a04	a05	m04
c_3	a04	a03	m05
c_4	a01	a04	m06

Collaborations[actor1] ⊆ Actors[aid]
Collaborations[actor2] ⊆ Actors[aid]
Collaborations[movie] ⊆ Movies[mid]

Figure 2: Database example. Key constraints are marked with underline and foreign-key references are specified under the corresponding relations.

4.1 Random Walks over Database Facts

We consider random walks over database facts, where the transition from one fact to another follows a pattern (or scheme) of FK (forward or backward) references. Formally, a *walk scheme* is a sequence s of the form

$$R_0[\boldsymbol{A}^0] - R_1[\boldsymbol{B}^1], R_1[\boldsymbol{A}^1] - R_2[\boldsymbol{B}^2], \dots, R_{\ell-1}[\boldsymbol{A}^{\ell-1}] - R_\ell[\boldsymbol{B}^\ell] \quad (1)$$

such that for all $k = 1, \dots, \ell$, either $R_{k-1}[\boldsymbol{A}^{k-1}] \subseteq R_k[\boldsymbol{B}^k]$ is an FK or $R_k[\boldsymbol{B}^k] \subseteq R_{k-1}[\boldsymbol{A}^{k-1}]$ is an FK. We say that s has *length* ℓ, *starts from* R_0 and *ends with* R_ℓ.

Example 4.1. Figure 3 depicts nine walk schemes, $s_1, \dots, s_9$ that start from the Actor relation. The reader can verify that these are all of the walk schemes of length at most three that start from Actor. For illustration, let us consider the scheme s_5 from the figure. In our notation, this walk scheme is written as

$$\text{Actors}[\text{aid}] - \text{Collaborations}[\text{actor2}],$$
$$\text{Collaborations}[\text{movie}] - \text{Movies}[\text{mid}]\,.$$

Note that s_1 ends with Collaborations while s_5 ends with with Movies. ◆

A *walk* with the scheme s is a sequence $(f_0, \dots, f_\ell)$ of facts such that f_k is an R_k-fact and $f_{k-1}[\boldsymbol{A}^{k-1}] = f_k[\boldsymbol{B}^k]$ for all $k = 1, \dots, \ell$. We say that $(f_0, \dots, f_\ell)$ *starts from*, or has the *source*, f_0, and that it *ends with*, or has the *destination*, f_ℓ.

Example 4.2. Continuing our example, consider again the walk scheme s_5 of Figure 3. Starting at the fact a_1 of Figure 2, there are two walks that follow the scheme s_5, namely (a_1, c_1, m_3) and (a_1, c_4, m_6). ◆

Note that we allow walk schemes and walks of length zero. For each relation R there is a scheme of length zero that starts and ends in R. The walks of this scheme have the form (f_0) and simply end directly at the start fact f_0 in R.

Let s be a walk scheme as written in (1). By a *random walk* with the scheme s we refer to the walk obtained by uniformly selecting the next valid fact in the walk. More formally, let $f_0 = f$ be an R_0-fact. We denote by $\mathcal{W}(f, s)$ the distribution over the walks with the schema s where each walk is sampled by starting from f_0 and then iteratively selecting f_k, for $k = 1, \dots, \ell$, randomly and uniformly from the set $\{f \in R_k \mid f[\boldsymbol{B}^k] = f_{k-1}[\boldsymbol{A}^{k-1}]\}$. We denote by $d_{f,s}$ the random variable/element that maps each walk in $\mathcal{W}(f, s)$ to its destination, that is, the last fact in the walk. Then for a fact $g \in R^k(D)$, the probability that a walk sampled from $\mathcal{W}(f, s)$ ends with g is $\Pr(d_{f,s} = g)$. Observe that for every attribute A of R_k we get the random variable $d_{f,s}[A]$ that forms the value of the random walk's destination in the attribute A. Given a start fact f_0 in the start relation R_0 of walk scheme s, one can compute the distribution $d_{f,s}$ through a simple breadth first search along the sequence of foreign keys specified by s.

Example 4.3. Recall from Example 4.2 that the walks (a_1, c_1, m_3) and (a_1, c_4, m_6) are the only two walks that follow the scheme s_5 (Figure 3) and start from a_1 (Figure 2). Therefore, for the random variable d_{a_1,s_5} it holds that $\Pr(d_{a_1,s_5} = m_3) = 0.5$ and $\Pr(d_{a_1,s_5} = m_6) = 0.5$. Moreover, we have the following.

$$\Pr(d_{a_1,s_5}[\text{budget}] = 150\text{M}) = 0.5$$
$$\Pr(d_{a_1,s_5}[\text{budget}] = 100\text{M}) = 0.5$$
$$\Pr(d_{a_1,s_5}[\text{genre}] = \text{Bio}) = 1.0$$

Hence, each of $d_{a_1,s_5}[\text{budget}]$ and $d_{a_1,s_5}[\text{genre}]$ indeed defines a distribution over attribute values. ◆

Recall that databases (in real life and in our formal framework) may have missing values. A random walk starting at f might end at a fact R_ℓ-fact g which has no known value for an attribute A of R_ℓ. Therefore, the random variable $d_{f,s}[A]$ can in theory assume the value $\perp \notin dom(R_\ell.A)$. As a convention, we define the probability distribution of $d_{f,s}[A]$ as the posterior distribution after $d_{f,s}[A] \neq \perp$ is known. With this modification we enforce $d_{f,s}[A] \in dom(R_\ell.A)$. This will be crucial in Section 4.2, where we define similarity measures for $d_{f,s}[A]$ based on $dom(R_\ell.A)$. If all walks from f with scheme s end at facts g with $g[A] = \perp$, then $d_{f,s}[A]$ does not exist and is not considered by FoRWaRD. This also includes the case where no walks with scheme s exist from start f.

4.2 Kernelized Domains

For every attribute $R.A$ we assume an embedding into a Hilbert space: $\alpha_{R.A} : \mathrm{dom}(R.A) \to \mathcal{H}_{R.A}$. The embedding $\alpha_{R.A}$ can be the identity function (i.e., a value is embedded to into itself) in the case of a categorical domain, a word/sentence embedding in the case of a natural-language domain [24], a molecule embedding in the case of a domain of molecules [20], and so on.

Example 4.4. In the database of Figure 2 we can use the embedding $\alpha_{\text{Movies.budget}}(x) = x$ for the budget attribute of Movies, a word/document embedding as $\alpha_{\text{Movies.title}}$ for the title attribute, and a categorical one-hot encoding

$$\alpha_{\text{Movies.genre}}(x) = (0, \dots, 0, 1, 0, \dots, 0)$$

for the genre attribute. ◆

Using $\alpha_{R.A}$, we define the inner-product kernel function $\kappa_{R.A}$:

$$\kappa_{R.A}(v, v') =: \langle \alpha_{R.A}(v), \alpha_{R.A}(v') \rangle_{\mathcal{H}_{R.A}}$$

We assume that all domain embeddings are normalized such that $\kappa_{R.A}(v, v) = 1$ for all values $v \in \mathrm{dom}(R.A)$. FoRWaRD does not directly compute $\alpha_{R.A}$ for any values, but only evaluates $\kappa_{R.A}$ on value pairs. It is therefore sufficient to implement each kernel function without explicitly computing the underlying embedding. For example, one may use the Gaussian distance as a kernel for numerical domains:

$$\kappa(v, v') = \exp\left(-\frac{(v - v')^2}{2\sigma^2}\right),$$

where $\sigma^2 > 0$ is a predetermined variance. The embedding associated with this kernel does not have to be computed to obtain the value of the kernel function. This also applies to the identity kernel, which can be implemented directly with string comparison as $\kappa(v, v') = \mathbb{1}_{v=v'}$ rather than projecting each value to a unique vector. This is our default choice for categorical domains, where all distinct values are assumed to have a similarity of zero. The domains of foreign key constraints are typically unique IDs, which we also consider to be categorical data. We also associate these domains with the categorical kernel. However, this is just a formality since FoRWaRD only evaluates the kernel on domains that are not involved in foreign keys.

Kernel functions offer a straightforward way of encoding domain knowledge by modeling the similarity of the domain's values. Kernels are also helpful when dealing with noisy data. For example, on text domains kernels based on the edit distance can be used to smooth out random typos. In the following we use these kernel functions to define similarity measures for the random variables $d_{f,s}[A]$.

Let s be a walk scheme of length ℓ from relation R_1 to R_ℓ. Let A be an attribute of R_ℓ and let f and f' be two distinct R_1-facts. Under our assumption, $d_{f,s}[A]$ and $d_{f,s'}[A]$ are random variables over a shared kernelized domain $dom(R_\ell.A)$. We utilize this to quantify the similarity between $d_{f,s}[A]$ and $d_{f',s}[A]$ with respect to the underlying kernel $\kappa_{R_\ell.A}$. To this end, we define two different similarity measures:

The *Expected Kernel Distance* $K_{\mathbf{EK}}$ is the expected distance between two random values selected independently at random.

$$K_{\mathbf{EK}}(d_{s,f}[A], d_{s,f'}[A]) = \mathop{\mathbb{E}}_{\mathcal{W}(f,s)\times\mathcal{W}(f',s)}[\kappa(d_{s,f}[A], d_{s,f'}[A])] \quad (2)$$

Secondly, we define a measure based on the *Maximum Mean Discrepancy* (MMD):

$$K_{\mathbf{MMD}}(d_{f,s}[A], d_{f',s}[A]) =: 1 - \frac{1}{2}\mathrm{MMD}(d_{f,s}[A], d_{f',s}[A]) \quad (3)$$

Here, the *Maximum Mean Discrepancy* $\mathrm{MMD}(x, x')$ quantifies the difference of two random variables x and x' over domain Δ with kernel κ through:

$$\begin{aligned}\mathrm{MMD}(x, x') = \left\|\mathbb{E}[x] - \mathbb{E}[x']\right\|^2_{\mathcal{H}} = \\ \sum_{v\in\Delta}\sum_{v'\in\Delta} \Pr(x = v)\cdot\Pr(x = v')\cdot\kappa(v, v') \\ + \sum_{v\in D}\sum_{v'\in D} \Pr(x' = v)\cdot\Pr(x' = v')\cdot\kappa(v, v') \\ -2\sum_{v\in D}\sum_{v'\in D} \Pr(x = v)\cdot\Pr(x' = v')\cdot\kappa(v, v')\end{aligned}$$

4.3 Embedding

Let D be a relational database and let R be the relation for which we want to compute a vector embedding $\varphi : R(D) \mapsto \mathbb{R}^d$ of dimension $d \in \mathbb{N}$. Intuitively, FoRWaRD embeddings aim to model the similarity of the random walk destinations $d_{s,f}$ for all R-facts f and all walk schemes $s \in \mathcal{T}(R, \ell_{\max})$ up to a certain walk length $\ell_{\max}$ with one vector representation φ. Formally, we define $\mathcal{T}_{\text{attr}}(R, \ell_{\max})$ as the set of all tuples (s, A) such that $s \in \mathcal{T}(R, \ell_{\max})$ and A is an attribute of the destination R_ℓ of s that is not involved in any foreign-key constraints. We train an additional matrix embedding $\psi : \mathcal{T}_{\text{attr}}(R, \ell_{\max}) \to \mathbb{R}^{d\times d}$ that maps each pair (s, A) to a symmetric matrix $\psi(s, A)$. We optimize φ and ψ to satisfy

$$\varphi(f)^\top \psi(s, A)\varphi(f') \overset{!}{=} K(d_{s,f}[A], d_{s,f'}[A]). \quad (4)$$

Here, K is any similarity measure for probability distributions over kernelized domains, such as $K_{\mathbf{EK}}$ or $K_{\mathbf{MMD}}$. $\psi(s, A)$ modulates the inner product between $\varphi(f)$ and $\varphi(f')$ in Equation (4). This allows us to encode multiple similarity measures (one for each (s, A)) into one vector embedding for tuples. After training, the embedding φ is the primary output of FoRWaRD.

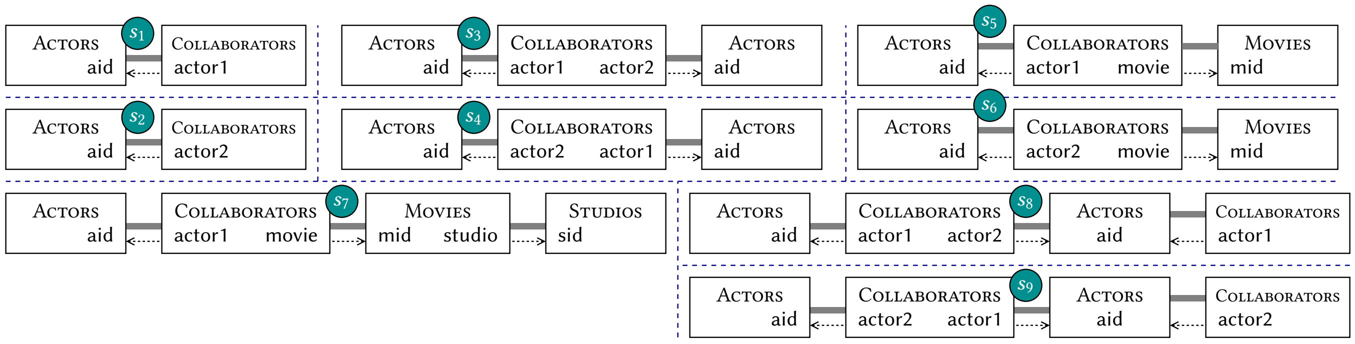


Figure 3: All walk schemes of length at most three, according to the database schema of Figure 2, that start from the Actor relation.

4.4 Optimization

We utilize gradient descent to optimize FoRWaRD embeddings. That is, the embeddings φ and ψ are initialized randomly and then stochastic gradient descend is used to minimize the ℓ_2-loss of the objective.

We sample a large number of examples (f, f', s, A) where f and f' are R facts from the database and $(s, A) \in \mathcal{T}_{\text{attr}}(R, \ell_{\max})$. To this end, we specify a hyperparameter $n_{\text{samples}} \in \mathbb{N}$. For each R-fact f and each $(s, A) \in \mathcal{T}_{\text{attr}}(R, \ell_{\max})$ for which $d_{s,f}[A]$ exists we uniformly sample n_{samples} distinct R facts f' with $f' \neq f$ and add (f, f', s, A) to the training examples. If the total number of such samples is less then n_{samples}, then we just use all samples without duplicates. During gradient descent we aim to minimize

$$\mathcal{L} = \frac{1}{2}|\varphi(f)^\top \psi(s, A)\varphi(f') - K(d_{s,f}[A], d_{s,f'}[A])|^2$$

for every sample. Due to the large number of samples, we use stochastic gradient descent where the samples are divided into batches of a fixed batch size.

Note that this requires us to compute the distributions $d_{s,f}[A]$ and $d_{s,f'}[A]$ as well as their kernel value explicitly, which may be prohibitively expensive for large databases. When using the Expected Kernel Distance $K_{\mathbf{EK}}$, we deviate from this procedure for the sake of efficiency. We sample tuples of the form (f, f', s, A, g, g') with R_ℓ facts g and g', which are the destinations of random walks sampled for f and f', respectively. We then use $\kappa_{R_\ell.A}(g[A], g'[A])$ as a (stochastic) estimate of $K_{EK}(d_{s,f}[A], d_{s,f'}[A])$ and minimize

$$\mathcal{L} = \frac{1}{2}|\varphi(f)^\top \psi(s, A)\varphi(f') - \kappa_{R_\ell.A}(g[A], g'[A])|^2.$$

With this modification, we do not have to compute $d_{s,f}[A]$ and $d_{s,f'}[A]$) explicitly.

4.5 Extending Embeddings to New Tuples

We consider the situation where a new R-fact f_{new} is inserted into the database D, and our goal is to extend the existing embedding φ over D to incorporate f_{new}. Hence, our goal is to determine the vector $\varphi(f_{\text{new}}) \in \mathbb{R}^d$.

Let f_{old} be an R-fact such that the embedding $\varphi(f_{\text{old}}) \in \mathbb{R}^d$ is already known and let $(s, A) \in \mathcal{T}_{\text{attr}}(R, \ell_{\max})$. Then $\psi(s, A) \in \mathbb{R}^{d \times d}$ is a known matrix (that we referred to in Equation (4)). We wish for our new embedding $\varphi(f_{\text{new}})$ to satisfy the objective in Equation (4) with respect to $f = f_{\text{new}}$ and $f' = f_{\text{old}}$:

$$\varphi(f_{\text{new}})^\top \cdot \psi(s, A) \cdot \varphi(f_{\text{old}}) \overset{!}{=} K(d_{s,f_{\text{old}}}[A], d_{s,f_{\text{new}}}[A])$$

Hence, we obtain a linear equation $\varphi(f_{\text{new}})^\top \cdot c = y$ where $c \in \mathbb{R}^d$ and $y \in \mathbb{R}$ are known. If we stack these linear equations for many choices of (f', s, A), then we obtain an overdetermined system of linear equations that we can (approximately) solve for $\varphi(f_{\text{new}})$.

We randomly sample a sufficiently large number of such triples. In particular, we sample $n^{\text{new}}_{\text{samples}} \in \mathbb{N}$ distinct samples for each $(s, A) \in \mathcal{T}_{\text{attr}}(R, \ell_{\max})$ where $n^{\text{new}}_{\text{samples}}$ is a hyperparameter. Let k be the total number of drawn samples and let (f_i, s_i, A_i) be the i-th sample with $i \in [k]$. We define the matrix $C \in \mathbb{R}^{k \times d}$ such that

$$C_i = \psi(s_i, A_i) \cdot \varphi(f_i).$$

Define $b \in \mathbb{R}^k$ to be the vector with

$$b_i = K(d_{s_i,f_i}[A_i], d_{s_i,f_{\text{new}}}[A_i]).$$

To obtain a new embedding for f_{new} we solve for

$$C \cdot \varphi(f_{\text{new}}) = b.$$

Thus, we can induce embeddings for novel data by solving overdetermined linear systems. Note that we aim to find an approximate solution, since there exists no exact solution for overdetermined linear systems in general. Any standard method for solving such systems can be applied. In our case, we use the pseudoinverse C^+ of C to obtain a solution solution that is optimal in the Euclidean norm:

$$\varphi(f_{\text{new}}) = C^+ \cdot b$$

4.6 Hyperparameters

The main hyperparameters are the embedding size d, the maximum walk length $\ell_{\max}$, the similarity measure K (either $K_{\mathbf{EK}}$ or $K_{\mathbf{MMD}}$) and the number of samples n_{samples}, as described in Section 4.4. The batch size, learning rate and number of epochs of the gradient descent training are additional parameters. The domain kernels $\kappa_{R.A}$ for each domain can also be viewed as hyperparameters. However, most common data types allow for simple default choices.

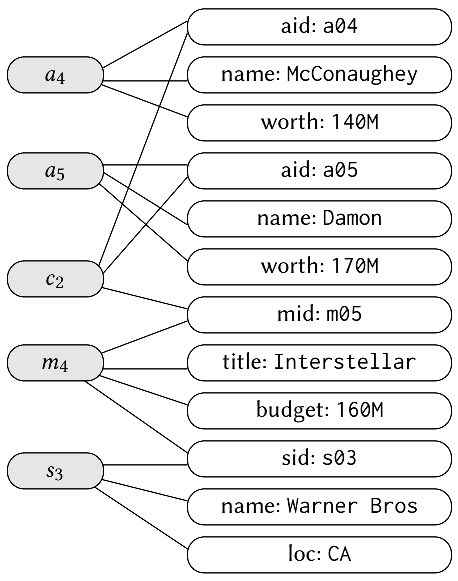


Figure 4: A partial graph of the database in Figure 2.

5 ADAPTATION OF NODE2VEC

We now describe an algorithm for dynamic database embedding that is based on adapting Node2Vec to relational databases. In the static setting, it is in the spirit of the work of Cappuzzo, Papotti and Thirumuruganathan [10], where we first build a graph that represents the database and then apply a Node2Vec—graph embedding based on random walks.

Specifically, we build a bipartite graph such that one side represents the facts and the other side represents the attribute values that occur in the facts, as illustrated in Figure 4 for a fragment of the database of Figure 2. We encode the attribute of a value together with the value itself; that is, the node is a pair that consists of both the attribute name and the attribute value. Formally, given a fact $f = R(a_1, \dots, a_k)$ over $R(A_1, \dots, A_k)$ we have a node v_f representing the fact f and the nodes $u_{f[A_1]}, \dots, u_{f[A_K]}$ representing the values $(a_1, \dots, a_k)$, along with the edges $\{(v_f, u_{f[A_i]}) | i \in [k]\}$. We denote $u_{f[A_i]}$ rather than u_{a_i} to ensure that value nodes are also bounded to a certain column. Hence, the same values in different attributes will have different value nodes.

In the above graph representation, we make an exception for FK values. Given the R-fact $f = R(a_1, \dots, a_k)$, the S-fact $g = S(b_1, \dots, b_k)$ and an FK φ of the form $R[\boldsymbol{B}] \subseteq S[\boldsymbol{C}]$ such that $f[\boldsymbol{B}] = g[\boldsymbol{C}]$, the nodes $\boldsymbol{U} = \{u_{f[B_i]} | B_i \in \boldsymbol{B}\}$ and $\boldsymbol{V} = \{v_{g[C_i]} | C_i \in \boldsymbol{C}\}$ are the same (i.e, $\boldsymbol{U} = \boldsymbol{V}$). For illustration, see Figure 4. Both actor1 and actor2 attributes reference aid, their nodes are the same, and the new attribute name is based on the original attribute name that appears in the matching FK constraint.

We chose the above graph representation over the specific one suggested by Cappuzzo et al. [10] since, in preliminary experiments, we found that our representation is slightly more efficient and better suited for handling multiple relations with foreign key constraints.

After constructing the graph, we run the Node2Vec [15] algorithm, which is widely used for embedding graph nodes, on this graph. As established by Cappuzzo et al. [10], this results in good performance in various tasks. We denote this method as N2V.

5.1 Extension to the Dynamic Setting

In this section we discuss three methods based on Node2Vec that allow us to extend the embeddings to new tuples, while leaving the original embeddings untouched. The topic of extending Node2Vec embeddings was already discussed in the context of dynamic graphs. Beres et al. [4] suggested an online extension, in the setting where the graph arrives as a stream of edges, and Mahdavi et al. [27] suggested a method based on evolving walks generation, in the setting where the input is a series of graphs that arrive at discrete timestamps. Both of these do not apply to our setting, because they do not allow freezing of the old embeddings

(i.e., in these methods the old embeddings change when new nodes arrive), although we use some ideas from Mahdavi et al. [27].

We consider three different methods for extending Node2Vec embeddings, *without* changing the old embedding. The first two methods rely on aggregating previously learned embeddings, while the third samples new walks and continues the training of the model without changing the old embeddings. We assume that we already have a Node2Vec model trained on the graph from the static phase.

5.1.1 Neighbors Based Aggregation. This method computes a new embedding based on the neighbors of the new node, which is the simplest and the most trivial method. Given the new graph $G = (V, E)$, the graph after inserting the new node v_n and its incident edges, we define $N(v_n) = \{u \in V | (u, v_n) \in E\}$, the neighbors of v_n. We define the new embedding of v_n to be the average of the embeddings of the nodes in $N(v_n)$. If $N(v_n) = \emptyset$, then the new embedding will be randomly drawn from a standard normal distribution.

5.1.2 Similar Tuples Aggregation. This is a more advanced aggregation method, based on averaging the embedding of similar tuple nodes. Given the new graph $G = (V, E)$, the set of old *tuple* nodes V_t, we define $ST(v_n) = \{u \in V_t | N(u) \cap N(v_n) \neq \emptyset\}$, the old tuple nodes that share at least one value node with v_n (as the graph is bipartite). Then, we define the embedding of v_n as the weighted average of the embedding of the nodes in $ST(v_n)$, weighted by the IoU (Intersection over Union) of the neighbors sets, that is, the ratio $\frac{|N(v_n) \cap N(u)|}{|N(v_n) \cup N(u)|}$. Again, if $ST(v_n) = \emptyset$ the new embedding will be randomly drawn from a standard normal distribution.

5.1.3 Frozen Training. In this method, we sample new walks from the new graph G, then continue training with the Node2Vec model while not changing the old embeddings. First, we save the underlying Node2Vec model from the static phase, then sample walks from the new (and complete) graph and continue training the model. Thus, we have an initial training phase for the static embedding. In the dynamic phase, we have an additional training phase after the arrival of every new node. The static phase is the same as a regular Node2Vec run. In the dynamic phase, we freeze the old embeddings during training such that only the embeddings for the new nodes will change - and continue the training the Node2Vec model only on the new embeddings, using gradient descent. The method tries to imitate the continuation of training the model, but without changing the previously learned embeddings.

6 EXPERIMENTAL EVALUATION

We now describe our experimental study. The goal of this study is to evaluate the quality of the embeddings produced, in both the static phase and (more importantly) the dynamic phase. Our quality evaluation is via a collection of downstream tasks of tuple classification. (We can also view the tuple-classification task as column prediction, where the class of the tuple can be seen as a new column.) We focus on databases that involve multiple relations, and particularly relations that are not the target of the downstream tasks; yet these relations can be used as context for inferring embeddings, as we do in our proposed embedding FoRWaRD. We first describe the downstream tasks (Section 6.1), the compared methods (Section 6.2), and the general runtime setup (Section 6.3). We then describe our experiments on the static setting (Section 6.4) and the dynamic setting (Section 6.5).

6.1 Datasets and Tasks

We now describe the datasets (tasks) of our experiments. The information on the structure of the datasets is summarized in Table 1. Each dataset is a database of multiple relations, where one relation contains an attribute that we wish to predict. Hereafter, we refer to this relation as the *prediction relation*. Note that neither FoRWaRD nor Node2Vec see the predicted attribute during training.

6.1.1 Hepatitis. This database is from the 2002 ECML/PKDD Discovery Challenge.[2] We use the modified version of Neville et al. [30]. The goal in this task is to predict the type column, which is either *Hepatitis B* or *Hepatitis C* based on medical examinations. There are in total 206 instances of the former and 484 cases of the latter. The relation with the predicted column contains, in addition to the type classification, the age, sex and identifier of the patient. The other relations contain the rest of the medical data. The dataset contains seven relations with a total of 26 attributes and 12,927 tuples.

6.1.2 Mondial. This dataset contains information from multiple geographical resources [40]. We predict the religion of a country. There are 114 countries classified as *Christian* and 71 as *non-Christian*. The prediction is based on a variety of fields such as the language, population, geography, and government of the country. The target relation, where we predict a column, is binary—it contains only the name of the country and the (predicted) classification. The dataset contains 40 different relations with a total of 167 attributes and 21,497 tuples. We use the whole database and use the Target relation as the prediction relation as previously done by Bina et al. [5].

6.1.3 Genes. This dataset is from the KDD 2001 competition [11], and contains data from genomic and drug-design applications. We predict the localization of the gene, based on biological data, with 15 different labels. The prediction relation contains only the class and an identifier for the gene, while the rest contain the biological data such as the function, gene type, cellular location and the expression correlation between different genes. The dataset contains 3 relations with a total of 15 attributes and 6,063 tuples. We remove two tuples which have a unique class to prevent split in-balances during cross-validation.

6.1.4 Mutagenesis. This dataset contains data on the mutagenicity of molecules on Salmonella typhimurium [12]. We predict the mutagenicity (the mutagenic attribute) of the molecules, based on chemical properties of the molecule, with 122 positive samples and 63 negative samples. The prediction relation contains the binary class, molecule ID, and some of the chemical data, while the other relations contain more chemical data and information about the relations between the molecules. The dataset contains 3 relations with a total of 14 attributes and 10,324 tuples. Note that we do not use any external features, in contrast to some past methods for this dataset [26] that we revisit later on.

6.1.5 World. This dataset contains data on states and their cities. We predict the continent of a country with 7 different labels. The prediction is based on general data on the country such as population, GNP, Capital city and information on the spoken languages and cities. The dataset contains 3 relations, with a total of 24 attributes and 5,411 tuples.

6.2 Compared Methods

We compare between the following alternatives.

- N2V: In the static phase, this is the Node2Vec method, with our own implementation that is based on the original paper. In the dynamic phase, we use the three adaptations of Node2Vec that we described in Section 5.1:
  - $\mathrm{N2V}_{\mathrm{FT}}$: the *frozen-training* method described in Section 5.1.3.
  - $\mathrm{N2V}_{\mathrm{NA}}$: the *neighbor aggregation* method described in Section 5.1.1.
  - $\mathrm{N2V}_{\mathrm{ST}}$: the *similar-tuple aggregation* method described in Section 5.1.2.
- S.o.A.: These are state-of-the-art methods that solve the multi-relational classification problem without using an embedding, such as multi-relational decision trees and forests [3, 5], multi-relational Bayes nets [37] and Inductive Logic Programming (ILP) [26, 42]. This applies only to the static experiment.
- $\mathrm{FWD}_{\mathrm{EK}}$: This is the version of FoRWaRD that is based on the kernel $K_{\mathbf{EK}}$ of Equation (2).
- $\mathrm{FWD}_{\mathrm{MMD}}$: This is the version of FoRWaRD that is based on the kernel $K_{\mathbf{MMD}}$ of Equation (3).

In the next section, we discuss the implementation of the N2V and FoRWaRD variants.

In all datasets and experiments, we have a full separation between the embedding process and the downstream task. This means that we generate the embedding independently from the task (as opposed to training for the task), and then use these embeddings as the input to a downstream classifier (that sees only the embeddings and none of the other database information).

Specifically, we train and apply an SVM classifier (Scikit-learn's SVC implementation) as the downstream machine-learning architecture, in both experiments. Performance assessment is conducted via k-fold cross validation with $k = 10$ folds.

[2]https://sorry.vse.cz/~berka/challenge/PAST/

Table 1: Information about the structure of the datasets used in the experiments. The "Common Class" column refers to the frequency of the common value of the predicted bit.

Dataset	Prediction Rel.	Prediction Attr.	#Samples	#Relations	#Tuples	#Attributes	Common Class
Hepatitis	Dispat	type	500	7	12927	26	58.8%
Genes	Classification	localization	862	3	6063	15	42.5%
Mutagenesis	Molecule	mutagenic	188	3	10324	14	66.4%
World	Country	continent	239	3	5411	24	24.2%
Mondial	Target	target	206	40	21497	167	54.8%

6.3 Experimental Setup

6.3.1 Implementation. We implemented Node2Vec on our own based on its original publication [15]. For the state-of-the-art results (S.o.A.) we used the reported numbers. The algorithms Node2Vec, FWD_{EK} and FWD_{MMD} (in both static and dynamic versions) are implemented in Python using PyTorch [32], Numpy [19] for the numerical operations, Scikit-learn [33] for the downstream classifiers and validation, and NetworkX [18] for the graph implementations. The code is publicly available on github[3].

Note that in the FoRWaRD implementations we embed only the relation that contains the tuples that we wish to classify. We use the default kernels in all of our experiments: Gaussian distance for numbers, and equality for all other data types.

6.3.2 Hyperparameters. The hyperparameters of the FoRWaRD and Node2Vec implementations are listed in Table 2. Note that we use different hyperparameters for the Genes dataset, since the distribution of values between the relations is different compared to the rest of the datasets. In the dynamic experiment the number of epochs for the second training phase of N2V_{FT} was set to 5 and $n^{\text{new}}_{\text{samples}}$ was set to 2500 for all FoRWaRD runs.

6.3.3 Hardware. We run the experiments on a server with 2 Intel Xeon E5-2683 v4 processors, 256 GB RAM, and a Nvidia GTX 1080 Ti GPU. We utilize the GPU in all methods to achieve better runtimes.

6.4 Results for Static Database Embeddings

Table 3 summarizes the results for the static case. Also note that for each of the ten folds, we train a new embedding. Hence, to account for the randomness of both the folds and the embeddings, we report the standard deviation ($\pm x$) next to each number. Note that the embedding methods always see the full database, and the downstream classifier uses the different splits. For the state-of-the-art (S.o.T.), we use the reported results from the publications that we mention in the table.

As can be seen in Table 3, our method performs better than the state-of-the-art methods on most datasets and is competitive with dataset-specific methods even without using external knowledge. Furthermore, the experiment shows that the embeddings produced

[3]https://github.com/netafr/Dynamic-Database-Embeddings-with-FoRWaRD

Table 2: Hyperparameters in the FoRWaRD implementations. In the Genes dataset we use #samples of 1,000, batch size of 10,000, and 10 epochs.

Alg.	Param.	Value
$\text{FoRWaRD}_{\text{EK}}$	embedding dim. (k)	100
	#samples (n_{samples})	5,000
	batch size	50,000
	max walk len. ($\ell_{\max}$)	1–3
	#epochs	5–10
$\text{FoRWaRD}_{\text{MMD}}$	embedding dim. (k)	100
	#samples (n_{samples})	5,000
	batch size	5,000
	max walk len. ($\ell_{\max}$)	1–3
	#epochs	25
Node2Vec	embedding dim.	100
	#walks per node	40
	#steps per walk	30
	context window	5
	#neg/#pos samples	20
	batch size	40,000
	#epochs	10

Table 3: Accuracy for static classification, including standard deviation (±). S.o.A. stands for state-of-the-art, where we take the best result via a general (non-dataset-specific) method.

Task	FWD_{EK}	FWD_{MMD}	N2V	S.o.A.
Hepatitis	84.20% ±4.94	85.40% ±4.2	**93.60%** ±2.5	84.00%(*) [5]
Genes	97.91% ±0.87	**99.42%** ±0.58	97.19% ±1.25	85.00% [3]
Mutagenesis	**90.00%** ±7.96	**90.00%** ±7.24	88.23% ±4.56	**91.00%**(**) [42]
World	85.83% ±5.34	81.25% ±6.25	**94.00%** ±4.4	77.00% [42]
Mondial	80.95% ±6.73	81.56% ±6.57	77.62% ±5.24	**85%** [37]

(*) 95% achieved in a method specific for the Hepatitis dataset [2].

(**) 96% achieved in a method specific for the Mutagenesis dataset [26].

with FoRWaRD are as good as the embeddings produced by the Node2Vec based algorithm even for static classification, and that the Node2Vec method also produces state-of-the-art competitive results. Finally, we can see that both FoRWaRD variants achieve similar results in all of the datasets.

We also offer some interesting insights about the algorithms:

- Node2Vec preforms better on Hepatitis and World - two datasets where most of the information lays in *categorical* data. This is in-fact expected as the graph that is used by Node2Vec encodes this data very well.
- The standard deviation of all methods are on the same scale in all datasets.
- Both Node2Vec and FoRWaRD excel at the Genes dataset, managing to capture the database's structure almost perfectly. Generally, both methods are competitive with the state-of-the-art in all of our datasets.

All in all, we can conclude that both Node2Vec and FoRWaRD achieve very good results on these datasets, and that the structure of the database affects the quality of the embeddings - and both manage to capture the structure very well.

6.5 Results for Dynamic Database Embeddings

6.5.1 Experiment Description. In this experiment we look to measure how well our method can produce new embeddings when new data arrives. We test that by splitting the database into two parts and treat the first part as the static database, and the other part as the new data that arrives. The experiment is comprised of five steps:

(1) Partition all facts of the database into two sets $\mathcal{F}_{\text{old}}$ and $\mathcal{F}_{\text{new}}$. The facts in $\mathcal{F}_{\text{new}}$ are then removed from the database.
(2) Train an embedding *only* on the static part of the split.
(3) Label the vectors in this embedding using the correct class labels and train a downstream classifier on these labelled data.
(4) Add the facts from from $\mathcal{F}_{\text{new}}$ back to the database to simulate the arrival of new data. Generate new embeddings for these new facts.
(5) Evaluate the trained classifier on the only on the embeddings of "new" data.

It is important to note that, by (5), the accuracy results we obtain refer exclusively to new data added after the embedding was trained. Rather than adding the new tuples one-by-one and computing a new embedding after each arrival, for efficiency reasons we added all new tuples at once. In particular for N2V$_{\text{FT}}$, sequential updates would have come with the significant overhead of a training phase after every new tuple insertion. To keep the results comparable, we used batch updates for FoRWaRD as well.

The first question that arises is how to split the data in a way that simulates a real-life scenario. In real-world scenarios, new facts tend to arrive in batches that are distributed across all relations. For our example database from Figure 2, a new fact that is added to the Movies relation may be accompanied by new actors or new collaborations that are also added to the database. In our experiments we would like to simulate this behavior, where the new facts that arrive are are semantically related. Note that such a setup is more challenging than a simple random partition, since the new facts are less connected to the old data.

Formally, we compute the partition as follows. We first partition the prediction table according to a specified ratio of *old* and *new* tuples. This split is randomly chosen and stratified, i.e. the classes of the downstream task are (roughly) equally distributed in both partitions. We then remove the *new* tuples from the prediction relation. Based on this initial removal we remove facts from the remaining relation of the database. More specifically, we iterate over the relations and remove all orphaned child facts as well as all childless parent facts. This processes is similar to an "On Delete Cascade" deletion in SQL, except that all foreign keys are viewed as symmetric and therefore, parent facts whose children have all been deleted are also deleted. The facts that are deleted in this process form the $\mathcal{F}_{\text{new}}$ partition of our experiment. In that way, we compute a partition where the facts in the new data partition are semantically related and distributed across the whole database.

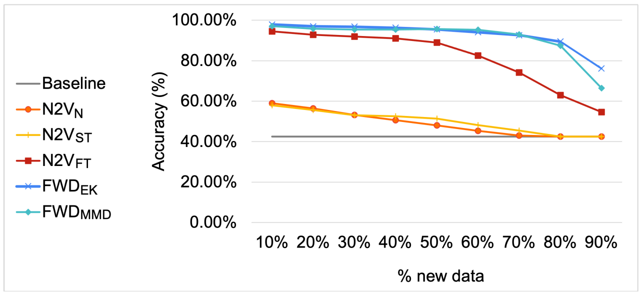
Example 6.1. Consider again the database of Figure 2 and suppose that our prediction relation is Collaborations. If we remove the fact c_1, then we will also remove the fact m_4 (Interstellar) from the Movies relation, and the tuple a_2 (Watanabe) from the Actors relation. Note that we will *not* remove a_1 (DiCaprio) as it is still connected to non-removed facts in other relations (specifically, it is connected to c_4 in the Collaborations relation). ◆

In our experiment we vary the ratio of old and new facts to understand how the relative amount of data effects the quality of the induced emddings for the new data. For each tested ratio, we run the experiment 10 times with different partitions of the database. We measure the evaluation accuracy of the downstream classifier on the new data and the corresponding standard deviation across the 10 runs.

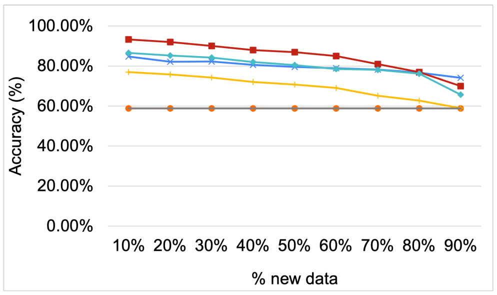
6.5.2 Results. Figure 5 presents the results of this experiment. The term "baseline" here means the accuracy obtained by always predicting the most common class.

The experiment shows that both FWD$_{\text{EK}}$ and FWD$_{\text{MMD}}$ have similar performance, achieving high accuracy throughout all percentages of new data, and having a more consistent performance than the Node2Vec based methods. It is not surprising that the aggregation-based Node2Vec versions N2V$_{\text{NA}}$ and N2V$_{\text{ST}}$ perform poorly compared to FoRWaRD. On the other hand, the frozen-training method N2V$_{\text{FT}}$ proved as a good way to compute embeddings for new data solely based on Node2Vec—it had reasonable results in all of the datasets and was even better than FoRWaRD in two of the datasets. This result shows that when continuing the Node2Vec training one can selectively choose embeddings to freeze and still achieve good results.

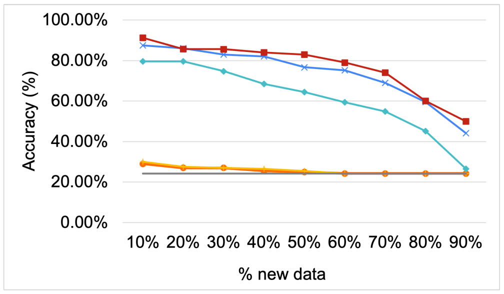
Overall, we can see that FoRWaRD and its dynamic extension empirically achieve the goal of producing viable embeddings, while keeping the old embeddings intact. In most of the datasets, the accuracy is almost as good as the static case, even when up to 50% of the data is new. In addition, we also showed that the trivial graph-based Node2Vec methods for extending the embeddings fail to work consistently. Finally, we showed that the method of frozen-training works fairly well, although not as good and not as stable as FoRWaRD across all datasets. Thus, FoRWaRD proved



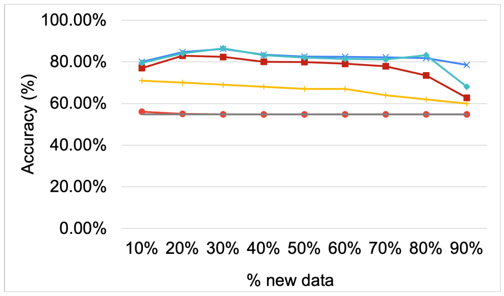
(a) Genes



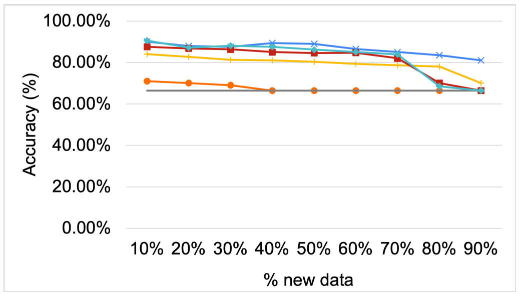
(b) Hepatitis



(c) World



(d) Mondial



(e) Mutagenesis

Figure 5: Results for the dynamic experiment - baseline is the accuracy of guessing the most common class. We report the accuracy as a function of the percentage of new tuples.

Table 4: Accuracy and standard deviation (±) for the dynamic experiment with a ratio of 10% new tuples.

Task	$N2V_{FT}$	FWD_{EK}	FWD_{MMD}
Hepatitis	**93.34%**±2.7	82.20%±4.94	86.6%±3.58
Genes	94.50%±1.89	**97.91%**±0.87	97.07%±1.19
Mutagenesis	87.58%±7.8	**90.00%**±6.84	**90.53%**±6.14
World	**91.25%**±4.95	87.50%±3.73	79.58%±7.32
Mondial	77.62%±6.75	**80.00%**±7.32	79.52%±7.69

to be a very good method for creating embeddings for new tuples compared to Node2Vec based methods. Furthermore, we can see that the rate which the accuracy deteriorates is also better when using FoRWaRD, even when most of the data is new and even when looking at datasets where Node2Vec has the advantage (e.g., World).

Interestingly, while $K_{\mathbf{MMD}}$ was a better choice in the static case, it seems that in the dynamic case $K_{\mathbf{EK}}$ produces better results across all datasets. This can be explained by the fact that in the $K_{\mathbf{MMD}}$ variant, for each walk scheme we sample only *once* per pair of tuples, thus having less information to learn from during the dynamic phase. Furthermore, at Table 4 we look to compare both FoRWaRD versions and $N2V_{FT}$ looking at a the specific scenario of 10% new tuples, as a more convient way to look at specific numbers. We can see that in most datasets the standard deviation is in a similar order of magnitude as in the static experiment although it is a little higher which is expected. In addition, we can see that already at 10% of new tuples, the rate which the accuracy for FoRWaRD deteriorates is better than $N2V_{FT}$.

Table 5: Execution times (in seconds) to compute static embeddings with $N2V_{FT}$, FWD_{EK} and FWD_{MMD}.

Task	$N2V_{FT}$	FWD_{EK}	FWD_{MMD}
Hepatitis	**188.5**	540	310
Genes	**77.9**	204	373
Mutagenesis	166.2	230	**100**
World	**218.7**	440	405
Mondial	**461.6**	810	562

6.6 Execution Times

We look to compare FoRWaRD's execution times to that of Node2Vec in both the static and dynamic case. As reported in Table 5 for the static classification experiments, we can see that Node2Vec is faster than FoRWaRD in most of the datasets, and with a different margin for each dataset. This is due to the different structure of the databases. In addition, we can see that in most cases using FoRWaRD with $K_{\mathbf{MMD}}$ is faster than the $K_{\mathbf{EK}}$ version.

Finally, we are also interested in the average time for generating an embedding for a newly arrived tuple using our method. The results are shown in Table 6. We compare both FoRWaRD variants with the $N2V_{FT}$ as the other Node2Vec based methods fail at producing new embedding with high quality. The reported numbers are the average time (in seconds) it takes to embed a newly-arrived tuple - we already have the embedding for the old tuples, and we measure only how much time it takes to infer a new embedding for a new tuple. Note that these numbers change dramatically from one dataset to another, as according to the experiment each new tuple in the classification relation is also accompanied by new tuples from other relations, thus the structure of the database affects the execution time. As can be seen, FWD_{MMD} is the fastest method when it comes to generating embeddings for new tuples - in the vast majority of cases it is much faster than both Node2Vec and FWD_{EK}. The difference between the FoRWaRD variants is mainly due to the fact that when using $K_{\mathbf{MMD}}$ we sample less tuple-pairs than when using $K_{\mathbf{EK}}$. Note that although FWD_{MMD} is much faster in embedding new tuples, the quality of the embedding is lower as discussed in Section 6.5.

Overall, we conclude that although Node2Vec is usually faster at embedding a whole database, FoRWaRD is faster when it comes to embedding new tuples that arrive after the initial embedding. At hindsight, this is unsurprising, because embed a new tuple with forward only requires solving a system of linear equations, whereas embedding a new tuple with $N2V_{FT}$ requires a new training phase.

Remark 1. *Note that our experiments give a major runtime advantage to the Node2Vec algorithm: it would be much slower if we added the tuples one at a time (or few at a time), as we would need to re-train after each insertion. For FoRWaRD, it does not make a big difference if we insert tuples in a batch or one at a time, because we have to solve system of equations for each tuple anyway. We implemented the updates of Node2Vec in a batch since otherwise the experiments would have been too slow.*

7 CONCLUDING REMARKS

We studied the problem of computing an embedding for database tuples, focusing on dynamic databases where the embedding should be extensible to arriving tuples. Past techniques for the static database embedding problem are not designed for dynamic database embedding. Our new algorithm FoRWaRD is designed specifically for the dynamic embedding problem, yet we can experimentally show that it performs well in the static as well as the dynamic settings.

There are several future directions for enhancing FoRWaRD and investigating its performance. It is important to optimize the execution cost of FoRWaRD, particularly in the case of batch updates with missing information (nulls). It will also important to understand the performance of FoRWaRD (in comparison to alternatives) on machine-learning tasks other than column prediction: record linking [13, 29], entity resolution [10, 21], data imputation [22, 41], data cleaning [1, 36, 41] and so on.

We argued in the Introduction that tuple deletion is a trivial operation in our dynamic setup where we preserve the embedding of existing tuples. This is why we focused on tuple insertions in this paper. However, there is a subtle issue about deletions that poses interesting questions somewhat orthogonal to what we study here. When deleting a tuple t^- and the corresponding point from the embedding, we do not delete all information about t^-, because the existence of the tuple had impact on how the other tuples where embedded. This has consequences for privacy considerations. For instance, if a user wants to be deleted from a database, then all information about the user must be deleted; this may even be a legal requirement. For this, the embedding of the remaining tuples needs to be adapted. Rather than re-computing the whole embedding, we may try to find a minimal set of changes that removes all information about the deleted tuple, but keeps the overall embedding intact.

Table 6: Average time (in seconds) to embed a new tuple for $N2V_{FT}$, FWD_{EK} and FWD_{MMD}.

Task	$N2V_{FT}$	FWD_{EK}	FWD_{MMD}
Hepatitis	0.265	0.62	**0.083**
Genes	**0.062**	0.176	0.08
Mutagenesis	0.65	0.28	**0.198**
World	0.64	0.733	**0.36**
Mondial	1.55	1.09	**1.05**

REFERENCES

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting Data Errors: Where are we and what needs to be done? *Proc. VLDB Endow.* 9, 12 (2016), 993–1004.

[2] Mohammed H Afif, Abdel-Rahman Hedar, Taysir H Abdel Hamid, and Yousef B Mahdy. 2013. SS-SVM (3SVM): a new classification method for hepatitis disease diagnosis. *Int. J. Adv. Comput. Sci. Appl* 4 (2013).

[3] Anna Atramentov, Hector Leiva, and Vasant Honavar. 2003. A Multi-relational Decision Tree Learning Algorithm – Implementation and Experiments. In *Inductive Logic Programming*, Tamás Horváth and Akihiro Yamamoto (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 38–56.

[4] Ferenc Béres, Domokos M Kelen, Róbert Pálovics, and András A Benczúr. 2019. Node embeddings in dynamic graphs. *Applied Network Science* 4, 1 (2019), 1–25.

[5] Bahareh Bina, Oliver Schulte, Branden Crawford, Zhensong Qian, and Yi Xiong. 2013. Simple decision forests for multi-relational classification. *Decis. Support Syst.* 54, 3 (2013), 1269–1279. https://doi.org/10.1016/j.dss.2012.11.017

[6] Rajesh Bordawekar, Bortik Bandyopadhyay, and Oded Shmueli. 2017. Cognitive Database: A Step towards Endowing Relational Databases with Artificial Intelligence Capabilities. *CoRR* abs/1712.07199 (2017). arXiv:1712.07199 http://arxiv.org/abs/1712.07199

[7] Rajesh Bordawekar and Oded Shmueli. 2017. Using Word Embedding to Enable Semantic Queries in Relational Databases. In *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning* (Chicago, IL, USA) *(DEEM'17)*. ACM, New York, NY, USA, Article 5, 4 pages. https://doi.org/10.1145/3076246.3076251

[8] Rajesh Bordawekar and Oded Shmueli. 2019. Exploiting Latent Information in Relational Databases via Word Embedding and Application to Degrees of Disclosure. In *CIDR*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p27-bordawekar-cidr19.pdf

[9] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*. 2787–2795.

[10] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In *SIGMOD*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1335–1349. https://doi.org/10.1145/3318464.3389742

[11] Jie Cheng, Christos Hatzis, Hisashi Hayashi, Mark-A. Krogel, Shinichi Morishita, David Page, and Jun Sese. 2002. KDD Cup 2001 Report. *SIGKDD Explor. Newsl.* 3, 2 (Jan. 2002), 47–64. https://doi.org/10.1145/507515.507523

[12] Asim Kumar Debnath, Rosa L. Lopez de Compadre, Gargi Debnath, Alan J. Shusterman, and Corwin Hansch. 1991. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry* 34, 2 (1991), 786–797. https://doi.org/10.1021/jm00106a046 arXiv:https://doi.org/10.1021/jm00106a046

[13] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq R. Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed Representations of Tuples for Entity Resolution. *Proc. VLDB Endow.* 11, 11 (2018), 1454–1467.

[14] Martin Grohe. 2020. word2vec, node2vec, graph2vec, X2vec: Towards a Theory of Vector Embeddings of Structured Data. In *PODS*, Dan Suciu, Yufei Tao, and Zhewei Wei (Eds.). ACM, 1–16. https://doi.org/10.1145/3375395.3387641

[15] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. ACM, New York, NY, USA, 855–864. https://doi.org/10.1145/2939672.2939754

[16] Michael Günther. 2018. FREDDY: Fast Word Embeddings in Database Systems. In *SIGMOD Conference*. ACM, 1817–1819.

[17] Michael Günther, Maik Thiele, Erik Nikulski, and Wolfgang Lehner. 2020. RetroLive: Analysis of Relational Retrofitted Word Embeddings. In *EDBT*. OpenProceedings.org, 607–610.

[18] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). 11 – 15.

[19] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[20] Sabrina Jaeger, Simone Fulle, and Samo Turk. 2018. Mol2vec: Unsupervised Machine Learning Approach with Chemical Intuition. *J. Chem. Inf. Model.* 58, 1 (2018), 27–35.

[21] Shrinu Kushagra, Shai Ben-David, and Ihab F. Ilyas. 2019. Semi-supervised clustering for de-duplication. In *AISTATS (Proceedings of Machine Learning Research)*, Vol. 89. PMLR, 1659–1667.

[22] Kamakshi Lakshminarayan, Steven A. Harp, and Tariq Samad. 1999. Imputation of Missing Data in Industrial Databases. *Appl. Intell.* 11, 3 (1999), 259–275.

[23] Jey Han Lau and Timothy Baldwin. 2016. An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation. In *Rep4NLP@ACL*. ACL, 78–86.

[24] Quoc V. Le and Tomás Mikolov. 2014. Distributed Representations of Sentences and Documents. In *ICML (JMLR Workshop and Conference Proceedings)*, Vol. 32. JMLR.org, 1188–1196.

[25] Yang Li and Tao Yang. 2018. *Word Embedding for Understanding Natural Language: A Survey.* Springer International Publishing, Cham, 83–104.

[26] Huma Lodhi and Stephen Muggleton. 2005. Is mutagenesis still challenging?. In *ILP*. 35–40.

[27] Sedigheh Mahdavi, Shima Khoshraftar, and Aijun An. 2018. dynnode2vec: Scalable Dynamic Network Embedding. *CoRR* abs/1812.02356 (2018). arXiv:1812.02356 http://arxiv.org/abs/1812.02356

[28] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, and J. Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems*. 3111–3119.

[29] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD Conference*. ACM, 19–34.

[30] Jennifer Neville, David D. Jensen, Lisa Friedland, and Michael Hay. 2003. Learning relational probability trees. In *KDD*. ACM, 625–630.

[31] M. Nickel, V. Tresp, and H.-P. Kriegel. 2011. A three-way model for collective learning on multi-relational data. In *Proceedings of the 28th International Conference on Machine Learning*. 809–816.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[34] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 1532–1543. https://doi.org/10.3115/v1/d14-1162

[35] B. Perozzi, R. Al-Rfou, and S. Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 710–710.

[36] Christopher De Sa, Ihab F. Ilyas, Benny Kimelfeld, Christopher Ré, and Theodoros Rekatsinas. 2019. A Formal Framework for Probabilistic Unclean Databases. In *ICDT (LIPIcs)*, Vol. 127. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:18.

[37] Oliver Schulte, Bahareh Bina, Branden Crawford, D. Bingham, and Yi Xiong. 2013. A hierarchy of independence assumptions for multi-relational Bayes net classifiers. In *IEEE Symposium on Comxputational Intelligence and Data Mining, CIDM 2013, Singapore, 16-19 April, 2013*. IEEE, 150–159. https://doi.org/10.1109/CIDM.2013.6597230

[38] Tobias Schumacher, Hinrikus Wolf, Martin Ritzert, Florian Lemmerich, Jan Bachmann, Florian Frantzen, Max Klabunde, Martin Grohe, and Markus Strohmaier. 2020. The Effects of Randomness on the Stability of Node Embeddings. *ArXiv* 2005.10039 [cs.LG] (2020). https://arxiv.org/abs/2005.10039

[39] Vraj Shah, Arun Kumar, and Xiaojin Zhu. 2017. Are key-foreign key joins safe to avoid when learning high-capacity classifiers? *arXiv preprint arXiv:1704.00485* (2017).

[40] Ben Taskar, Pieter Abbeel, and Daphne Koller. 2002. Discriminative Probabilistic Models for Relational Data. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence* (Alberta, Canada) *(UAI'02)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 485–492.

[41] Richard Wu, Aoqian Zhang, Ihab F. Ilyas, and Theodoros Rekatsinas. 2020. Attention-based Learning for Missing Data Imputation in HoloClean. In *MLSys*. mlsys.org.

[42] X. Yin, J. Han, J. Yang, and P. S. Yu. 2006. Efficient classification across multiple database relations: a CrossMine approach. *IEEE Transactions on Knowledge and Data Engineering* 18, 6 (2006), 770–783. https://doi.org/10.1109/TKDE.2006.94