Leva: Boosting Machine Learning Performance with Relational Embedding Data Augmentation

Zixuan Zhao
The University of Chicago
Chicago, USA
zhaozixuan@uchicago.edu

Raul Castro Fernandez
The University of Chicago
Chicago, USA
raulcf@uchicago.edu

ABSTRACT

In this paper, we present Leva, an end-to-end system that boosts the performance of machine learning tasks over relational data. Leva builds a relational embedding by representing relational data as a graph and then using embedding methods to represent the graph as vectors. The embedding represents information from the entire database, including useful information for the downstream machine learning task. At the same time, some information in the graph will be erroneous, for example, corresponding to incorrect inclusion dependencies. However, we show that the supervision signal from the downstream task filters out information that is not useful. The result is a boost in ML performance. This result means that it is possible for analysts to avoid the time-consuming effort of collecting features across multiple relations—which requires solving a data discovery and integration problem—and instead rely on these techniques to train better-performing models. We demonstrate Leva's performance on different classification and regression datasets and compare it with multiple other baselines.

CCS CONCEPTS

• **Information systems** → **Data management systems**; *Mediators and data integration.*

KEYWORDS

data discovery; feature engineering; training data augmentation

ACM Reference Format:
Zixuan Zhao and Raul Castro Fernandez. 2022. Leva: Boosting Machine Learning Performance with Relational Embedding Data Augmentation. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/XXXXXX.YYYYYY

1 INTRODUCTION

There is an explosion of interest in applying machine learning in organizations that is slowed down by "data preparation work" [36]. The training datasets used to fit machine learning models consist of features and a target variable. The features are often stored as attributes in tables that may be sprinkled throughout the organization.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN XXXXX.YYYYY.ZZZZZ...$15.00
https://doi.org/10.1145/XXXXXX.YYYYYY

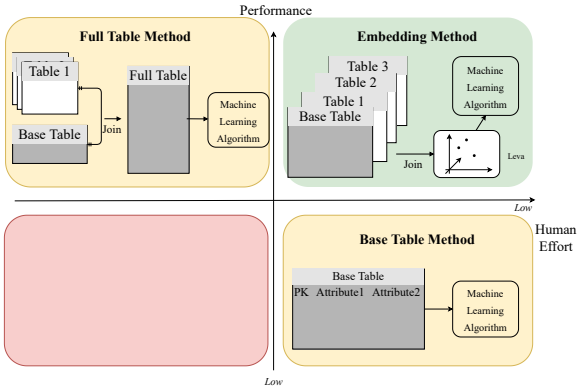


Figure 1: Performance and Effort Tradeoff During the creation of a Training Dataset

To build a training dataset, an analyst must solve three problems: feature engineering, discovery, and integration. First, the analyst must determine what features are relevant for the problem at hand. Then they need to identify what tables and databases among the many available in modern organizations contain those features. Finally, they need to combine those tables into a single one, which corresponds to the desired training dataset. The discovery problem is challenging because of the volume of tables and the lack of guidance to choosing the right features. The integration problem is challenging because of the lack of *keys* and *join paths* that indicate how to correctly combine tables. Although join information is often available within individual databases, it does not exist across databases. Despite the difficulty, analysts are pressed to address these problems because identifying the right features is crucial to achieving good quality on the downstream machine learning task. Today, analysts are forced to make an uncomfortable decision:

• **Base Table.** Analysts could stick to the table that contains the target variable—called *Base Table*—to train the ML model. Unfortunately, this means they may leave behind many features that would boost the performance of the model.

• **Full Table.** Alternatively, the analyst could aim to join the Base Table with as many tables as possible, hoping the relevant features are included in the final result. The search could be more guided if the analyst knew what features are relevant to the problem, but this information is often not available a priori. Unfortunately, this *Full Table* approach requires identifying what tables are joinable and how to join them, which is time-consuming.

• **Full Table + Feature Engineering.** To improve on the previous option, the analyst could run a feature engineering algorithm on the *Full Table*, using compute time to detect features that will be most helpful for the ML task.

Base Table does not require addressing the discovery and integration problem, but it potentially yields low performance (bottom-right quadrant in Fig. 1). Full Table may achieve much higher performance but at the great cost of time spent in identifying what tables may be helpful and how to combine them: *an effort that has to be repeated for every new ML task*. Consequently, Full Table sits on the upper-left quadrant of Fig. 1. In summary, either analysts invest time and resources in joining multiple tables with the goal of improving downstream machine learning tasks—assuming the risk of not achieving so—or they settle for a simpler training dataset that will lead to worse performance of the ML task but is less time-consuming. Both options reduce the potential value of machine learning within organizations.

This undesirable tradeoff has spun interesting research contributions that include dataset augmentation for AutoML pipelines [15], principled approaches to determine when joining two datasets is profitable for the target task [27], and many others [38]. These approaches assist with a process that remains otherwise manual, or as a boost to fully automated AutoML pipelines. However, these state-of-the-art approaches assume knowledge of the correct way to join tables. We target a *more general scenario, where we do not know table keys and we do not know join paths*. Our goal is to provide an approach that does not require effort from users to discover and integrate data and that achieves high quality on downstream tasks; we want a method that sits in the top-right quadrant in Fig. 1.

In this paper, we introduce Leva, a system to construct a *relational embedding* that represents elements of the relational data as vectors. The embedding is used to featurize the Base Table. Because each vector in the embedding is a distributed representation over all tokens in the database, useful information from other tables is included in the featurized training dataset and that boosts the performance of the downstream ML task. Crucially, the performance achieved with the embedding closely resembles the performance of manually engineered datasets, removing the awkward tradeoff of choosing between convenience and performance.

The main contribution of Leva is the *strategy employed to reconstruct join information* across tables without prior knowledge of table keys or other schema information. Leva constructs a graph that represents the relational information, including approximate inclusion dependencies—some of which may be unavoidably wrong—and then embeds that graph into a high-dimensional embedding. Despite the existence of wrong inclusion dependencies, the embedding boosts the performance of downstream machine learning tasks. This is because the supervision signal from the downstream ML task removes edges that do not bring useful information, including spurious inclusion dependencies. Then, by bringing information from other tables into the Base Table and using the supervision signal to adapt the embedding at training time, the relational embedding we present in this paper boosts the performance of downstream machine learning tasks without user intervention.

We evaluate Leva on common classification and regression tasks using multiple real and synthetic datasets. We show that the embedding representation outperforms the Base Table method and matches the performance of Full Table + Feature Engineering despite minimal human input. We show that Leva scales to much larger datasets than previous embedding methods. We complement the evaluation results with a study of the effects of different embedding methods, which we believe will contribute to this growing area in data management, and we include ablation experiments that explore various design decisions we make.

Table Name	Attributes
Expenses	Name, Gender, School Name, Total Expenses
Order Info	Name (FK, ref Expenses), Item (FK, ref Price Info)
Price Info	Item, Prices

Table 1: Schemas for Synthetic Dataset: STUDENT

In summary, the contributions of this paper are:

- Relational embeddings specifically constructed to boost the performance of downstream ML models.
- An end-to-end system, Leva, that takes as input a list of tables and produces a ready-to-use relational embedding.
- An analysis of the hypothesis that explains why the relational embedding works in downstream tasks. We believe our insights will be useful to build upon these methods, which have applications in other areas of relational data management.

The paper outline is as follows. Section 2 presents the background and the problem statement. Section 3 explains the core graph construction logic of converting relational data into graphs. Section 4 covers how the graph construction module lies in the whole pipeline of Leva. Section 5 justifies why the embedding works. We present evaluation results in Section 6, related work in Section 7 and conclusions in Section 8.

2 BACKGROUND AND PROBLEM SETTING

In Sections 2.1 and 2.2, we discuss current approaches to assemble training datasets for machine learning problems over relational data. We review the embedding literature in Section 2.3. We finish the section with a problem statement in Section 2.4.

2.1 Choosing a Training Dataset

A supervised machine learning task, such as classification and regression, learns functions that match input-output pairs based on its training data. Formally, it requires data to be the form (X, Y) where X represents the set of available features, and Y is the target for the prediction tasks. The goal is to find a model that minimizes both the error on the training dataset and the empirical error on previously unseen data X', which is often referred to as the test set. In the relational data scenario, both X and Y come from table attributes. However, since real-world data is often scattered over multiple tables and databases, it requires extensive human effort to identify good attributes and to join them to form the input X.

Consider a STUDENT dataset with three tables (schema and attributes listed in Table 1). The underlined attributes correspond to the keys of each table, and in parentheses are the foreign keys that some attributes are referring to. Consider an ML task where the goal is to predict the value of *Total Expenses* (in table *Expenses*). For illustration purposes, we assume that *Total Expenses* is fully explained by order information (in table *Order Info*) and it is uncorrelated with *Gender* and *School Name*. In this case, *Total Expenses* is the response variable, Y. To train a model to predict Y, we need to find a set of explanatory variables, X. One straightforward choice is

to use only attributes in table *Expense*, excluding Y. We call the table that contains the target attribute/feature, Y, the **Base Table**. Using the *Base Table* as training data is effortless but the approach may leave potential performance untapped because the attributes *Gender* and *School Name* are not good predictors of the target variable. This approach corresponds to the low effort but low performance quadrant of Fig. 1. Instead, we wish to use all relevant information across tables to boost the ML task's performance.

2.2 Machine Learning over Relational Data

Another option is to assemble a training dataset by joining multiple tables from different databases, aiming to bring in relevant features for the downstream task. This is the **Full Table** approach. In this scenario, an analyst needs to solve two problems: identify relevant features and identify where they are stored. For example, in Table 1, the attribute *Prices* from *Order Info* is relevant for predicting *Total Expenses*. Second, the analyst must identify the way to join tables with relevant features, e.g., find the KFK relations among the tables aforementioned. While this task is straightforward in a single database where schema information (i.e., join paths) is available to the user, it becomes more complex when either the number of attributes or the number of tables increases. In either case, it is increasingly complicated for users to understand where and how useful features are stored and what the best practice is to assemble a training dataset. In these scenarios, the burden of identifying and executing joins falls upon the analysts [37].

Consequently, the **Full Table** approach falls in the top left quadrant of Fig. 1. We note that, if the resulting joined table has many irrelevant features, there is an increased chance of overfitting the downstream machine learning model and causing worse quality. For that reason, analysts may apply a feature selection strategy to filter out less relevant features. The resulting approach is **Full Table + Feature Engineering**.

Related work. Because identifying joinable tables and executing the join is a high-effort activity, much work has been done to assist with these tasks. Kumar et al. [27] provide several rules to assist users to determine which joins are profitable for a downstream task. Data-driven rules such as *tuple-ratio* and *risk-of-representation* are simple to understand and implement for users. ARDA [15] uses a ranking method based on random injection to augment the Base Table. Although these approaches ameliorate the data integration problem, they still require the user to provide a set of valid joins, whether manually, or via the use of data discovery systems like Aurum [12], Lazo [13], LSHEnsemble [42], and Auctus [16] which can help with identifying new datasets with their corresponding join paths. In addition, these approaches require the users to handle different join cardinalities, e.g., *1:N* and *N:M*, to guarantee the resulting table maintains a similar distribution of rows as the Base Table. This task becomes harder with the increasing number of tables and with the varied cardinality of the join relationships.

To summarize, even with the assistance from data discovery and join discovery systems, assembling a full table is hard. We include a discovery baseline in Section 6 to provide empirical evidence.

The goal. We want a *keyless* and *pathless* method of assembling relational training data. We propose using a relational embedding that featurizes X to boost the performance of the downstream model without using human effort.

2.3 Embedding Relational Data

One key problem of machine learning is to find representations of input data that incorporate structural information into the machine learning model. Good representations usually come in the form of real-valued vectors that preserve ideal structural information, which depends on the problem of interest.

Text and Graph Embeddings. There are many approaches to transform text and graphs as numerical vectors that can be used as the input of downstream models. The vectors are trained to preserve certain desirable properties of their original subject. In language modeling, the intuition is simple: words that appear under similar language contexts have embeddings that are close to each other in the embedding space. Embedding training approaches like Word2Vec [31], GloVE [32], BERT [17] extract contextual information and produce embeddings that are fed into downstream models in place of original words. Similarly, in graph embedding, vectors are trained to learn a mapping that embeds nodes in a vector space such that geometric relationships of the embeddings reflect structures of the original graph (e.g., degrees, clusterings) [23].

Relational Embeddings. Embedding has also been used in the context of relational data. Bordawekar and Shmueli [7] constructed word embeddings for tokens and enabled queries like semantics matching, predict queries, etc. Termite [19] studied the approaches of learning both structured and unstructured datasets in the same embedding space. Lees et al. [29] developed an embedding-based tool TURL for web table understanding. EmbDI [11] and DeepER [18] approach data integration tasks, such as schema matching and record linkage, with relational embeddings.

An Opportunity. In this paper, we propose a graph construction method that represents relational data. When embedded, the resulting relational embedding is used to featurize the *Base Table* and boost the performance of machine learning models (Section 3). We then incorporate the relational embedding into an end-to-end system, Leva, that helps solve the problem with minimal human intervention (Section 4).

2.4 Problem Statement and Goal

Assume a *Base Table*, T_b with schema (X, Y) and a database with n tables $D = \{T_1, \cdots, T_n\}$. X is a set of features in *Base Table* and Y is the target variable, the goal is to find an embedding mapping $\mathcal{E}: t_i \rightarrow e_i$, from tokens t_i in $D \cup T_b$ to a multidimensional vector e_i. Here, tokens t_i are defined symbolically: it could either be as a specific value, or an abstract token representing a set of relation rows, columns, etc. This embedding presents analysts with an alternative featurization strategy for the training dataset (i.e., an alternative to one-hot encoding). We want to design a system, Leva, that builds $\mathcal{E}$ with high scalability, and robustness to the existence of missing and dirty data in the input relations.

Using the Embedding. With the embedding available, each token in **Base Table** is represented with the corresponding high-dimensional vector in the embedding, and the resulting dataset is used as the input to the ML task. Because ML models are built to

predict out-of-sample data, X', we assume the test data is not part of the input to Leva when the embeddings are calculated. Finally, at inference time, seen test data is replaced with its embedding while unseen data is either handled with binning quantization, or replaced with one-hot encoding. More details on how data are transformed are covered in Section 4.1.

3 RELATIONAL DATA AS GRAPHS

In this section, we explain how to represent relational data as a graph to recover join information (i.e., inclusion dependencies). We discuss how this graph construction process differs from others in the literature in Section 3.3.

High-Level Insight. Leva represents relational data as a graph, which is then embedded into a high-dimensional vector space. The graph may include spurious information, such as inaccurate KFK relationships, and consequently, the embedding will reflect this faulty information as well. Despite that, the supervision signal from the downstream task removes spurious edges and selects those useful for the task at hand. Because the embedding incorporates other useful information from the entire database into the Base table, it boosts the downstream ML task performance.

Preliminaries. Assume a blackbox textification module that converts a collection of input tables into a collection of string tokens. A direct approach to produce an embedding is to train word embeddings directly over the collection of string tokens. This approach loses information captured in the relational model. Recent research has shown [11] that capturing the relational structure in a graph yields better quality in downstream data integration tasks. We propose a new graph construction process that produces an embedding specifically constructed for boosting the performance of downstream machine learning tasks.

Algorithm 1 Graph Construction and Refinement

```
1:  INPUT: Dataset D
2:  tableTexts ← TEXITIFY(D)
3:  edges = {}, # keys are value nodes, values are lists of row nodes
4:  for row ∈ tableTexts do
5:      for col ∈ table.columns do
6:          token ← row[col]
7:          Vote for value token with attribute col
8:          edges[token] ← row
9:      end for
10: end for
11: attr ← Attributes with enough votes for each tokens
12: edges ← REFINE(edges, attr)
13: edges ← ADD_WEIGHT(edges)
14: RETURN edges
```

Graph Construction Overview. The graph is constructed in two stages: node and edge construction (Section 3.1), and refinement step (Section 3.2).

3.1 Graph Construction With Value Nodes

We start with a simplified version of the graph construction process. In the graph, nodes represent rows and edges between i and j are created according to the following similarity metric:

$$M_{i,j} = \begin{cases} 1 & \exists \text{ attributes } attr1 \text{ and } attr2,\ i.attr1 = j.attr2 \\ 0 & otherwise \end{cases}$$

$attr$ indicates an attribute (i.e., column of a table) and i.attr indicates the value of that attribute on that row. Edges represent relationships between rows of different tables, for example, they may represent inclusion dependencies that establish the graph structure. Constructing the graph by computing the above metric for all pairwise row nodes is inefficient. It yields $O(MN^2)$ complexity where N is the total number of rows and M is the total number of attributes across all tables. Furthermore, it constructs a graph with an overly complicated structure. For example, for a set of rows with the same value under one attribute, the above algorithm creates a complete subgraph, handling equivalence relationships independent of each other. A larger number of edges makes deriving the embedding harder in the downstream stage.

In short, we would like to build the graph efficiently and represent similarity metric with as few edges as possible. To achieve that, we introduce a different type of graph node, N_V, called *value nodes* that help reduce the total number of edges (Line 4 - 10 in alg. 1). A node from N_V represents a specific value v and is only connected to a row node r if r has v under one of its attributes. Thus, row nodes that share values under the same attribute are connected via a common value node representing the value. Even under the same attribute, multiple value nodes are used to represent different values that attribute can have. In most datasets, the number of rows N is much greater than the number of attributes M and many values appear multiple times in the dataset. The introduction of the value nodes reduces the total number of edges from $O(MN^2)$ to $O(MN)$. Furthermore, the embeddings for the value nodes also preserve critical information about graph connectivity, corresponding to inclusion dependency relationships, and they can be used as additional features for downstream models. Lastly, the adjacency matrix is more sparse, which enables fast approximation algorithms which we will describe in Section 4.2. We only create value nodes when values are shared between multiple rows. In summary, this achieves a large reduction in the number of edges in exchange for a few value nodes. The number of value nodes itself is small and can be further reduced with techniques such as binning.

3.2 Graph Refinement: Voting and Weighting

Because the similarity metric that determines the creation of an edge uses a purely syntactic criterion they represent inclusion dependencies, but some edges do not reflect true KFK relationships between the rows they connect. For example, *Washington* could appear under *Name* and *State* and creating edges because of this does not contribute to the task of linking related records with each other. There are two common cases where the similarity metric fails: missing values that share a representative token, and different words sharing the same syntactic representation. For a particular value node, we want to know if it represents an informative connection and if so, with which row nodes. Since value nodes are inherently defined by which attributes they correspond to, a natural variant of the second question is to understand which set of attributes a value node should belong to. We propose an *attribute voting mechanism* that helps answer the two questions above (Line 11-12 in alg. 1).

Essentially, we want to create an edge only when we have enough *evidence* that the edge connects two attributes with a real underlying relationship. To quantify the amount of *evidence* we have for one value to belong to a certain attribute, we allow rows to *vote* for the attributes value nodes should fall under. When some row in the database has value v under one of its attributes $attr$, we are gaining evidence for the relationship between v and $attr$ and give out a vote for v being under $attr$. For a particular value node, the more votes one attribute has, the more *evidence* a true relationship between the value and the attribute exists. We use the distribution of attributes to help make that decision.

Missing Values. As mentioned before, missing values exist in the data with different representations so a simple static table lookup does not suffice. Instead, we use the voting mechanism to help detect missing values.

Missing values usually appear under multiple attributes while common data usually only appears under a small subset of them. We identify value nodes that received more than θ_{range} votes for different attributes as missing data. The refinement stage removes such value nodes and their connected edges. The θ_{range} is used as a percentage of all attributes in the database and the default value used in our system is $\theta_{range} = 50\%$. The parameter is set after empirical observation. The intuition is that when tokens appear under more than 50% of all attributes, they do not add much information to the graph, embedding, and hence, to the final ML task.

Words With the Same Syntactic Representation. For each value node, we only want to include edges induced by attributes for which we have high evidence, i.e., number of votes. That is, the value-attribute relations are witnessed at least some number of times in the dataset. Otherwise, it is likely that the edges are created accidentally (i.e., the "Washington" example). We rank all the potential attribute names by the number of votes it receives and remove attributes with votes under threshold θ_{min}. θ_{min} is presented as a percentage of all the votes this value node receives and in our implementation, θ_{min} has a default value of 5%.

Graph Weighting. We want to annotate edges with weights that indicate the degree to which the value-node relation contains useful information (Line 13 in alg. 1). Value nodes that are shared between a large number of row nodes are less likely to indicate useful inclusion dependencies i.e., they do not likely represent a KFK relationship. Conversely, value nodes that are only shared between two row nodes likely convey information that is specific to two row nodes. Leva assigns each row - value edge with a weight that is inversely proportional to the number of edges the value node is connected to. The weighting of the graph has other implications too: it also balances the less-visited nodes by giving them more weights during random walk generation—an effect that we will discuss more in Section 4.2.2. However, weighted graphs might create a high memory footprint as the random walk process relies on alias tables to preprocess transition probabilities. The default of Leva is to use weighted graphs unless the expected memory exceeds the user's resource limits.

3.3 Graph Construction Strategies

Graph representations of relational data are good for ML tasks due to their ability to cluster similar rows across tables and to express rows' relationships with their attribute values. Previous work on relational embeddings either directly embeds text [7, 29], or constructs a graph as we do, but tailored to data integration tasks [11]. We summarize the key differences in graph construction between previous work and Leva.

- *Value Nodes.* Value nodes are used to describe relationships between rows and it greatly reduces the total number of nodes and edges constructed. This not only reduces workloads for downstream embedding training modules, but also directly generates representation for critical values, which can be used as an additional source of input for downstream tasks. We show the positive effect in the evaluation section.
- *Edge Refinement and Weighting.* The edge refinement and edge weighting steps are crucial to embedding construction. Leva refines the graph through node and edge removal and encodes more *important* information through edge weighting. This prevents the graph representation of relational data to be heavily concentrated on information that is duplicated many times.
- *Values As Strings.* Graph construction stage handles heterogeneous relational data by viewing value tokens as strings. This relies on the textification module introduced in Section 4 and extends Leva's ability to make predictions on unseen data.

4 LEVA OVERVIEW

In this section, we embed the graph construction module into the complete pipeline of Leva. Leva's architecture is presented in Fig. 2. It consists of 5 stages: *input and textification* (Section 4.1), *graph construction*, *graph refinement*, *embedding construction* (Sections 4.2 and 4.3), and *embedding deployment* in Section 4.4.

4.1 From Relation To Text

This stage prepares the input relations for the downstream pipeline and has important performance implications. Columns in the input relations are read in a streaming fashion. Leva classifies each column into a set of types and applies a textification strategy:

Keys. Since key information is not always available in the schema, Leva detects key columns under "keyless" scenarios with two simple heuristics: i) the ratio between the number of unique values and the total values in the column is close to one; ii) the attribute is not a floating point number. For i) Leva seeks ratios close to 1 to increase robustness to data errors, such as duplicates. When these two heuristics are satisfied, the attribute is considered a Key. Keys are encoded directly, as opposed to other numerical data, which we bin to reduce the cardinality.

Numerical and Datetime Data. Numerical and datetime data's cardinality is often high. Unlike keys, encoding this data directly would lead to large embeddings and low scalability. Worse, the notion of numerical distance is lost when numbers are encoded directly. To maintain numerical distance and scale to large datasets, we quantize numerical data and encode the data's distribution via histograms. Histograms have a pre-defined number of bins. We use both equi-width and equi-depth histograms based on the data distribution. Concretely, Leva computes the kurtosis of the numerical values and compares it with that of a normal distribution. If the distribution of the data is heavy-tailed, the system chooses equi-depth histograms to include outliers. Otherwise, the system

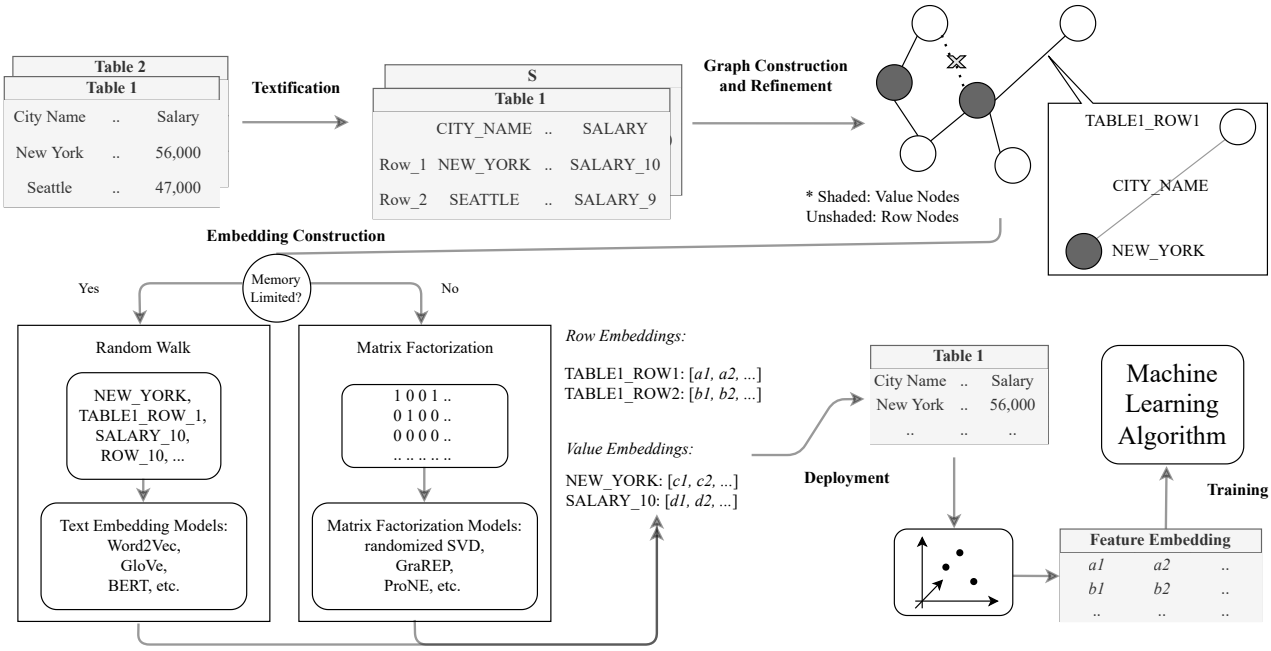


Figure 2: Leva's Architecture Overview

chooses equi-width histograms. Binning the numerical values not only preserves numerical distances between tokens, but also greatly reduces the number of textified tokens the system needs to handle during graph construction. Both numerical and datetime data are represented downstream, instead of with their original values, a string concatenation of their attributes and their *bin ID*.

Missing Data. Without treatment, missing data will negatively impact performance. Missing data has many representations such as "?", "NULL", "N/A", NULL and others. Instead of doing exact comparison with a static list of potential representations, we follow a dynamic removal approach that is explained in Section 3.2.

String Data. String data should either be viewed as an atomic piece of data, or as a formatted list of data (i.e., data separated by comma, a string formatted list). Leva relies on an internal parser to check if strings can be broken down into a list of elements. If not, we directly encode the string as textifed output. Otherwise, we apply similar textification strategies to each individual element and return the list of textified elements as output.

After textification, the graph construction stage explained above produces a graph as the input of the embedding construction.

4.2 Embedding Construction

The embedding construction component accepts different embedding methods in a plug'n'play fashion so it can readily adopt newer approaches as they appear. Nevertheless, we include two default methods in Leva's implementation: matrix factorization (MF) and a random walk-based (RW) method.

Why Two Methods? One of the criteria we set to construct the embedding is scalability. There are two compute resources we care about: memory consumption and runtime. Matrix factorization methods execute faster but the representation of the graph as an adjacency matrix has a higher memory footprint. On the other hand, random walk based methods require a longer time to generate random walks and embeddings. The exact tradeoff is based on the computing resources available. In particular, Leva uses matrix factorization when there is sufficient memory and falls back to the random walk based method when there is not. To determine the memory required ahead of choosing one embedding method or the other, Leva analyzes the graph and uses the number of nodes to estimate the memory consumption.

High-level intuition. Both embedding methods create node and context embeddings to approximate the proximity matrix M.

$$M_{i,j} = \begin{cases} \log(P_{i,j}) - \log(\tau P_{D,j}) & (i,j) \in D \\ 0 & (i,j) \notin D \end{cases}$$

$P(i,j)$ are edge weights assigned to edge i,j and the second term measures negative sampling with sampling ratio τ [31]. MF learns the node and context embeddings by factorizing the proximity matrix, which defines *closeness* between nodes. RW relies on random walks to estimate the *closeness* of nodes and treats preceding and succeeding nodes as contexts for training.

4.2.1 Graph Factorization. Matrix factorization is inspired by dimensionality reduction methods and aims to generate embedding such that the inner product between the learned embedding approximates some proximity measure of the graph. Popular choices of measure include adjacency matrix or probability transition matrix.

Algorithm 1 produces a sparse graph by using value nodes to reduce the total number of edges created. This introduces the possibility of using sparse matrix factorization methods, which are both more compute-efficient and memory-efficient than dense

factorization methods. Traditionally, matrix factorization approximates the proximity matrix M with a Singular Value Decomposition $M \approx U_d \Sigma_d V_d^T$ where Σ_d are first-d singular values and U_d and V_d are orthonormal matrices corresponding to the singular values. The node embedding matrix is given by,

$$\mathcal{E} = U_d \Sigma_d^{1/2}$$

However, the computational complexity for SVD makes factorization methods not applicable to larger graphs and therefore, larger relational datasets. Halko et al. [22] proved that with a good theoretical error bound, it is possible to approximate SVD results through randomized algorithms in $O(d^2N)$ time where d is the target number of singular vectors in M. The algorithm relies on restricting the matrix to a lower-dimensional subspace and computes factorization in the reduced subspace. We construct an orthonormal basis Q of the M's range such that $M \approx QQ^TM$. Then, we compute SVD on the reduced matrix $Q^TM = U\Sigma V^T$. The embedding is generated by transforming the SVD back to the original space,

$$\mathcal{E} = QU\Sigma^{1/2}$$

In fact, we can replace the randomized SVD method above with any state-of-the-art methods on matrix factorization. In recent years, we have seen many approaches for matrix factorization that exploit the parallelism available in modern hardware, as well as new approximation algorithms [10, 21, 30, 35, 41]. In our evaluation section, we present results from using the randomized SVD methods with spectral propagation techniques enhancement from [41].

4.2.2 Random Walk Generation and Balancing. Matrix factorization methods produce embeddings fast at the cost of a high memory footprint. When there is not sufficient memory available in the system, Leva uses a random walk based method that represents the graph as an adjacency list instead of as a matrix.

The general pipeline of random walk methods is as follows. First, a method is used to produce random walks on the graph, which are represented in a textual format. Second, a language modeling technique such as Word2Vec[31], BERT[17], etc. is applied to the corpus, producing the final embedding. Note that, unlike applying a language modeling technique directly to a textual representation of the data, here it is applied to the output of the random walks from the graph. Hence, the structure of the graph is preserved. Our method is oblivious to the specific language technique used, and we use Word2Vec in our implementation.

Balancing Random Walks. Unlike in text, some tokens in relational data appear only a few times. These tokens are visited less often in the random walks and therefore, not represented well in the embedding space. For word embeddings, the impact is small due to the large amount of training data and the fact that some words by nature appear less often than other words. However, in the context of predicting downstream tasks, badly-represented nodes have equal weights in the training data and if not represented well, they could disrupt the training for downstream models. In the graph we construct, since the value nodes are shared between more rows, it is often the case that value nodes are visited more frequently. In an effort to balance out the representation of tokens in the random walks, we propose the following two solutions. One is to find the less represented nodes, and force the system to *restart* walks from those nodes more often. Another one is to set *visit limits* and skip certain nodes when they are visited more times than the limit. Since the most frequently visited nodes are mostly value nodes, we equivalently force the random walks to directly walk on row nodes, which in turn increases the representation of row nodes in the walks that our system generates. We show in the evaluation section that enabling balanced random walks improves the performance of downstream tasks.

4.3 Embedding Method Analysis

Matrix factorization methods and random walk methods have time complexity $O(d^2N)$ and $O(N \log N)$ respectively, where d is the number of desired singular values and N is the number of nodes. The theoretical complexities are only approximations to runtime. In practice, the complexity of the embedding construction depends heavily on the connectivity of the graph, which in turn depends on the properties of the dataset. The choice of algorithm again depends on the dataset and resources available to the user.

To optimize for runtime, we maintain the graph as small as possible. These include minimizing the number of edges by introducing value nodes, minimizing the number of nodes by treating missing data, and representing each row with only one node.

Lastly, we would like to comment more on the space complexity of RW methods on both weighted and unweighted graphs. For a graph with N nodes, weighted random walks use the alias method that requires $O(N \log N)$ time preprocessing to achieve $O(1)$ time for drawing from distribution. Large alias tables pose challenges not only to runtime but also to memory as well. The unweighted version, on the other hand, requires less preprocessing and scales better with large datasets and large graphs.

4.4 Embedding Deployment

We discuss a few important aspects of using the embeddings produced by Leva to featurize training datasets.

Dimensionality. Because the output embedding is high-dimensional it may overfit downstream models that are not regularized [6]. This means that a model that was appropriate before, can overfit if the embedding is employed to featurize the base table. There are two solutions to this problem: (1) reduce the embedding dimensionality (2) apply regularization penalty to the downstream model. The second is interesting because it may further improve the ML model's performance. We explore both options in the evaluation section. Our observation is that as the number of unique tokens grows, larger embedding dimensions are usually required. However, the ideal dimension depends on the graph structure, which cannot be known a priori and varies a lot from dataset to dataset.

Featurizing Training Dataset. There are two options to featurize the base table. The first uses the embeddings for the rows only. The second is to augment embeddings with value node embeddings, which capture connectivity relationships between rows. Using the second method has the potential of improving the downstream performance by including more information in the featurization of the Base Table. We explore both options in the evaluation section.

Hyperparameter Tuning. A goal of Leva is to reduce human effort. Leva includes strategies to choose the configuration parameters. These are summarized in Table 2 with their default values.

Stage	Parameter
Textification	Bin Size (User, Default: 50), Histogram Type (Default: Kurtosis)
Graph Construction Graph Refinement	θ_{range}, θ_{min} (Default: 50%, 5%) Graph Weighting (Default: Yes)
Embedding Construction	Embedding Size (User, Default: 100), Specific Method Parameter(User)
Embedding Deployment	Featurization Method (Default: Row + Value)

Table 2: Leva Parameters Specification

Dataset Name		Genes		Bio		Financial	
Method		RW	MF	RW	MF	RW	MF
Within Entities, Percentile	50%	2.63	1.04	3.71	1.32	3.59	1.30
	90%	3.46	1.48	4.79	2.20	6.42	2.13
Randomly, Percentile	50%	3.76	1.35	4.09	1.40	4.29	1.34
	90%	5.04	2.21	6.32	1.98	6.88	2.05
50% Distance, Ratio	50%	0.69	0.77	0.90	0.94	0.83	0.97

Table 3: Percentile L_1 Distances Between Node Embeddings and Percentage Ratio of Medium Distance

5 EMBEDDING: WHY DOES IT WORK?

The relational embedding boosts the performance of downstream machine learning tasks because when the relational embedding is used to encode the Base table, it brings useful predictive information from other tables. In this section, we look into this claim in more detail. We do so by separating the claim into two hypotheses. The first hypothesis is that whenever there is useful predictive information available, this is represented in the embedding. For the information to be represented in the embedding, it first has to be represented in the graph. Hence, the first hypothesis says that the graph represents useful relationships between the Base table and other tables, including KFK relationships. The second hypothesis is that non-predictive (spurious) information (including spurious inclusion dependency relationships, which will also get represented in the graph), is removed during the training process by using the supervision signal from the downstream task and hence do not harm the downstream performance. In this section, we analyze these two hypotheses in detail.

Before presenting these two hypotheses in more detail, we revisit the problem statement of Section 2.4. Assume the response variable is Y, Base Table is T_b, and the set of all attributes in $D = \{T_1, \cdots, T_n\} \cup T_b$ is X_{all}. For Y, assume we let $\tilde{X}$ to be a set of features (their corresponding KFK relations to be joined with Base Table) from $D \cup T_b$ that are perfect predictors of Y. The two hypotheses we make are:

• **Embedding incorporates information from relations.** During the graph construction stage, row nodes in $D \cup T_b$ that pass the similarity metric used in Section 3.1 are connected in the graph. This means that any information related to the Base table and captured by the similarity metric (including inclusion dependency relationships) will be represented in the graph by means of an edge connecting the entities. During the embedding construction, nodes that are connected via an edge will be represented closer to each other. Therefore, when embeddings in T_b are used, they bring in information from other tables with which they are connected via an inclusion dependency relationship.

• **Training selects predictive information from the embedding.** In addition to edges that represent related entities, some edges will be spurious. We show that the supervision signal from the downstream training process removes non-predictive information. In other words, when the embedding is used for training, it boosts the performance of the task because it brings information from other tables: the positive information remains, while the negative information is filtered out during the training process.

5.1 Information Integration From Embeddings

For each row in T_b, we can find a set of rows in $D \cup T_b$ that describe a similar entity, where the similarity metric is defined in Section 3.1. These rows are connected with an edge during graph construction. Among the relationships between these rows, some incorporate KFK relationships, which bring in attributes that describe the same entity and therefore can be joined to augment T_b.

During Leva's embedding construction, we create value nodes only to serve as an intermediate step between two row nodes. Thus, our constructed embedding directly approximates the similarity metric defined between rows. If the embedding construction method is able to represent rows close to each other in the embedding space, they will bring in additional information for the downstream task, even if we deploy only embeddings from *Base Table* as feature inputs. We demonstrate this is exactly the case.

We set up the microbenchmark experiment to evaluate the *clustering effect*[1] with two control groups: 1) Within Entities Row nodes that according to the ground truth should belong to the same entity. 2) Randomly Row nodes that are randomly selected from the total pool of row nodes. Within each group, we select 5 nodes and compute the median of the pairwise L_1 distance. Then, we generate a distribution of such medium distance for 5000 entities. If the embedding represents related nodes closer to each other the median distance should be lower than with respect to unrelated nodes.

Table 7 shows different quantile (50%, 90%) distances for 3 different datasets and the ratio between two groups for the medium (50%) distance. We observe distances in the Within Entities group are lower than those in the Random group. This holds when using both RW and MF and both the 50% and 90% percentile and also across different datasets. This highlights the ability of the embeddings to represent related rows together; when the Base Table is encoded, it incorporates information from other tables as well.

5.2 Removal of Nonpredictive Feature

In addition to correct edges, the graph also includes spurious edges (i.e., not representing real KFK relationships) that bring in irrelevant features. We show how the learning process filters these out.

If we knew a priori the set of fully predictive features $\tilde{X}$, the embedding generated from $\tilde{X}$ (denoted as $\mathcal{E}_{clean}$) would directly represent these features: value nodes are trained to represent the embedding of the feature and row nodes sharing that value would be clustered accordingly. The question inherently can be transformed into the following: given that $\mathcal{E}_{clean}$ boosts downstream

[1] the ability of the embedding to represent related nodes close to each other

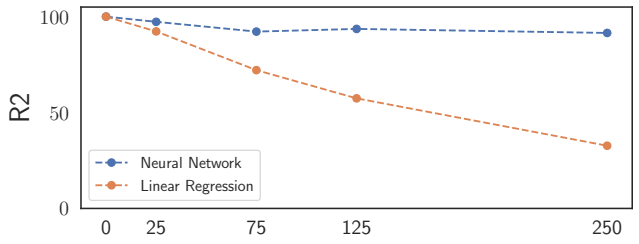


Figure 3: Percentage of Noisy Data v.s. R^2, (Higher is Better)

task performance, is the embedding $\mathcal{E}_{all}$ generated from X similar to $\mathcal{E}_{clean}$? In other words, does the supervision signal help remove non-useful information? The answer is *yes*.

In real datasets, the predictive features are usually only a small portion of all the features. In order to understand the influence of non-predictive attributes on the embedding, we construct a synthetic dataset STUDENT (schema in Table 1). Similar to Section 2, the response variable *Total Expenses* can be fully predicted by *Prices* in *Price Info*. The KFK to join *Total Expenses* and *Total Expenses* itself are considered the set of *clean* attributes to be kept in $\mathcal{E}_{clean}$. To understand the robustness of the embedding with respect to edges added due to non-predictive attributes, we inject to all three tables K new attributes that are generated as white noise numerical values. Using a bin size of 10, the generated values would induce *noisy* edges between row nodes, which are uncorrelated with the response variable. We constructed $\mathcal{E}_{all}$ from the database that contains tables injected with *noisy* edges.

To see if the downstream model is capable of recovering $\mathcal{E}_{clean}$ from $\mathcal{E}_{all}$, we trained a fully connected network and a linear regressor that finds the mapping $\mathcal{M}$ between shared tokens in $\mathcal{E}_{all}$ and $\mathcal{E}_{clean}$. We only trained the embedding on 80% of the shared tokens and used the rest 20% as the test set. Specifically, we minimize for each token t_i in the training set, the mean squared error between vectors $\mathcal{M}(\mathcal{E}_{all}(t_i))$ and $\mathcal{E}_{clean}(t_i)$. Default parameters are used to construct both $\mathcal{E}_{all}$ and $\mathcal{E}_{clean}$. The relationship between R^2 on testing data with respect to the amount of injected *noisy* attributes (as a percentage of clean data) is shown in Fig. 3.

Fig. 3 shows that even when the percentage of noisy attributes increases, the neural network model can still recover the information stored in $\mathcal{E}_{clean}$. This is also true with the linear model, but its degradation is faster than in the case of the neural network. This demonstrates that the supervision signal will help remove spurious information from the embedding.

6 EVALUATION

In this section, we present the evaluation results. We organize the evaluation around the main research questions of the paper:

- **RQ1:** Does relational embedding boost the performance of downstream ML tasks?
- **RQ2:** Does relational embedding improve the performance of downstream tasks compared to other embedding methods?
- **RQ3:** Does Leva scale to large datasets?
- **RQ4:** Are the embedding deployment techniques effective?
- **RQ5:** How sensitive is the embedding to parameters that configure the system?
- **RQ6:** How does Leva perform in other data management tasks?

We start by describing the experimental setup, including metrics, baselines, and datasets (Section 6.1).

6.1 Experimental Setup

Metrics. To measure the performance of downstream models, we use accuracy for classification tasks and mean absolute error (MAE) for regression tasks. We report the best performance after configuring model hyper-parameters using grid search.

Baselines. We compare the results of downstream tasks using our embedding with other baselines: **Base Table** (referred to as Base in the experiments), **Full Table** (Full) and **Full Table + Feature Engineering** (Full + FE). Full and Full + FE require access to correct joins—we use datasets for which this information is available so we make sure we use correct keys and join paths. Finally, to demonstrate that a discovery system does not solve the problem, we include Disc. In this baseline, we use an open-source discovery system [12] to identify and materialize join to the Base table.

In addition, we also show a Max Reported baseline, that indicates the best performance reported in the literature. Some datasets have been long used as benchmarks in the machine learning community and there are bespoke methods to maximize their performance. Unlike these approaches, we do not perform any dataset-specific tasks when constructing embeddings. But to show that embeddings are able to achieve similar accuracy to hand-tuned models, we include an experiment where we fine tune embeddings as well.

Embedding Setup. For each dataset, we train the embedding with default textification strategies, using a weighted graph approach and applying refinement techniques with $\theta_{range} = 50\%$ and $\theta_{min} = 5\%$. Embeddings have a size of 100 and a negative sampling rate 1e-3. Settings are pre-determined and agnostic to downstream models and tasks. For embedding methods, we report results from both random-walk method (RW) and matrix factorization method (MF).

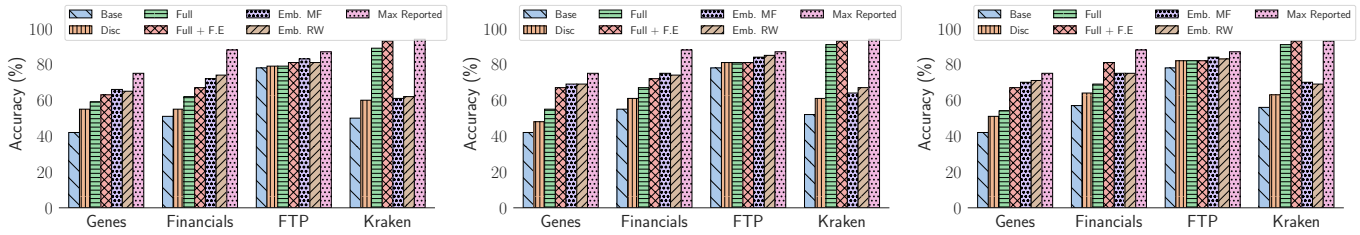
Datasets. We present the dataset characteristics in Table 4. We include datasets both real and synthetic, with and without missing data, with varied numbers of tables and rows, etc.

For classification datasets, Genes [14] predicts localization of proteins and the dataset includes genes' individual characteristics and pairwise interactions as features. Kraken [15] consists of anonymized sensors and usage statistics from the Kraken supercomputers. The response variable is the machine state. FTP [2] predicts a binary gender label from product viewing logs and their corresponding session information. Financial [4] contains successful and unsuccessful loans along with loan and account information and the task is to predict if a new loan will default.

For regression datasets, Bio [5] has atom-level, bond-level information and the task is to predict the molecule's bioactivity. Restbase [1] predicts customers' review information with restaurant information and their geographic locations.

6.2 RQ1: Downstream ML Performance

In this section, we measure the performance of downstream tasks when using both MF and RW approaches and other baselines presented above. We concentrate on classification and regression tasks.

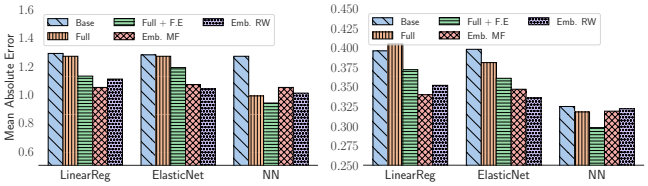


(a) Random Forest (b) Logistic Regression (c) Neural Network

Figure 4: Classification Accuracy (Higher is Better) on Different Datasets and Models

Name	#Tables	#Rows	Task	Missing Data	% String columns
Genes [14]	3	6K	C	Y	93%
Kraken [15]	32	31K	C	N	0%
FTP [2]	2	96K	C	Y	50%
Financial [4]	8	1M	C	N	17%
Restbase [1]	3	19K	R	N	67%
Bio [5]	3	22K	R	Y	69%

Table 4: Classification (C) / Regression (R), Yes (Y) / No (N)



(a) Bio (b) Restbase

Figure 5: Regression MAE (Lower is Better) on Different Datasets and Models

Classification. We evaluate many different baselines, on different datasets, and using different models. The results are presented in Fig. 4, where we show results for 3 different ML models: random forest, logistic regression with ElasticNet regularization, and 2-layer fully connected neural network, with a hidden layer dimension of 64. Hyperparameters are selected via grid search for every dataset. Within each plot, the x axis shows the classification datasets from Table 4, and the y axis shows (after hyper-parameter configuration) the accuracy (i.e., higher is better).

We highlight several aspects of the experiment results. First, Full, Full+FE, and Disc always outperform Base, and Disc never outperforms Full or Full+FE. This demonstrates the value of enriching the Base Table with features from other tables, as previous research has shown, and as practitioners are used to doing today. Second, the relational embeddings outperform Full across models and for 3 datasets, and they always outperform Disc. They stay within 5% performance with respect to Full+FE. The best news is that the embedding was trained in a completely unsupervised fashion, unlike Full, Disc, and Full+FE, which requires careful supervision by a human that needs to know the schema and perform the adequate joins, even using a discovery system (i.e., Disc). Without human effort, the embeddings match the performance of carefully-crafted training datasets even in the Financials and Kraken datasets, which contain several relations. The results above are consistent across models (see Fig. a, b, and c), despite variability across datasets.

The embeddings outperform Full+FE in some cases due to the ability of the embeddings to encode string similarity better than the one-hot encoding used by Full+FE. We include the percentage of string columns in Table 4.

Do embeddings reduce total achievable performance?. The relational embeddings boost the performance of the downstream task without human effort. This makes them attractive to users without domain knowledge or the ability to fine tune complex machine learning models. And such users exist, as evidenced by the growing popularity of AutoML techniques. Despite this benefit of embeddings, we want to understand if they can perform as well as the Max Reported accuracy. In other words, if the objective is to improve the accuracy at any cost in human effort, we want to verify whether this is possible to accomplish using embeddings.

Fig. 6a shows the results of the experiment using 3 of the previous datasets. We show the Max Reported, the Emb MF and the Emb RW accuracy again for reference. This time, we also perform manual fine tuning on both embedding methods, Emb MW Fine Tuned and Emb RW Fine Tuned. The fine tuning includes using domain knowledge to drop tables from the database when they do not include relevant information for the task at hand and doing a more exhaustive grid search over the model parameters when training the model. Fig. 6a shows fine tuning improves the quality of the embeddings, bringing them close to the Max Reported accuracy. Given the evidence, we believe the small gap between Max Reported and the embedding is due to the amount of domain-specific knowledge we successfully incorporated in the training process, and not to some fundamental limitation of the embedding approach.

Regression. We structure the experiments for regression tasks similar to the classification ones. In this case, for space efficiency, we present different models within each plot, in the x-axis, and we include a plot per dataset. We use linear regression, ElasticNet, and a 2-layer fully-connected neural network, and the two regression datasets from Table 5. We measure performance with MAE.

Similar to the classification results, we observe the Full and Full + FE outperform Base across models and datasets. Additionally, we observe that the embedding method outperforms Base in all datasets and even Full and Full + FE when using linear regression and ElasticNet. Concretely, the average reduction for MAE is around 10% to 20% compared with Base and 5% to 10% compared with Full. Both

regression datasets contain a high percentage of string columns (see Table 4) which explains why the embeddings outperform even Full+FE, except in the case of neural networks. We attribute the poor performance of neural networks here to the relatively smaller size of the datasets. Using the neural network achieves the best results in this experiment for all baselines and methods, and the embedding performance is close to Full and slightly worse than Full+FE, but without any human effort. Finally, there is no obvious difference between MF and RW. Their real difference is related to their scalability, which we measure later in this section.

Summary. Relational embeddings boost the performance of downstream ML tasks with respect to Base. They match the performance of Full and Full+FE despite not requiring human involvement. This resolves the tradeoff between human effort and performance with a method that does not require human involvement but produces comparable results with Full and Full+FE.

6.3 RQ2: Comparing Embedding Methods

The results of the previous section show embeddings boost the performance of downstream machine learning tasks. In this section, we demonstrate that not any embedding method boosts performance and that the way in which we construct our graph is necessary to do so. For that, we compare the relational embedding as created by Leva with several other methods to create embeddings, including other state-of-the-art systems. We report the results in Table 5.

Emb. Method	Genes	Financial	FTP
Word2Vec [31]	55	63	79
Node2Vec [20]	61	69	81
EmbDI [11]	63	67	81
DeepER [18]	70	73	82
Emb. MF	72	**76**	**84**
Emb. RW	**73**	74	83
Max Reported	76 [3]	86 [33]	87 [2]

Table 5: Classification Accuracy with Different Embedding Training Methods

The table includes results for Word2Vec, Node2Vec, EmbDI [11], DeepER [18], and for three different classification datasets. Word2Vec directly textifies relational datasets row by row into a text corpus that is trained to produce embeddings. Node2Vec builds a graph directly based on syntactic relationships without additional refinement and weighting. EmbDI[11] constructs a graph by linking cell nodes with their corresponding rows and columns. DeepER [18] looks up embedding through some pre-trained dictionary and advocates for a *retrofitting* mechanism for unseen words: creating a graph with words as vertices and edges connecting vertices if words co-occur in some tuple. We first observe that all graph-based methods outperform the sequential baseline significantly. Among graph-based methods, EmbDI performs similarly to Node2Vec and DeepER outperforms EmbDI on this task. Both Leva's embedding methods, MF and RW, outperform all other baselines by 3-10 points the performance across the 3 datasets. This highlights that the specific way of constructing and refining the graph helps to better describe relationships between entities.

6.4 RQ3: Scalability

The relational embedding is more useful when schemas are complex, data volumes large, and as a consequence, humans cannot join them manually. Hence, scalability is a crucial property of a practical system. We first characterize the performance of the different embedding methods.

Performance Profile. For both RW and MF methods, We measure the relative time in each of the pipeline stages (see Fig. 2). The results are shown in Fig. 6 with runtime on the left of the plot.

The results of Fig. 6b and Fig. 6c show that the performance bottlenecks are the embedding training stages: walk generation and training when using the graph-based embedding method, and matrix factorization stage when using that method. In particular, textification and graph generation stages are negligible.

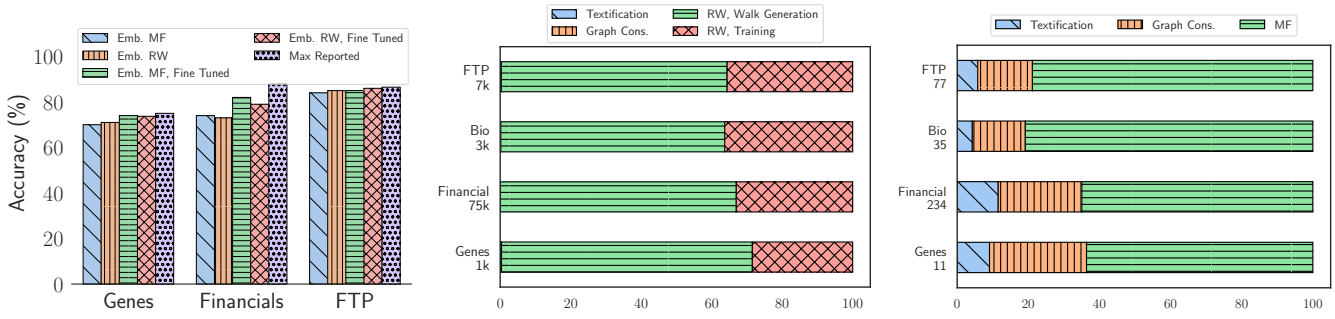
Scalability. To understand the growth of complexity, we build a synthetic dataset with 3 tables, 2000 rows and 5 columns that contains 4000 unique tokens in total. We control the growth of the data by controlling a replication factor, K. At each replication factor, we replicate the dataset K times and each time tokens are suffixed with a version number from 1 to K. With this design of experiment, both the number of rows and the number of distinct tokens grow as a linear function of K. We compare the runtime and memory footprint required to build the embedding. We report results for EmbDI, Leva using RW and Leva using MF. Fig. 7a shows the total runtime of each method as the replication factor, K grows. The runtime behavior shows that random walk based methods, including EmbDI and Leva with RW are an order of magnitude slower than Leva with MF. For example, when the duplicate factor is 100, MF only takes around 3 minutes to complete while RW takes around 18 minutes. For memory consumption, RW consumes around 50% less memory compared to MF across duplicate factors.

6.5 RQ4: Deployment Strategies

6.5.1 Inputting Embedding to Downstream Tasks. We explore the effects of different featurization strategies as explained in Section 4.4. For a row in the *Base Table*, one method of featurizing is to directly feed the row embedding vectors as features for ML task (Row only); Another method is to concatenate row embeddings with value node embeddings that share edges (Row + Value).

We note that the regularization methods employed on the ML model influence the choice of deployment strategies of the embedding. In Table 6, we use the results from Row as a baseline and include results for Row + Value, with and without regularization. The regularization methods are requiring a minimum number of nodes per leaf node, penalizing with a Lasso penalty and adding a dropout layer. We observe that in all cases, Row + Value and regularization outperforms themselves without regularization. In most of the cases, Row + Value outperforms Row, indicating the potential boost of performance by adding value-specific information.

6.5.2 Dimension Reduction and Performance. Embedding outputs are stored as key-value pairs, where keys are string tokens that represent the node and values are floating-point embedding vectors. The cost of storage mainly comes from storing the high-dimensional vectors. We consider the effect of dimension reduction on trained embeddings. In Table 7, the entry on row i and column j shows the



(a) Fine Tuning Results **(b) Perf. Profile: Random Walk Approach** **(c) Perf Profile: Matrix Factorization Approach**

Figure 6: Fine Tuning Results and Performance Profiles

Name	Row+Value No Regularization	Row+Value Regularization
Genes, RF	-3.03	-0.88
Genes, LR	+0.46	+2.97
Genes, NN	-0.14	+1.94
FTP, RF	-0.23	+0.17
FTP, LR	+0.39	+1.05
FTP, NN	+0.07	+3.39

Table 6: Ablation: Deployment Strategy

accuracy after projecting the embedding with original dimension i into dimension j for Genes with standard PCA techniques.

Original \ Reduced	5	25	50	100	200
5	57				
25	55	63			
50	56	62	72		
100	52	61	68	**74**	
200	51	59	63	66	68

Table 7: Accuracy Performance (Genes) with Different Embedding Sizes Before and After PCA

The results in Table 7 demonstrate that larger embedding sizes do not necessarily lead to higher performance, as evidenced by the different performance when the size is 100 and 200. More importantly, smaller embedding sizes (e.g., 50), achieve performance that already outperforms other baselines. This suggests that even if the memory footprint becomes a problem, smaller-sized embeddings yield benefits. Apart from the embedding with size 200, we observe that projecting embedding to smaller dimensional spaces only decreases performance by a moderate amount. Therefore, when storage is limited, users can avoid re-training the embedding by projecting it to a smaller dimension without a significant quality loss.

6.6 RQ5: Ablation Experiments

6.6.1 Numerical Value Binning. In Section 4.1, we discussed the importance of preserving numerical properties of the input data. Binning numerical values not only reduces the total number of value nodes created, but also preserves numerical proximity and ordering. However, creating too few bins would lead to information loss as all numerical values would be under the same bin; creating too many bins would make it more likely for some bins to only have one value. Since we create edges based on shared syntactic relationships, no edges would be created for bins with single values. This piece of information is therefore lost with too many bins.

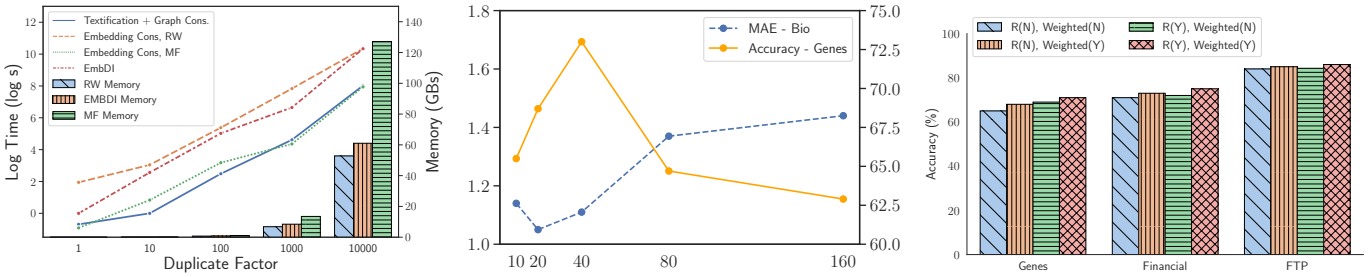
As in Fig. 7b, we present the ablation experiments to understand the influence of number of bins on downstream models. We note that the orange line corresponds to accuracy on the Genes dataset and the blue line corresponds to MAE on the Bio dataset. The plot is made by constructing and evaluating embeddings with bin number 10, 20, 40, 80 and 160. When the number of bins is small, increasing boosts performance on both the classification and regression task. When bin number surpasses 80, we observe decreasing performance, which indicates that overly binning numerical data removes information that could have been useful to the ML task.

6.6.2 Weighted and Unweighted Graph. We present the tradeoff between running time and performance in Section 4. To understand the effect of graph weights on downstream model performance, we construct for dataset Genes, Financial and FTP an unweighted graph, and a weighted graph, where edges connecting value nodes and row nodes have weights inversely proportional to the number of edges value nodes have. The weights are then normalized per node. In 7c, we observe that throughout the three datasets, we observe a 1% to 3% performance boost in accuracy.

6.6.3 Restart walks. To better represent badly-represented nodes in Section 4.2.2, we modify the random walk generation to force restart from badly-represented nodes. Fig. 7c shows results with and without restart walks, using random walk models on both the Genes, Financial FTP dataset. For RW without restart, the system starts from each node in the graph and generates random walks of length 80. This process is repeated 10 times to generate the final corpus; for RW with restart, the system only repeats the process 6 times and replaces the rest 4 iterations with the same number of walks but only from the worst-represented nodes. We observe that Fig. 7c the performances for two of the three datasets are boosted by around 3% and the rest FTP dataset is boosted by 0.3%.

6.7 RQ6: Leva on Entity Resolution

Leva is designed to boost the performance of machine learning tasks. Despite its original goal, its core technical contribution is



(a) Runtime and Memory v.s. Duplicated Factor **(b) Ablation: Number of Bins** **(c) Ablation: Graph Weight and Restart Walks**

Figure 7: Scalability Experiment w.r.t Duplicate Factor and Ablation Experiments

a relational embedding that resembles the one built by other approaches in the literature that target entity resolution tasks. In this section, we ask what Leva's performance is on entity resolution. We compare against EmbDI and DeepER, two methods that build embeddings for this task. We use three datasets from the EmbDI [11] paper. The results are shown in Table 8.

Leva performs better than EmbDI-S and DeepER, two baselines that do not assume any pre-processing to the input data. And it outperforms EmbDI-F in 1/3 datasets, where EmbDI-F models split tokens individually.

Name	EmbDI-S	EmbDI-F	DeepER	Leva
BeerAdvo-RateBeer	0.50	0.82	0.58	0.75
Walmart-Amazon	0.59	0.75	0.63	0.67
Amazon-Google	0.14	0.57	0.62	0.59

Table 8: Entity Resolution Experiments, F1 Score

Leva performs well in a task for which it was not originally designed. This suggests the relational embedding may have uses beyond those presented in the paper.

7 RELATED WORK

Dataset Augmentation. Both ARDA [15], Kumar et al. [27] study how to augment the base table to improve downstream model performance. As discussed earlier, these approaches rely on a set of proposed potential joins and on the user's prior knowledge of schema, an assumption that we relax in our work. Auctus [16] is proposing a system for automatic augmentation of training datasets, including the discovery of web resources. We believe Leva's embeddings can complement some aspects of Auctus's system.

Data and Join Discovery. Data Discovery systems such as Aurum [12], and libraries such as LSHEnsemble [42] help users to identify new datasets, including join paths. [34] proposes DLearn that learns relational models from inconsistent and dirty data with the help of database constraints. These approaches are orthogonal and potentially complementary to Leva. They may help users identify join paths, but they do not focus on augmenting training datasets in an unsupervised manner.

Language and Graph Embedding Methods. Language Embedding models such as Word2Vec [31], Node2Vec [20], GloVe [32], and recently-developed transformer-based ones, such as BERT [17] are largely orthogonal to Leva's contributions. Similar to language embeddings, we have seen many works on efficiently computing graph embeddings through matrix factorization techniques [22, 30, 41] and graph embedding methods, i.e., GCN [25], GAT [40]. GCN motivates a variant of convoluted neural network that uses a first-order approximation of spectral graph convolutions. GAT replaces expensive matrix multiplications with self-attentional layers. Newer language and graph embedding techniques, as well as faster matrix factorization methods, can be easily incorporated in the pipeline in Section 4. Leva directly benefits from development in these areas and their contributions are orthogonal to Leva's as well.

Knowledge Base Embedding. There is extensive work in embedding knowledge graphs (bases) [24]. RESCAL [26] models triples from a knowledge base via the pairwise interactions of latent features. Similarly, structured embeddings and subsequent work [8, 9, 28, 39] learn embeddings for each relation from the triples. These approaches focus on learning latent variables that describe the triples, to later fill in values of an incomplete knowledge base. These approaches can be used to embed the graph Leva constructed.

8 CONCLUSIONS

This paper presents Leva, a system that creates relational embedding for downstream machine learning models. Leva reconstructs join information and recover predictive features from the embeddings during downstream model training time. It consists of an *input and textification* stage, that converts heterogeneous data into textified inputs, a *graph construction and refinement* stage that builds and refines the graph built upon syntactic relationship, an *embedding construction* stage that converts graphs into embeddings through matrix factorization or random walks, and an *embedding deployment* stage where embeddings are featurized into downstream model. We show that Leva is able to preserve information distributed across tables and resembles, if not outperforms, many of the manually joined methods.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their feedback, authors of EmbDI for kindly sharing their code and offering their help, and other members of ChiData for valuable discussions. We would also like to thank Chameleon Cloud for providing computing resources.

REFERENCES

[1] [n.d.]. Home. https://sourceforge.net/projects/proper/files/Datasets/raw/

[2] [n.d.]. pakdd'15 data mining competition: gender prediction based on e-commerce data. http://challenge.mimuw.edu.pl/contest/info.php?id=107

[3] Anna Atramentov, Hector Leiva, and Vasant Honavar. 2003. A multi-relational decision tree learning algorithm–implementation and experiments. In *International Conference on Inductive Logic Programming*. Springer, 38–56.

[4] Petr Berka. 1999. Workshop notes on Discovery Challenge PKDD'99. http://lisp.vse.cz/pkdd99/

[5] Hendrik Blockeel, Sašo Džeroski, Boris Kompare, Stefan Kramer, and Bernhard Pfahringer. 2004. Experiments In Predicting Biodegradability. *Applied Artificial Intelligence* 18, 2 (2004), 157–181. https://doi.org/10.1.1.2.3797

[6] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. 1989. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM (JACM)* 36, 4 (1989), 929–965.

[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*. ACM, Chicago IL USA, 1–4. https://doi.org/10.1145/3076246.3076251

[8] Antoine Bordes, Nicolas Usunier, et al. 2013. Translating Embeddings for Modeling Multi-relational Data. In *NIPS*.

[9] Antoine Bordes, Jason Weston, et al. 2011. Learning Structured Embeddings of Knowledge Bases.. In *AAAI*.

[10] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2015. GraRep: Learning Graph Representations with Global Structural Information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management* (Melbourne, Australia) *(CIKM '15)*. Association for Computing Machinery, New York, NY, USA, 891–900. https://doi.org/10.1145/2806416.2806512

[11] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, Portland OR USA, 1335–1349. https://doi.org/10.1145/3318464.3389742

[12] R. Castro Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. 2018. Aurum: A Data Discovery System. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1001–1012. https://doi.org/10.1109/ICDE.2018.00094

[13] Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. 2019. Lazo: A Cardinality-Based Method for Coupled Estimation of Jaccard Similarity and Containment. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1190–1201. https://doi.org/10.1109/ICDE.2019.00109

[14] Jie Cheng, Christos Hatzis, Hisashi Hayashi, Mark-André Krogel, Shinichi Morishita, David Page, and Jun Sese. 2002. KDD Cup 2001 report. , 47 pages. https://doi.org/10.1145/507515.507523

[15] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. 2020. ARDA: automatic relational data augmentation for machine learning. *Proceedings of the VLDB Endowment* 13, 9 (May 2020), 1373–1387. https://doi.org/10.14778/3397230.3397235

[16] Fernando Chirigati, Rémi Rampin, Aécio Santos, Aline Bessa, and Juliana Freire. 2021. Auctus: A Dataset Search Engine for Data Augmentation. arXiv:2102.05716 [cs.IR]

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[18] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed representations of tuples for entity resolution. *Proceedings of the VLDB Endowment* 11, 11 (Jul 2018), 1454–1467. https://doi.org/10.14778/3236187.3236198

[19] Raul Castro Fernandez and Samuel Madden. 2019. Termite: a system for tunneling through heterogeneous data. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management - aiDM '19*. ACM Press, Amsterdam, Netherlands, 1–8. https://doi.org/10.1145/3329859.3329877

[20] 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*. ACM, San Francisco California USA, 855–864. https://doi.org/10.1145/2939672.2939754

[21] Anshul Gupta, George Karypis, and Vipin Kumar. 1997. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed systems* 8, 5 (1997), 502–520.

[22] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. 2010. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. arXiv:0909.4061 [math.NA]

[23] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584* (2017).

[24] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S. Yu. 2021. A Survey on Knowledge Graphs: Representation, Acquisition, and Applications. *IEEE Transactions on Neural Networks and Learning Systems* (2021), 1–21. https://doi.org/10.1109/TNNLS.2021.3070843

[25] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[26] Denis Krompaß, Maximilian Nickel, et al. 2013. Non-negative tensor factorization with rescal. In *ECML workshop*.

[27] Arun Kumar, Jeffrey Naughton, Jignesh M. Patel, and Xiaojin Zhu. 2016. To Join or Not to Join?: Thinking Twice about Joins before Feature Selection. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, San Francisco California USA, 19–34. https://doi.org/10.1145/2882903.2882952

[28] Timothée Lacroix, Nicolas Usunier, and Guillaume Obozinski. 2018. Canonical tensor decomposition for knowledge base completion. In *International Conference on Machine Learning*. PMLR, 2863–2872.

[29] Alyssa Whitlock Lees, Cong Yu, Huan Sun, Will Wu, and Xiang Deng. 2020. TURL: Table Understanding through Representation Learning.

[30] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *Proceedings of the 2nd SysML Conference*. Palo Alto, CA, USA.

[31] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.), Vol. 26. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

[32] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[33] Matic Perovšek, Anže Vavpetič, Janez Kranjc, Bojan Cestnik, and Nada Lavrač. 2015. Wordification: Propositionalization by unfolding relational data into bags of words. *Expert Systems with Applications* 42, 17-18 (2015), 6442–6456.

[34] Jose Picado, John Davis, Arash Termehchy, and Ga Young Lee. 2020. Learning Over Dirty Data Without Cleaning. *CoRR* abs/2004.02308 (2020). arXiv:2004.02308 https://arxiv.org/abs/2004.02308

[35] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Chi Wang, Kuansan Wang, and Jie Tang. 2019. Netsmf: Large-scale network embedding as sparse matrix factorization. In *The World Wide Web Conference*. 1509–1520.

[36] Ram Sagar. 2021. Big Data To Good Data: Andrew Ng Urges ML Community To Be More Data-Centric And Less Model-Centric. https://analyticsindiamag.com/big-data-to-good-data-andrew-ng-urges-ml-community-to-be-more-data-centric-and-less-model-centric/

[37] Aécio Santos, Aline Bessa, Fernando Chirigati, Christopher Musco, and Juliana Freire. 2021. Correlation Sketches for Approximate Join-Correlation Queries. *arXiv preprint arXiv:2104.03353* (2021).

[38] 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).

[39] Richard Socher, Danqi Chen, et al. 2013. Reasoning with Neural Tensor Networks for Knowledge Base Completion. In *NIPS*.

[40] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[41] Jie Zhang, Yuxiao Dong, Yan Wang, Jie Tang, and Ming Ding. 2019. ProNE: Fast and Scalable Network Representation Learning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 4278–4284. https://doi.org/10.24963/ijcai.2019/594

[42] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet Scale Domain Search. *CoRR* abs/1603.07410 (2016). arXiv:1603.07410 http://arxiv.org/abs/1603.07410