DeepER – Deep Entity Resolution

Muhammad Ebraheem Saravanan Thirumuruganathan Shafiq Joty† Mourad Ouzzani Nan Tang
QCRI, HBKU, Qatar NTU, Singapore†
{mhasan, sthirumuruganathan, mouzzani, ntang}@hbku.edu.qa, srjoty@ntu.edu.sg

***Abstract*—Entity Resolution (ER) is a fundamental problem with many applications. Machine learning (ML)-based and rule-based approaches have been widely studied for decades, with many efforts being geared towards which features/attributes to select, which similarity functions to employ, and which blocking function to use – complicating the deployment of an ER system as a turn-key system. In this paper, we present DEEPER, a turn-key ER system powered by deep learning (DL) techniques. The central idea is that distributed representations and representation learning from DL can alleviate the above human efforts for tuning existing ER systems. DEEPER makes several notable contributions: encoding a tuple as a distributed representation of attribute values, building classifiers using these representations and a semantic aware blocking based on LSH, and learning and tuning the distributed representations for ER. We evaluate our algorithms on multiple benchmark datasets and achieve competitive results while requiring minimal interaction with experts.**

I. INTRODUCTION

Entity Resolution (ER) (*a.k.a.* deduplication and record linkage) is a fundamental problem in data integration and data cleaning. Machine learning (ML)- and rule-based approaches have been widely studied for decades, whereby ML-based approaches deliver better performance, and rule-based methods are preferred when interpretability, maintenance, and explicit definition of domain knowledge are needed [1].

Despite all the work in ER, most existing systems need heavy tuning of features, similarity functions, and blocking functions, among other things – these highlight the emerging need of a turn-key ER system that can perform ER with minimal interaction with experts [2], [3], [4].

A. Desiderata for a Turn-Key ER System

▷ **Minimal Feature Engineering:** Traditional ML- or rule-based approaches often rely on handcrafted features, similarity functions for every attribute and their corresponding thresholds. Good feature engineering is labor intensive, time consuming, domain specific or even dataset specific [5].

▷ **Semantic Similarity Measures:** Choosing a symbolic similarity measure is typically not sufficient. For example, the words "NIPS" and "ICDE" have the same edit distance with "VLDB"; while we know that "ICDE" is closer to "VLDB". This is an important issue as many real-world datasets often contain semistructured or unstructured data (*e.g.,* product description), where simple similarity metrics are not adequate.

▷ **Automated and Customizable Blocking:** Blocking is a key technique used by ER systems to avoid comparing all possible pairs of tuples. Most of the prior work (including those on approximate blocking - see Section VII) do not use semantic similarity for blocking. Further, traditional blocking functions are concise and often utilize a small number (typically 1-3) of attributes for blocking. However, for effective blocking, it might be necessary to take a holistic view of the entire tuple.

▷ **Robustness to Expert Mistakes:** Most ER systems are especially sensitive to expert mistakes in a number of scenarios such as labels for training data, feature engineering, selection of similarity functions/thresholds, and selection of blocking functions. Sub-optimal choices in any of them often incurs a major performance penalty.

▷ **Minimal Portability Effort:** In a traditional ER system, one has to repeat the entire workflow process of feature engineering, blocking function selection, labeling training data and so on for each dataset. This often leads to a significant cost in porting an ER system developed for one dataset to another. An ideal turn-key system should only require the labels for the training data of the new dataset.

▷ **Less Training Data:** Both training an ER ML model or learning ER rules require a reasonably large training dataset. For example, [5] shows that the best performing ML system can require hundreds of labeled tuple pairs. In many cases, training dataset preparation is often the most expensive part of ER as it has to be done manually by experts.

▷ **Simple Architecture:** Finally, the ER system must have a simple and intuitive architecture. A complex architecture often requires more parameters to tune and reduces the overall turn-key performance.

B. DEEPER: A Turn-key ER System using Deep Learning

Deep Learning (DL) is an emerging paradigm that has achieved great performance in many applications such as image recognition, language translation and speech recognition. We introduce distributed representations, a fundamental concept in DL, that can be used to meet all the requirements for a turn-key ER system. Informally, distributed representations (DRs), such as word2vec [6] or GloVe [7], maps an entity (such as a word) to a high dimensional dense vector with real numbers such that the vectors for similar words are closer to each other in their semantic space. For example, the distributed representation for 'VLDB" and "ICDE" will be closer than the vectors for "VLDB" and "NIPS".

Contributions. While distributed representation is conventional wisdom in many fields such as NLP, it has not been explored in databases. In this paper, we make a series of

contributions. We propose a number of different mechanisms that represents tuples as high dimensional vectors such that (semantically) similar tuples have a high (cosine) similarity. Using pre-trained DR dictionaries, we make this conversion process automatic and thereby obviating the need for manual feature engineering. The pre-trained distributed representations are often trained on very large corpus from the Internet – *e.g.,* GloVe is trained on a corpus with 840 billion tokens – they already encode many general information such as the fact that "ICDE" and "International Conference on Data Engineering" are similar. This pre-training has the positive side-effect of requiring much less training data for performing ER on general datasets such as products, citations, and restaurants.

While a more complex DL model can eke out a small improvement, this is achieved at the cost of increased training or tuning. DEEPER uses a simple DL-based architecture that leverages distributed representations to pre-set most of the model parameters. For most of datasets (including all of ER benchmark datasets), DEEPER requires substantially less training data than state-of-the-art ML methods while still being competitive.

We propose a DL based classifier using DRs that is competitive or surpasses prior state-of-the-art methods. Both the representations and the similarity functions can be considered as trainable parameters of the DL model thereby avoiding the cornucopia of domain specific similarity functions and thresholds. DRs open up a number of new opportunities for blocking. We propose a series of LSH-based algorithms that can perform automated, semantic and customizable blocking based on the entire tuple (as against 1-3 attributes of traditional blocking) without any need for blocking functions from domain experts. By leveraging the extensive theoretical analysis of LSH, we can even provide various ER specific guarantees.

C. Summary of Experiments

We built DEEPER, a practical DL based ER system that we plan to open source. We conducted extensive experiments over 4 benchmark datasets that are widely used in prior work covering diverse domains such as citations and e-commerce. They also span a wide spectrum of data characteristics and difficulty levels. In a nutshell, DEEPER outperforms published state-of-the-art ML, non-ML and crowd approaches while requiring less data than other ML based approaches. DEEPER achieves impressive results even when we do not fine-tune the model for every dataset. In other words, we built a model that is an off-the-shelf turn-key solution that nevertheless provided competitive results to ML models that were tuned for that dataset. In contrast to prior work, our method is also robust to null values, noise and even incorrect labels. Even under an extreme scenario where 10% of the labels are incorrect, DEEPER still achieves respectable results.

The rest of the paper is organized as follows. In Section II, we overview DEEPER. In Section III, we describe an effective classifier for ER by leveraging DRs and feature composition. We describe our LSH-based blocking of DRs in Section IV. In Section V, we show how to tune the representations for the ER task. Section VI presents our experiments. We present related work and final remarks in Sections VII and VIII, respectively.

II. DEEPER OVERVIEW

One of the key contributions of our paper is an elegant synthesis of ideas from the theory of Entity Resolution (ER) and Deep Learning (DL). In this section, we give an overview of the main techniques and how they all fit together.

Let $T = \{t_1, \ldots, t_n\}$ be a relational table with n tuples and m attributes $A_1, \ldots, A_m$. We denote the value of attribute A_j of tuple t_i as $t_i[A_j]$. The problem of *entity resolution* (ER) is, given all distinct tuple pairs (t_i, t_j) from T where $t_i \neq t_j$, determine which pairs of tuples refer to the same real-world entities (*a.k.a.* a *match*).

Entity Resolution as a Classification Problem. Most machine learning (ML) based approaches treat ER as a binary classification problem [8], [4], [3], [9]. Given a pair of tuples (t_i, t_j), the classifier outputs *true* (resp. *false*) to indicate that t_i and t_j match (resp. mismatch). The Fellegi-Sunter model [8] is a formal framework for probabilistic ER and most prior ML works are simple variants of this approach. Intuitively, given two tuples t_i and t_j, we compute a vector of similarity scores between aligned attributes, based on predefined similarity functions. The vectors of known matched (resp. mismatched) tuple pairs – that are also referred to as positive (resp. negative) examples – are used to train a binary classifier (*e.g.,* SVMs, decision trees, or random forests). The trained binary classifier can then be used to predict for any arbitrary tuple pair.

Distributed Representations (DRs) for Tuples. A *distributed representation* of words (*a.k.a. word embeddings*) maps individual words to a dense high dimensional vector (say with 300 dimensions) such that semantically related concepts (such as "VLDB" and "ICDE") are close to each other in the vector space [6]. DEEPER transforms each tuple into a high dimensional vector such that similar tuples have high similarity.

Our initial approach converts each attribute of the tuple into its corresponding distributed representation by averaging the word vectors obtained from Glove [7]. The similarity between two values for an attribute can be computed as the distance between their corresponding vector representations, for example, using the cosine similarity metric. The similarity vector is a m-dimensional vector where the i-th entry is the similarity score for attribute A_i for the given pair of tuples. A classifier is then trained based on the similarity vectors for known matched and mismatched tuple pairs.

An alternate approach computes a representation for each attribute through sophisticated compositional methods in DL such as LSTM-based recurrent neural networks. Given composed vectors for a pair of tuples, we compute their *distributional similarity* through subtracting (vector difference) or multiplying (Hadamard product) the corresponding entries. The ER classifier is then trained based on the distributional similarity vector for the training examples.

Tuning Representations for Entity Resolution. Almost all of the benchmark datasets for ER are in the English language

Algorithm 1 DEEPER Approach

1: Convert each tuple into a compositional distributed representation
2: Compute a similarity vector for each pair of tuples in the training set
3: Train a classifier on the similarity vectors
4: Perform blocking for all tuples in the dataset
5: Run the classifier over all tuple pairs within each block
6: Return all matched tuple pairs

and are general purpose domains such as matching products and publications. Hence, we can use any of the popular pre-trained representations such as GloVe [7], word2vec [6] (or fastText [10] for multilingual datasets). However, these representations are generic in the sense that they are not "tuned" for the specific ER task. We propose an end-to-end mechanism to learn the appropriate representation for a particular dataset.

Blocking for Distributed Representations. Blocking is a key technique for improving the efficiency of ER by grouping tuples into blocks. There has been extensive work on designing appropriate blocking functions that operate in the symbolic realm. We propose two efficient and effective blocking algorithms based on the distributional representation for tuples that also take semantic relatedness into account.

Overall Approach. Algorithm 1 shows the pseudocode for our overall approach. We describe how each of the steps is instantiated in the remainder of the paper.

III. Entity Resolution with Deep Learning

In this section, we describe how to design an effective classifier for ER by leveraging the idea of distributed representation and feature composition in the DL framework.

A. Distributed Representations for Words

We start by providing an abbreviated and simplistic description for distributed representation of words (please refer to [11] for more details).

Distributed representation of words is an embedding method that seeks to map each word in a given vocabulary into a high dimensional vector (*e.g.,* 300 dimensions). In other words, each word is represented as a distribution of weights (positive or negative) across these dimensions. Often, many of these dimensions can be independently varied. The representation is considered "distributed" since each word is represented by setting appropriate weights over multiple dimensions while each dimension of the vector contributes to the representation of many words. Distributed representations can express an exponential number of "concepts" due to the ability to compose the activation of many dimensions [11]. In contrast, the symbolic (*a.k.a.* discrete) representation often leads to data sparsity and requires substantially more data to train ML models successfully [12].

A number of methods have been proposed to compute the distributed representation of words including word2vec [6], GloVe [7], and fastText [10]. Generally speaking, these approaches attempt to capture the semantics of a word by considering its relations with neighboring words in its context. In this paper, we use GloVe, which is based on a key observation that the ratios of co-occurrence probabilities for a pair of words has some potential to encode a notion of its meaning. GloVe formalizes this observation as a log-bilinear model with a weighted least-squares objective function. Informally, GloVe is trained on a global word-word co-occurrence matrix and seeks to encode general semantic relationships as vector offsets in a high dimensional space. This objective function has a number of appealing properties such as the vector difference between the representations for (man, woman), (king, queen), (brother, sister) are roughly equal.

B. Distributed Representations for Tuples

In this paper, we advocate for distributed representations where each tuple is represented as a dense high dimensional vector of real numbers. Additionally, tuples that are similar (based on expert defined similarity functions) must have vector representations that have high Cosine similarity. This alternate representation has become conventional wisdom in other fields such as natural language processing and has many appealing properties. For example, this approach minimizes feature engineering and encodes semantic similarity. Due to its generality, it is possible that the same intermediate representation (such as from word2vec, GloVe, or fastText) can work for multiple datasets out-of-the-box. In contrast, traditional ER approach require hand tuning for each dataset. Further, DR toolkits such as fasttext [10] provide support for almost 300 languages allowing them (and thereby DEEPER) to work seamlessly on different languages. For the rest of this section, we assume the availability of GloVe pre-trained word vectors (of dimension d).

Let $\mathbf{v}(t)$ be the distributed representation for a tuple t while $\mathbf{v}(t[A_k])$ be the distributed representation for the value of attribute A_k in tuple t.

From Tuple to Vector – Basic Approach. Given a tuple t with m attributes, our objective is to convert it into a distributed representation as a matrix of real values $\mathbf{v}(t) \in \mathbb{R}^{m \times d}$, where the k-th row $\mathbf{v}_k(t)$ represents the vector for attribute A_k in tuple t, *i.e.,* $\mathbf{v}(t[A_k])$. We first break each attribute into its individual words using a standard tokenizer. For each token (word), we look up the GloVe pre-trained vector and retrieve the d-dimensional vector. Glove contains a special token UNK to represent any out-of-vocabulary word. If a word is not found in the GloVe dictionary (dubbed out-of-vocabulary scenario) or if the attribute has a NULL value, it is considered as UNK. In our initial approach, the vector representation for an attribute value is obtained by simply *averaging* the vectors of its tokens. Algorithm 2 describes the process.

From Tuple to Vector – Compositional Approach. An alternative approach is to use a compositional technique mo-

Algorithm 2 Tuple2vec-Averaging

Input: Tuple t, pre-trained vectors such as GloVe
Output: Distributed representation $\mathbf{v}(t)$ for t
for each attribute A_k of t **do**
    Pre-process and tokenize $t[A_k]$
    Look up vectors for tokens $w_l \in t[A_k]$ in GloVe
    $\mathbf{v}_k(t)$ = average of vectors of tokens in $t[A_k]$

Algorithm 3 Tuple2Vec-Compositional

Input: Tuple t, pre-trained vectors such as GloVe
Output: Distributed representation $\mathbf{v}(t)$ for t
for each attribute A_k of t **do**
    Pre-process and Tokenize $t[A_k]$
    Look up vectors for tokens $w_l \in t[A_k]$ in GloVe
    Pass the GloVe vectors for tokens through a LSTM-RNN composer to obtain $\mathbf{v}(t[A_k])$

tivated by the linguistic structures (*e.g.,* n-grams, sequence, and tree) in Natural Language Processing (NLP). In this approach, instead of simple averaging, we use a neural network to semantically compose the word vectors (retrieved from Glove) into an attribute-level vector. Different neural network architectures have been proposed to consider different types of linguistic structures, *e.g.,* convolutional network for n-gram level composition [13], recurrent network for sequential composition [14], [15], and recursive network for hierarchical tree-based composition [16]. The most popular compositional methods use a recurrent structure [14].

In our work, we use uni- and bi-directional recurrent neural networks (RNN) with LSTM hidden units [17], *a.k.a.* LSTM-RNNs. As shown in Figure 1(b), an RNN encodes a sequence (*e.g.,* an attribute value) into a vector by processing its word vectors sequentially, at each time step, combining the current input word vector with the previous hidden state. RNNs thus create internal states by remembering the output of the previous time step, which allows them to exhibit dynamic temporal behavior. We can interpret the hidden state $\mathbf{h}_l$ at time l as an intermediate representation summarizing the past. The output of the last time step $\mathbf{h}_L$ thus represents the attribute. LSTM cells contain specifically designed gates to store, modify or erase information, which allow RNNs to learn long range sequential dependencies. The LSTM-RNN shown in Figure 1(b) is unidirectional in the sense that it encodes information from left to right. Bidirectional RNNs [18] capture dependencies from both directions, thus provide two different views of the same sequence. For bidirectional RNNs, we use the concatenated vector $[\overrightarrow{\mathbf{h}_L}, \overleftarrow{\mathbf{h}_L}]$ as the final representation of the attribute value, where $\overrightarrow{\mathbf{h}_L}$ and $\overleftarrow{\mathbf{h}_L}$ are the encoded vectors from left to right and right to left, respectively.

For each word token in an attribute, we first look up its Glove vector. Then we use a shared LSTM-RNN to compose each attribute value in a tuple into a vector.[1] This results in a matrix $\mathbf{v}(t) \in \mathbb{R}^{m \times d}$, where the k-th row $\mathbf{v}_k(t)$ represents the vector for attribute A_k in tuple t. Algorithm 3 demonstrates the overall compositional process. It is important to note that the parameters of the LSTM-RNN model need to be learned on the ER task in a deep learning framework before it can be used to compose vectors for other off-the-shelf classifiers.

Computing Distributional Similarity. Given the distributed representation of a pair of tuples t_i and t_j, the next step is to compute the similarity between them. We compute similarity by applying a similarity metric to each corresponding rows of $\mathbf{v}(t_i)$ and $\mathbf{v}(t_j)$. More formally, $sim\left(\mathbf{v}_k(t_i), \mathbf{v}_k(t_j)\right)$, $\forall k \in [1, m]$. This returns a m-dimensional similarity vector. The most commonly used metric is cosine similarity. Other methods include subtracting (vector difference) or multiplying (hadamard product) the corresponding entries of the two vectors. One can also compute similarity using multiple similarity metrics and concatenate the resulting vectors. If we use q different metrics, the resulting similarity vector will have $q \times m$ entries.

Algorithm 4 ER-Classifier

Input: Table T, training set S
Output: All matching tuple pairs in Table T
for each pair of tuples (t_i, t_j) in S **do**
    Compute the distributed representation for t_i and t_j
    Compute their distributional similarity vector
Train a classifier $\mathcal{C}$ using the similarity vectors for S and true labels
for each pair of tuples (t_i, t_j) in T **do**
    Compute the distributed representation for t_i and t_j
    Compute their distributional similarity vector
    Predict match/mismatch for (t_i, t_j) using $\mathcal{C}$

C. Building an ER Classifier

It is fairly straightforward to build a classifier for ER using the above steps. For each of the pair of tuples in the training dataset, we compute their distributed representation through either Algorithm 2 or Algorithm 3. We then compute the similarity vector by measuring the similarity between corresponding attributes using different metrics. Given a set of positive and negative matching examples, we pass their similarity vectors to a classifier such as SVM along with their labels. Alternatively, the whole procedure (from distributed representation to classification) can be trained end-to-end in a DL framework as shown in Figure 1 (we will provide more details in Section V). The learned classifier can then be used to predict the match/mismatch label for any pair of tuples. Algorithm 4 provides the pseudocode.

IV. Blocking for Distributed Representations

Blocking. Efficient ER systems avoid comparing all $\binom{n}{2}$ possible pairs of tuples through the use of blocking [19],

[1] By the term 'shared' we mean the parameters of the model are shared across the attributes. In other words, the LSTM-RNNs for different attributes in a table share the same parameters.

[20]. Blocking identifies groups of tuples (called *blocks*) such that the search for duplicates need to be done only within blocks, thus greatly reducing the search space. If there are B blocks with b_{max} as the size of the largest block, this requires $O(b_{max}^2 \times B)$ which can be orders of magnitude faster for small values of b_{max}. While blocking often substantially reduces the number of comparisons, it may also miss some duplicates that fall in two different blocks.

A. New Opportunities for Blocking

The distributed representation of tuples enables a number of novel approaches for tackling blocking in a turn-key fashion. We observe that blocking is very related to the classical problem of approximate nearest neighbor (ANN) search in a similarity space, which has been extensively studied (see [21]). Locality sensitive hashing (LSH) is a popular probabilistic technique for finding ANNs in a high dimensional space. In the blocking context, the more similar input vectors are, the higher the probability that they both will be put in the same block. While we are not the first to propose LSH for blocking or automated tuning for blocking (see Section VII), we are the first to propose a series of truly turn-key algorithms that dramatically simplify the blocking process.

Challenges in Traditional Blocking Approach

- Identifying good blocking rules often requires the assistance of domain experts.
- Typically, blocking rules consider only 2-3 attributes which could result in comparing tuples that agree on those attributes but have very different values for others.
- Prior blocking methods often do not take semantic similarity between tuples into account.
- It is usually hard to tune the blocking strategy to control the recall and/or the size of the blocks.

We can readily see that LSH for blocking over DR of tuples obviates many of these issues. First, we free the domain experts from providing a blocking function. Instead the combination of LSH and DR transforms the problem of blocking into finding tuples in a high dimensional similarity space. Note that DR encodes semantic similarity into the mix and that LSH considers the entire tuple for computing similarity. The extensive amount of theoretical work on LSH (see Section IV-E) can be used to both tune and provide rigorous theoretical guarantees on the performance.

B. LSH Primer

Definition 1: (*Locality Sensitive Hashing* [22], [21]): A family $\mathcal{H}$ of hash functions is called (R, cR, P_1, P_2)-sensitive if for any two items p and q,

- if $dist(p,q) \leq R$, then $Prob[h(p) = h(q)] \geq P_1$, and
- if $dist(p,q) \geq cR$, then $Prob[h(p) = h(q)] \leq P_2$,

where $c > 1$, $P_1 > P_2$, $h \in \mathcal{H}$. □

The smaller the value of ρ ($\rho = \frac{\log(1/P_1)}{\log(1/P_2)}$), the better the search performance. For many popular distance measures such as cosine, Euclidean, and Jaccard, there exists an algorithm for the (R, c)-nearest neighbor problem that requires $O(dn + n^{1+\rho})$ space (where d is the dimensionality of p, q), $O(n^\rho)$ query time, and $O(n^\rho \log_{1/P_2} n)$ invocations of hash functions. In practice, LSH requires linear space and time [22], [21].

Implementing LSH. Given a table T, LSH seeks to index all the tuples in a hash table that is composed of multiple buckets each of which is identified by a unique hash code. Given a tuple t, the bucket in which it is placed by a (single) hash function h is denoted as $h(t)$ - which is often a binary value. If two tuples t and t' are very similar, then $h(t) = h(t')$ with high probability. Typically, one uses K hash functions $h_1(t), h_2(t), \ldots, h_K(t)$, $h_i \in \mathcal{H}$, to encode a single tuple t. We represent t as a K dimensional binary vector which in turn is represented by its hash code $g(t) = (h_1(t), h_2(t), \ldots, h_K(t))$. Since the usage of K hash functions reduces the likelihood that similar items will obtain the same (K dimensional) hash code, we repeat the above process L times - $g_1(t), g_2(t), \ldots, g_L(t)$. Intuitively, we build L hash tables where each bucket in a hash table is represented by a hash code of size K. Each tuple is then hashed into L different hash tables where its hash codes are $g_1(t), \ldots, g_L(t)$. For example, if $K = 10$ and $L = 2$, every tuple is represented as a 10-dimensional binary vector that is stored in 2 different hash tables.

Hash Families for Cosine Distance. Cosine similarity provides an effective method for measuring semantic similarity between two DRs [7]. Since the distributed representations can have both positive and negative real numbers, the cosine similarity varies between -1 and $+1$. The family of hash functions for cosine is obtained using the *random hyperplane* method. Intuitively, we choose a random hyperplane through the origin that is defined by a normal unit vector v. This defines a hash function with two buckets where $h(t) = +1$ if $v \cdot t \geq 0$ and $h(t) = -1$ if $v \cdot t < 0$ where $\cdot$ denotes the dot product between vectors. Since we require K hash functions $h_1, \ldots, h_K$, we randomly pick K hyperplanes and each tuple is hashed with them to obtain a K dimensional hash code. This process is then repeated for all L hash tables.

C. LSH-based Blocking

We begin by generating hash codes $h_1, \ldots, h_K$ for each of the L hash tables using the random hyperplane method. The set of hash functions $h_1, \ldots, h_K$ is analogous to a single blocking rule. The K dimensional binary hash code is equivalent to an identifier to a distinct block where t falls into. Each hash table performs "blocking" using a different blocking rule.

We index the distributed representation of every tuple t in each of the L hash tables. LSH guarantees that similar tuples get the same hash code (and hence fall into same block) with high probability. Then, we consider each of the blocks for every hash table and invoke the classifier over the distinct pairs of tuples found in them. Algorithm 5 provides the pseudocode of applying the classifier using this blocking approach.

Algorithm 5 is a fairly straightforward adaptation of LSH to ER. As we shall show in experiments, it works well empirically. However, the number of times a classifier would be in-

Algorithm 5 ER Classifier with LSH based Blocking

```
1: Input: Table T, training set S, L
2: Output: All matching tuple pairs in Table T
3: Generate hash functions for g_1, ..., g_L using the random
   hyperplane method
4: for each tuple t do
5:    Index t into L hash tables using g_1, ..., g_L
6: for each hash table g in [g_1, ..., g_L] do
7:    for each non-empty bucket H in g do
8:       for each pair of tuples (t_i, t_j) in H do
9:          Apply classifier on (t_i, t_j)
```

voked by this approach can be as much as $O(L \times b_{max}^2 \times B_{max})$ where L is the number of hash tables, b_{max} is the size of the largest block in any hash table and B_{max} is the maximum number of non-empty blocks in any hash table. While the traditional LSH based approach is often efficient and effective, one can achieve improved performance with some additional domain knowledge. We next describe a sophisticated approach to reduce the impact of L and b_{max}.

D. Multi-Probe LSH for Blocking

Recall that by increasing K, we ensure that the probability of dissimilar tuples falling into the same block is reduced. By increasing L, we ensure that similar tuples fall into the same block in at least one of the L hash tables. Hence while increasing L ensures that we will not miss a true duplicate pair, it is achieved at the additional cost of making extraneous com parisons between non-duplicate tuples. We wish to come up with a LSH based approach that achieves two objectives: (a) reduce the number of unnecessary comparisons and (b) reduce the number of hash tables L without seriously affecting recall.

Reducing Unnecessary Comparisons. Intuitively, we expect duplicate tuples to have a high similarity with each other and thereby more likely to be "near" each other. Hence, even if a block has a large number of tuples, it is not necessary to compare all pairs of tuples. Instead, given a tuple t, we can retrieve the top-N nearest neighbors of t and invoke the classifier between t and these N nearest neighbors. This can be achieved by collating all the tuples that fall into the same block as t in each of the L hash tables. We then compute the similarity between t and each of the candidates and return the top-N tuples. If the block is large with b tuples, then we only require $\Theta(b \times N)$ classifier invocations instead of $\Theta(b^2)$. We can see that by choosing $N < b$, we can achieve considerable reduction in classifier invocations.

Reducing L. Naively decreasing the number of hash tables L can decrease the recall as a pair of duplicate tuples might fall into different blocks. The key idea is to augment a traditional LSH scheme with a carefully designed probing sequence that looks for multiple buckets (of the same hash table) that could contain similar tuples with high probability. This approach is called multi-probe LSH [23]. Consider a tuple t and another very similar tuple t'. It is possible that t and t' do not fall into the same bucket (especially where K is large). However, due to the design of LSH, we would expect that t' fell into a "close by" bucket whose hash code is very similar to the bucket in which t fell. Multi-probe leverages this observation by perturbing t is a systematic manner and looking at all buckets in which the perturbed t fell into. By carefully designing the perturbation process one can consider the buckets that have the highest probability of containing similar tuples. It has been shown that this approach often requires substantially less number of hash tables (as much as 20x) than a traditional approach [23]. Algorithm 6 provides the pseudocode of this approach.

Algorithm 6 Approximate Nearest Neighbor based Blocking

```
1: Index all tuples using LSH
2: for each tuple t do
3:    Get candidate tuples using Multiprobe-LSH
4:    Sort tuples in candidates based on similarity with t
5:    Invoke classifier on t and each of top-N neighbors of t
```

E. Tuning LSH Parameters for Blocking

In contrast to traditional blocking rules that are often heuristics, the hash functions in LSH allow us to provide rigorous theoretical guarantees. While the list of LSH guarantees is beyond the scope of this paper (see [24] for details), we highlight two major ones - the ability to tune the parameters to achieve a predictable recall and occupancy rate by trading off the indexing and querying time.

Parameter Tuning for Recall. Recall that we need to select the parameters such that if two tuples are within a distance threshold of R, then the probability of them hashing to the same block in at least one hash table must be close to 1. On the other hand, if the distance between two records is greater than cR for some constant c, then the probability of them hashing to the same block in at least one hash table must be close to 0. We can control the false positive and negative values (and thereby recall) by varying the values of c and R, such as by setting the values that get the best results for the tuples in the training dataset. We can obtain a fixed approximation ratio of $c = 1 + \epsilon$ by setting [21],

$$K = \frac{\log n}{\log 1/P_2} \quad L = n^{\rho} \text{ where } \rho = \frac{\log(1/P_1)}{\log(1/P_2)} \qquad (1)$$

Parameter Tuning for Occupancy. LSH also allows us to control the occupancy - the expected number of tuples in any given block. This can be achieved by varying the size K of the number of hash functions in every hash table. Informally, if one uses multiple hash functions, we would expect *very* similar items to be stored in the same blocks but at the expense of low occupancy and a large number of blocks. On the other hand, a smaller number of hash functions results in less similar tuples being put in the same block. Intuitively, if we use only one hash function, this results in 2 buckets - one for $+1$ and -1.

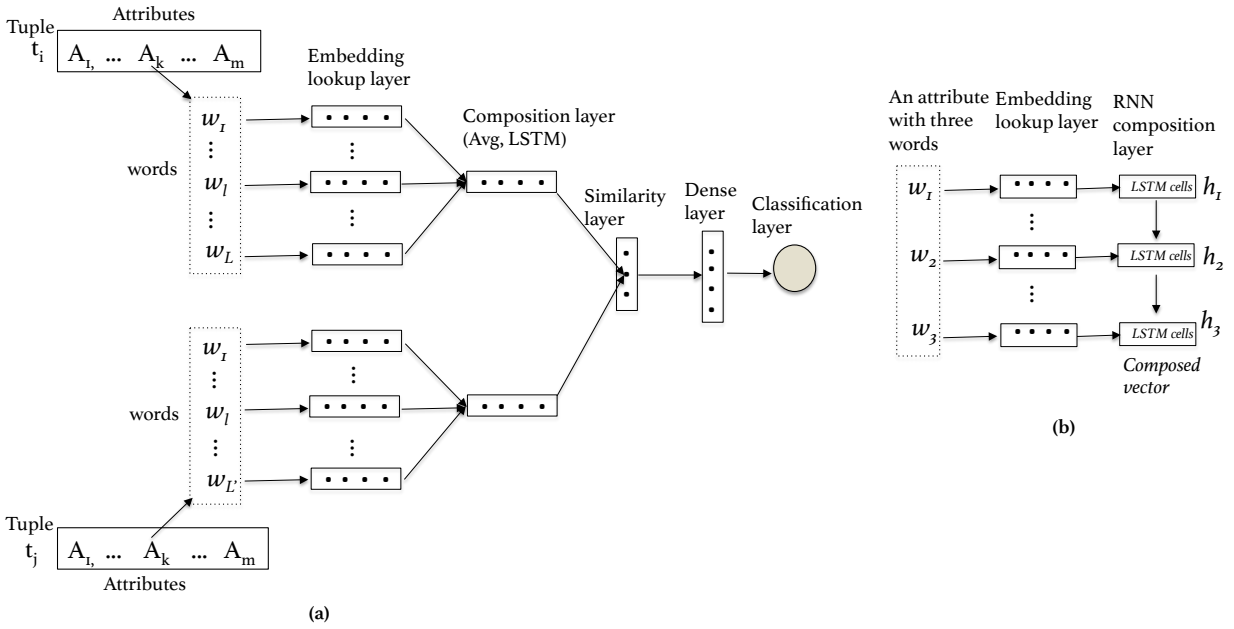


Fig. 1. (a) Deep entity resolution framework; (b) Recurrent Neural Network with Long Short-Term Memory cells in the hidden layer.

Since the hyperplane for the hash function is chosen randomly, we would expect each bucket to have an occupancy around 50% for all but most of the skewed data distributions. One can reduce the occupancy rate by increasing the number of hash functions. Alternatively, one can also use sophisticated methods such as [25] to achieve guaranteed limits.

V. Learning and Tuning Representations

In Section III, we described how to build a classifier by using pre-trained word vectors. In this section, we describe how the performance DeepER can be further improved by obtaining or tuning representations that are customized for the ER task. We begin by considering a number of practical scenarios where ER is performed.

Scenario 1: General Datasets. Many of the benchmark datasets used in ER [26] such as Citations, Products, Restaurants, and Movies, are often generic and do not require any specialized knowledge. While they may be noisy and incomplete, the content is often in English and use common words. For such generic datasets, the approach that we have proposed so far - convert pairs of tuples to similarity vectors using GloVe - is often adequate. As we shall show in the experiments, we obtain competitive results for all of them.

Scenario 2: Specialized Databases with Domain Knowledge Repository. Another common scenario occurs when ER is performed on a specialized database that might require some domain knowledge. Examples include performing ER on scientific articles for specialized fields or in data that is specific to an organization. Often, one could use a vast corpus of domain information in the form of unstructured data such as documents. As an example, one could use articles in PubMed for learning domain knowledge when deciding to perform ER on biological articles or use the organization's document repository for ER on data in the same organization. In this scenario, one can learn distributed representations using the available tools such as GloVe[7], word2vec[6] or fastText[10], where the training is done on the relevant corpus. Once trained, the representation might encode some semantic similarity whereby it might know that gene-A and gene-B are more similar to each other than to gene-C. This approach also works for multi-lingual data. Tools such as fastText[10] already provide vector representations for almost 300 languages.

Scenario 3: Specialized Databases with No Domain Knowledge Repository. Of course, the worst case scenario is a specialized database where no auxiliary resources are available to automatically learn the representation for key concepts. In this scenario, any machine learning approach is doomed to fail unless one provides hand crafted features or a substantially large number of training examples that are sufficient for learning representations using deep learning [6], [7], [10].

End-to-End Training

One crucial advantage of a DL framework is that it allows us to fine-tune the pre-trained word representations (scenarios 1 and 2) or to learn word representations directly from scratch starting from random initializations (scenario 3) for a specific task. In general, fine-tuning of representations on the task improves the accuracy [13]. This stems from the fact that the pre-trained vectors such as Glove and word2vec are learned through unsupervised methods that attempt to encode semantic relationships between words by exploiting contextual information (*i.e.,* neighboring words). Therefore, these representations often lack task-specific knowledge. Furthermore, unsupervised pre-training on a large corpus gives the network

better generalization. In fact, this paradigm of unsupervised pre-training followed by supervised fine-tuning often beats methods that are based on only supervision [13].

Let us now consider our deep neural network in Figure 1. We train this network using Stochastic Gradient Descent or SGD-based learning algorithms, where gradients (errors) are obtained via backpropagation. In other words, errors in the output layer are backpropagated through the hidden layers using the chain rule of derivatives. For learning or fine-tuning the embeddings, these errors are backpropagated till the word embedding layer. One common issue with backpropagation through a deep neural network (*i.e.,* neural networks with many hidden layers such as RNNs) is that as the errors get propagated, they may soon become very small (*a.k.a.* gradient vanishing problem) or very large (*a.k.a.* gradient exploding problem) that can lead to undesired values in weight matrices, causing the training to fail [27]. We did not observe such problems in our end-to-end training with simple averaging compositional method, and the gates in LSTM cells automatically tackle these issues to some extent [28].

VI. Experimental Results

A. Experimental Setup

Hardware and Platform. All our experiments were performed on a Core i7 6700HQ Skylake chip, with four cores running eight threads at speeds of up to 3.5GHz, along with 16GB of DDR4 RAM and the GTX 980M, complete with 8GB of DDR5 RAM. We used Torch [29], a deep learning framework, to train and test our models. Scikit-Learn [30] was used to train the baseline SVM classifiers.

Datasets. We conducted extensive experiments over 4 different datasets covering diverse domains such as citations and e-commerce. Table I provides some statistics of these datasets. All the datasets chosen are popular benchmark datasets and have been extensively evaluated by prior ER work using both ML and non-ML based approaches. We partition our datasets into two categories: "easy" and "challenging". The former consists of datasets that are mostly structured and often have less noise in terms of typos and missing information. On the "easy" datasets most of the best existing ER approaches routinely exceed an F-score of 0.9. The challenging datasets often have unstructured attributes (such as product description) and also noisy. On the "challenging" datasets that we study, both ML and rule based methods have struggled to achieve high F measures, with values between 0.6 and 0.7 being the norm. What these two categories have in common is that they require extensive effort from domain experts for cleaning, feature engineering and blocking to achieve good results. As we shall show later, our approach exceeds best existing results on all the datasets with minimal expert effort.

DeepER **Setup.** Our experimental setup was an adaptation of prior ER evaluations methods [33], [5], [1] to handle distributed representations. For example, [33] used an arbitrary threshold (such as 0.1) on Jaccard similarity of trigram to eliminate tuple pairs that are clearly non-matches. We make two changes to this procedure. First, we use Cosine similarity to compute similarity between tuple pairs as it is more appropriate for distributed representations [6], [7]. Second, instead of picking an arbitrary threshold, we set it to the minimum similarity of matched tuple pairs in the *training* dataset. We obtain the negative examples (non duplicates) by picking one tuple from the positive example and randomly picking another tuple from the relation that is not its match. For example, if (t_i, t_j) is a duplicate, we pick a pair (t_k, t_l) as a negative example such that (t_k, t_l) is not a duplicate already given in the training data and has cosine similarity with (t_i, t_j) below the above computed threshold. This approach is chosen to verify the robustness of our models against near matches. For each of the datasets, we performed K-fold cross validation with K=5. We report the average of the F-measure values obtained across all the folds. We observed that in all cases, the standard deviation of the F-measure values was below 1%.

Dataset	#Tuples	#Duplicates	#Attr
Walmart-Amazon (Prod-WA)‡ [26]	(2,554 - 22,074)	1,154	17
Amazon-Google (Prod-AG)‡ [31]	(1,363 - 3,226)	1,300	5
DBLP-Scholar (Pub-DS)* [31]	(2,616 - 64,263)	5,347	4
Fodors-Zagat (Rest-FZ)* [32]	(533 - 331)	112	7

TABLE I
Dataset Statistics - * (Easy), ‡ (Challenging)

DeepER **Architecture.** Since our objective is to highlight the turn-key aspect of DeepER, we choose the simplest possible architecture. We use GloVe[7] as our distributed representation. Each tuple is represented as a $m \times d$ dimensional vector where m is the number of attributes with d being the dimension of distributed representations. For each attribute, we apply a standard tokenizer and average the DR obtained from GloVe (as against more sophisticated approaches such as Bi-LSTM). Given a pair of tuples, the compositional similarity is computed as the Cosine similarity of the corresponding attributes resulting in a m dimensional similarity vector. As mentioned above, we used K-fold validation with a duplicate to non-duplicate ratio of of 1:100 that is comparable to the ratio used by competing approaches. Note that the non-duplicates are sampled automatically. We also do not tune the DR for the ER task. Even with this restricted setup, DeepER is competitive with competing approaches. In Section VI-D, we systematically vary each of these components of DeepER architecture and show that each of them improves the performance further.

B. Evaluating DeepER

We conducted an extensive set of experiments that show that distributed representations is a promising approach to build turn-key ER systems. The key questions that we seek to answer with our evaluation are:

Dataset	Magellan			DEEPER			F-Measure (Published)
	Precision	Recall	F-Measure	Precision	Recall	F-Measure	
Prod-WA	80.705	85.435	82.99	83.61	93.035	88.06	89.3 [34] (Crowd)
Prod-AG	79.92	97.11	87.68	96.68	95.38	96.029	62.2 [5] (ML)
Pub-DS	98.02	99.67	98.84	99.08	96.3	97.67	92.1 [34] (Crowd)
Rest-FZ	100	100	100	100	100	100	96.5 [34] (Crowd)

TABLE II
COMPARING THE PERFORMANCE OF DEEPER WITH STATE-OF-THE-ART PUBLISHED RESULTS FROM EXISTING RULE BASED, ML BASED AND CROWD BASED APPROACHES. WE ALSO COMPARED AGAINST MAGELLAN [2], ANOTHER END-TO-END EM SYSTEM.

- How does the performance of DEEPER compare against published state-of-the-art results from traditional ML, rule and crowd based approaches?
- How does DEEPER compare against similar end-to-end EM pipelines such as Magellan [2]?
- How does the performance of DEEPER improve when we augment its simple architecture with enhancements such as tuning representations and using sophisticated compositional approaches?
- How does LSH based blocking approaches work in practice?

C. Comparison with Existing Methods

In our first set of experiments, we first compare the performance of DEEPER with the best reported results from prior work. We consider non-learning, learning and crowd based approaches from [5], [34], [35]. Table II shows the F-measure values for both DEEPER and the best published result. These referenced F-scores represent the best effort from the respective authors and the simple DEEPER architecture is competitive with them.

Recall that performing ER on a large dataset requires a number of design choices from the expert such as feature engineering, selection of appropriate similarity functions and thresholds, parameter tuning for ML models, selection of appropriate blocking functions, and so on. Hence, it is incredibly hard to take any of the existing approaches and apply it as-is on a new dataset. A key advantage of DEEPER is the ability to dramatically reduce this effort. In order to highlight this feature, we evaluated DEEPER against Magellan [2] that also has a end-to-end EM pipeline. We would like to emphasize that both DEEPER and Magellan share the dream of making the EM process as frictionless as possible. While Magellan uses a series of sophisticated heuristics internally, DEEPER leverages distributed representations as a foundational technique. It is very easy to incorporate features from DEEPER and Magellan to each other. For example, one can augment Magellan's automatically derived similarity based features to DEEPER while Magellan can readily use the blocking of DEEPER and so on. Table II compares the performance of default settings of DEEPER and Magellan. Specifically, we adapted the end-to-end EM workflow for Magellan [36]. We can see that DEEPER beats Magellan on two datasets, while performing slightly worst in one datasets. Both systems delivered perfect results in the rather simple Fodors-Zagat dataset.

D. Understanding DEEPER Performance

In this section, we investigate how each of the enhancements to the basic DEEPER architecture impacts its performance. Specifically, we consider 4 dimensions - training data size, fraction of incorrectly labeled training dataset, tuning of DR and the compositional approach used. Hence, we compare the performance of DEEPER with and without the said modification. For this subsection, we use a positive to negative ratio of 1:4.

Varying the Size of Training Data. Figure 2 shows the results of varying the amount of training data. DEEPER is robust enough to be competitive with competing approaches with as little as 10% of training data. Note that 10% translates to as little as 11 examples that needs to be labeled in the case of Rest-FZ and to 543 in the case of Pub-DS. As expected, our method improves its excellent results with larger training data.

Impact of Incorrect Labels. Most of the prior work on ER assume that the training data is perfect. However, this assumption might not always hold in practice. Given the increasing popularity of crowdsourcing for obtaining training data, it is likely that some of the labels for matching and non-matching pairs are incorrect. We investigate the impact of incorrect labels in this experiment. For a fixed set of training data (10%), we vary the fraction of labels that are marked incorrectly. Figure 3 shows the results. While the F-measure reduces with larger fraction of incorrect labels, the experiments also show that our approach is very robust. The average drop in F-measure values compared to the perfect labeling case across all datasets at 10% noise is just 2.6 with a standard deviation of 2.6. At 30% the average drop is 8% with a standard deviation of 7%. We can also see that at 10% noising, our approach is still competitive with state-of-the-art approaches.

Dynamic vs Static Word Embeddings. In this set of experiments, we evaluated the effect of updating (or fine-tuning) the initial word embeddings obtained from GloVe as part of training the model. In other words, we evaluated if tuning the distributed representation for ER tasks improves the performance of our model. Figure 4 shows the results. The results matched our intuition that for the "challenging" datasets, updating the word embeddings in an end-to-end learning framework helped boost the results a little, whilst for the "easy" ones, it had either a small negative effect or no effect at all. Thus, we advise that in general, it is better to use the end-to-end framework.

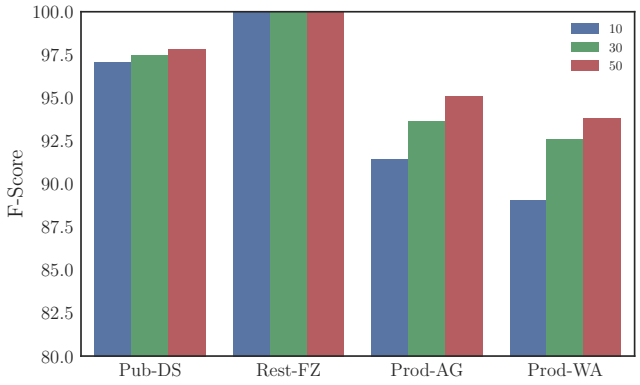


Fig. 2. Varying Size of Training Data

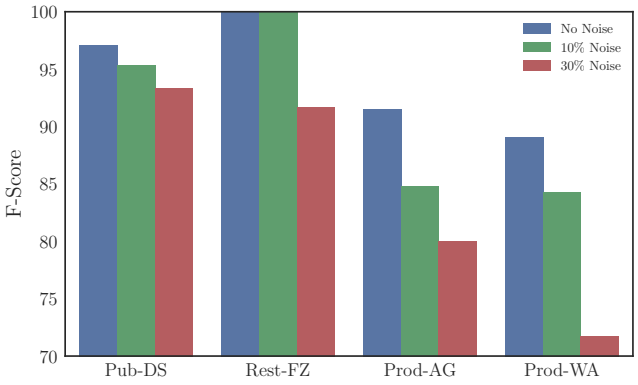


Fig. 3. Varying Noise

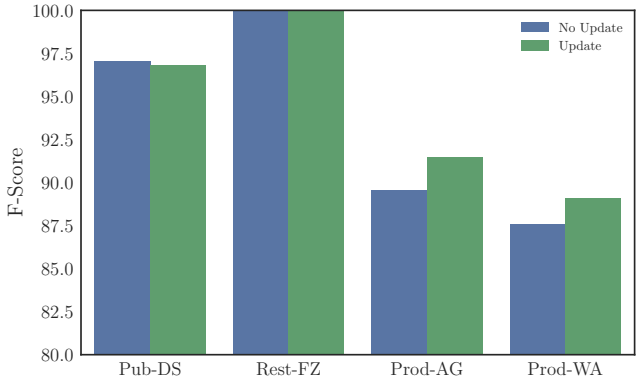


Fig. 4. Varying Vector Updates

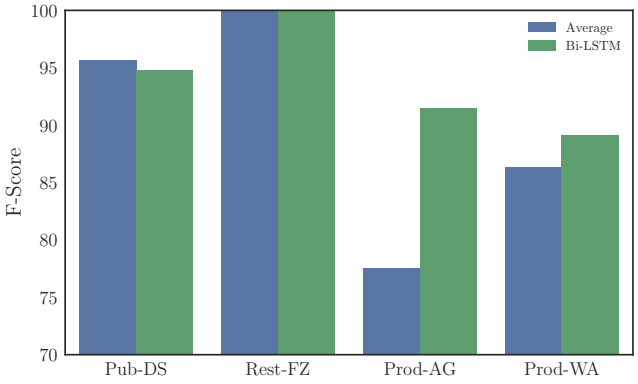


Fig. 5. Varying Composition

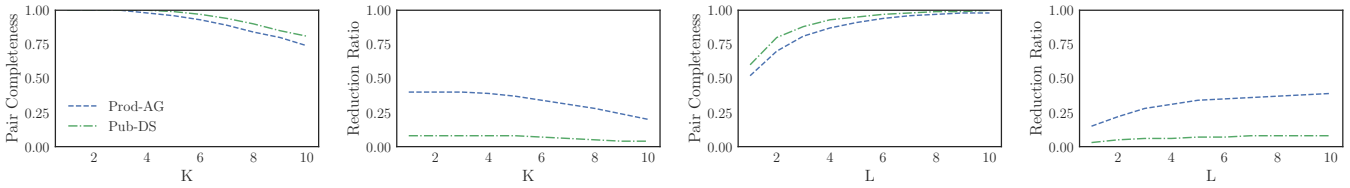
Varying Composition. In this set of experiments, we vary the compositional method we use to combine the individual word embeddings into a single representative vector for the tuple/attribute. Figure 5 shows that for the "easy" datasets, simple word averaging work better than recurrent compositional models (LSTM or BiLSTM). This flips for the "challenging" datasets. Intuitively, this is due to the importance of word order dependency in values of attributes like product names or product description, and that observing such order is either immaterial or rather hurtful for values of attributes like the list of authors. In order to use the more complex compositional methods (LSTM or BiLSTM) one has to pay the price of its longer training times, one also has to tune its additional hyperparameters. However, even the simple averaging compositional technique is competitive with prior approaches on all datasets.

E. Evaluating LSH based Blocking

In this subsection, we evaluate the performance of our LSH based blocking approach. Our approach allows us to vary K (the size of the hash code) and L (the number of hash tables) in order to achieve a tunable performance. Recall that one can use Equation 1 to derive K and L based on the task requirements. Suppose we wish that similar tuples should fall into same bucket with probability $P_1 = 0.95$ and dissimilar tuples should fall into the same bucket with probability $P_2 \leq 0.5$. Suppose that we index the DBLP dataset of Pub-DS. Then based on Equation 1, we need a LSH with $K = 12$ and $L = 2$.

In our first set of experiments, we verify that the behavior of blocking is synchronous with the theoretical expectations. We evaluate the performance of blocking based on two metrics widely used in prior research [19], [20], [37]. The first metric, efficiency or reduction ratio (RR), is the ratio of the number of tuple pairs compared by our approach to the number of all possible pairs in T. In other words, a smaller value indicates a higher reduction in the number of comparisons made. The second metric, recall or pair completeness (PC), is the ratio of the number of true duplicates compared by our approach against the total number of duplicates in T. A higher value for PC means that our approach places the duplicate tuple pairs in the same block.

Figures 6(a)-6(d) shows the results of our experiments. As K is increased, the value of PC decreases. This is due to the fact that for a fixed L, increasing K reduces the likelihood that two similar tuples will be placed in the same block which in turn reduces the number of duplicates that falls into the same block. However, for a fixed L, increasing K dramatically decreases the RR. This is to be expected as a larger value of



(a) K vs PC (L=10) (b) K vs RR (L=10) (c) L vs PC (K=4) (d) L vs RR (K=4)

Fig. 6. Impact of varying K and L on Pair Completeness (PC) and Reduction Ratio (RR). Legend for Figures 6(b)-6(d) is same as that of Figure 6(a)

K increases the number of LSH buckets into which tuples can be assigned to.

A complementary behavior can be observed when we fix K and vary L. When L is increased, PC also increases. This is to be expected as the probability that two similar tuples being assigned to the same bucket increases when more than one hash table is involved. In other words, even if a true duplicate does not fall into the same bucket in one hash table, it can fall into the same bucket in other hash tables. However, increasing L has a negative impact on RR as a number of false positive tuple pairs can fall into the same bucket in at least one hash table thereby increasing the value of RR.

Evaluating Multi-Probe LSH. We evaluate Algorithm 6 using Multi-probe and comparing a tuple only with top-N most similar tuples instead of all tuples in a block. Figure 7 shows the result for Pub-AG. We vary the number of multi-probes and pick the top-N most similar tuples to be classified. We measure the recall of this approach for $K = 10$ and an extreme case with a *single* hash table where $L = 1$. We wish to highlight two trends. First, even using a single multi-probe sequence can dramatically increases the recall. This supports our claim that one can increase recall using a small number of hash tables by using multi-probe LSH. Second, increasing the size of N does not dramatically increase the recall. This is due to the fact that duplicate tuples have high similarity between their corresponding distributed representations. Our top-N based approach would be preferable to reduce the number of classifier invocations when the block size is much higher than 10.

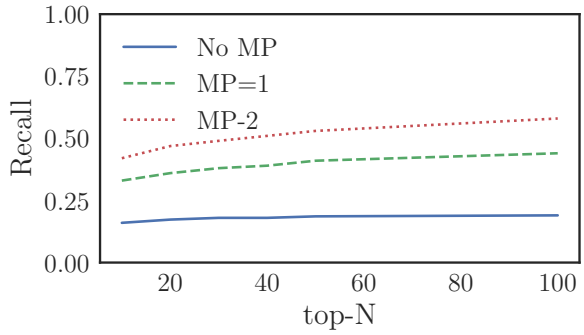


Fig. 7. Performance of MultiProbe LSH on Pub-AG

VII. Related Work

Entity Resolution. A good overview of ER can be found in surveys such as [4], [3]. Prior work on ER can be categorized as based on (a) declarative rules, (b) machine learning, and (c) expert or crowd based. Declarative rules such as DNF [9] that specify rules for matching tuples are easily interpretable [1] but often requires a domain expert. Most of the machine learning approaches are variants of the classical Fellegi-Sunter model [8]. Popular approaches include SVM [9], active learning [38], clustering [39]. Recently, ER using crowdsourcing has become popular [40], [34]. While there exist some work for learning similarity functions and thresholds [9], [41], ER often requires substantial involvement of the expert.

There has been extensive work on building EM systems. Please refer to [2] for a comprehensive survey of the current EM systems. Most of the prior works often do not cover the entire EM pipeline, require extensive interaction with experts and are not turn-key systems. The key objective of DeepER is the same as Magellan [2]. We aim to propose a end-to-end EM system based on distributed representations that minimizes the burden on the experts. Our techniques are modular enough and can be easily incorporated into any of the existing systems.

Blocking. Blocking has been extensively studied as a way to scale ER systems and a good overview can be found in surveys such as [19], [20]. Common approaches include key based blocking that partitions tuples into blocks based on their values on certain attributes and rule based blocking where a decision rule determines which block a tuple falls into. There has been limited work on simplifying this process by either learning blocking schemes such as [37] or tuning the blocking [42]. In contrast, our work automates the blocking process by requiring minimal input from the domain expert.

There has been some recent work on using LSH for blocking. [24] uses MinHashing where tuples with high Jaccard similarity with each other are likely to be assigned to the same block. [43] improves it by proposing a MinHashing with semantic similarity based on concept hierarchy to assign conceptually similar tuples to the same block. Our approach based on distributed representation does not require such taxonomy and can handle more sophisticated semantic similarity than [43]. [44] proposed a clustering based method to satisfy size constraints with upper and lower size thresholds for blocks for performance and privacy reasons. We leverage the prior work on tuning of LSH hash functions for occupancy rates

for achieving similar behavior *on expectation*.

Deep Learning. A good overview of deep learning can be found in [11]. We leverage two fundamental ideas from Deep Learning - Distributed Representations [12] and Compositions [45]. There are a number of popular and open source distributed representation techniques such as GloVe [7], word2vec [46], [6] and fastText [10]. Since distributed vectors are often defined for words, one has to use composition to obtain distributed representations for complex objects such as phrases [6], sentences [47] and documents [47].

VIII. Final Remarks

In this paper, we introduced DeepER, a deep learning based approach for entity resolution. Our fundamental contribution is the identification of the concept of distributed representation as a key building block for designing effective ER classifiers. We also propose algorithms to transform a tuple to a distributed representation, building DR aware classifiers and an efficient blocking strategy based on LSH. Our extensive experiments show that this approach is promising and already achieves or surpasses state-of-the-art results on multiple benchmark datasets. We believe that deep learning is a powerful tool that has applications in databases beyond entity resolution and it is our hope that our ideas be extended to build practical and effective entity resolution systems.

References

[1] R. Singh, V. Meduri, A. K. Elmagarmid, S. Madden, P. Papotti, J. Quiané-Ruiz, A. Solar-Lezama, and N. Tang, "Generating concise entity matching rules," in *PVLDB*, 2017.

[2] P. Konda, S. Das, P. Suganthan GC, A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton *et al.*, "Magellan: Toward building entity matching management systems," *PVLDB*, 2016.

[3] F. Naumann and M. Herschel, "An introduction to duplicate detection," *Synthesis Lectures on Data Management*, vol. 2, no. 1, pp. 1–87, 2010.

[4] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *TKDE*, vol. 19, no. 1, 2007.

[5] H. Köpcke, A. Thor, and E. Rahm, "Evaluation of entity resolution approaches on real-world match problems," *PVLDB*, 2010.

[6] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013.

[7] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *EMNLP*, 2014.

[8] I. Fellegi and A. Sunter, "A theory for record linkage," *Journal of the American Statistical Association*, vol. 64 (328), 1969.

[9] M. Bilenko and R. J. Mooney, "Adaptive duplicate detection using learnable string similarity measures," in *KDD*, 2003.

[10] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *arXiv preprint arXiv:1607.04606*, 2016.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[12] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *JMLR*, 2003.

[13] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *JMLR*, 2011.

[14] J. Li, T. Luong, D. Jurafsky, and E. Hovy, "When are tree structures necessary for deep learning of representations?" in *EMNLP*, 2015.

[15] P. Liu, S. Joty, and H. Meng, "Fine-grained opinion mining with recurrent neural networks and word embeddings," in *EMNLP*, 2015.

[16] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *EMNLP*, 2013.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[18] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE TSP*, 1997.

[19] R. Baxter, P. Christen, and T. Churches, "A comparison of fast blocking methods for record linkage," in *SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003.

[20] P. Christen, "A survey of indexing techniques for scalable record linkage and deduplication," *TKDE*, 2012.

[21] J. Wang, H. T. Shen, J. Song, and J. Ji, "Hashing for similarity search: A survey," *arXiv preprint arXiv:1408.2927*, 2014.

[22] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, vol. 99, no. 6, 1999, pp. 518–529.

[23] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: efficient indexing for high-dimensional similarity search," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 950–961.

[24] R. C. Steorts, S. L. Ventura, M. Sadinle, and S. E. Fienberg, "A comparison of blocking methods for record linkage," in *PSD*, 2014.

[25] M. Covell and S. Baluja, "Lsh banding for large-scale retrieval with memory and recall constraints," in *ICASSP*, 2009.

[26] S. Das, A. Doan, P. S. G. C., C. Gokhale, and P. Konda, "The magellan data repository," https://sites.google.com/site/anhaidgroup/projects/data.

[27] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Trans. Neur. Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994.

[28] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[29] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[30] F. e. a. Pedregosa, "Scikit-learn: Machine learning in python," *JMLR*, 2011.

[31] "Benchmark datasets for entity resolution," https://dbs.uni-leipzig.de/en/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution.

[32] "Duplicate detection, record linkage, and identity uncertainty: Datasets," http://www.cs.utexas.edu/users/ml/riddle/data.html.

[33] H. Köpcke and E. Rahm, "Training selection for tuning entity matching." in *QDB/MUD*, 2008.

[34] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. W. Shavlik, and X. Zhu, "Corleone: hands-off crowdsourcing for entity matching," in *SIGMOD*, 2014, pp. 601–612.

[35] S. Das, P. S. G. C., A. Doan, J. F. Naughton, G. Krishnan, R. Deep, E. Arcaute, V. Raghavendra, and Y. Park, "Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services," in *SIGMOD*, 2017, pp. 1431–1446.

[36] Magellan. (2017) End-to-end em workflows. [Online]. Available: http://anhaidgroup.github.io/py_entitymatching/v0.1.x/user_manual/guides.html#end-to-end-em-workflows

[37] M. Michelson and C. A. Knoblock, "Learning blocking schemes for record linkage," in *AAAI*, 2006, pp. 440–445.

[38] S. Sarawagi and A. Bhamidipaty, "Interactive deduplication using active learning," in *KDD*, 2002.

[39] W. W. Cohen and J. Richman, "Learning to match and cluster large high-dimensional data sets for data integration," in *KDD*, 2002.

[40] J. Wang, T. Kraska, M. J. Franklin, and J. Feng, "Crowder: Crowdsourcing entity resolution," *PVLDB*, vol. 5, no. 11, pp. 1483–1494, 2012.

[41] J. Wang, G. Li, J. X. Yu, and J. Feng, "Entity matching: How similar is similar," *PVLDB*, 2011.

[42] B. Kenig and A. Gal, "Mfiblocks: An effective blocking algorithm for entity resolution," *Information Systems*, 2013.

[43] Q. Wang, M. Cui, and H. Liang, "Semantic-aware blocking for entity resolution," *TKDE*, 2016.

[44] J. Fisher, P. Christen, Q. Wang, and E. Rahm, "A clustering-based framework to control block sizes for entity resolution," in *KDD*, 2015.

[45] J. Mitchell and M. Lapata, "Composition in distributional models of semantics," *Cognitive science*, vol. 34, no. 8, pp. 1388–1429, 2010.

[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[47] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *ICML*, 2014.