Noise Resistant Deep Entity Matching

Yibin Zhang
University of Toronto
ybzhang@cs.toronto.edu

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

ABSTRACT

Overview

PVLDB Reference Format:
. . *PVLDB*, (): xxxx-yyyy, .
DOI:

1. INTRODUCTION

Entity matching (EM) or entity resolution refers to the problem of determining whether two real world records refer to the same real world entity. For example in Figure 1 the problem is to decide which pairs of records one from each table refer to the same product. This problem has attracted profound research interest over the last fifty years pointing to its vast significance in research and practice []. As the tables typically involved in an EM process are large, the overall processing pipeline involves a process called *blocking* that effectively limits the pairs of records that have to be considered in the EM process.

In the last years, rapid advances in Deep Learning [] models and associated technologies have introduced highly sophisticated learning based architectures that have been successfully applied in the EM problem []. Very recently *Transformer based* language models [] have been introduced yielding impressive results for several language based tasks. Similarly such models have been applied recently to the EM problem [] demonstrating that a Transformer language model (TLM) based approach is the state of the art approach for learning based EM. The basic idea behind the success of TLMs, is that EM, especially for complex entities (e.g., product descriptions) requires detailed language understanding and this is precisely what TLMs offer. In particular such models offer highly contextualized embeddings that capture language constructs much better than more traditional word embeddings. When TLM are utilized for EM, a pre-trained TLM is deployed, suitably modified (e.g., BERT) and is fine tuned with EM specific training data.

Over the years, non-learning based EM techniques and algorithms [] have utilized a number of *similarity measures* [] to quantify closeness of attribute values or entities. One of the main motivations for such measures has been that depending on the application domain, traditional attributes types (e.g., strings) may contain errors. Such errors manifest as typographical errors (e.g., misspellings), abbreviations, character transpositions during typing, differences in encoding of certain types (e.g., date formats) to name a few []. In certain domains (e.g., product descriptions) differences between two real world entities manifest in ad-hoc ways much more broadly than misspellings. For example in a description of a product part, different sources may rely on different metric systems to express product specifics.

Deep models rely on learning to make their inferences. As such, their predictions are as good as the training data utilized. In the case of deep EM models, typical errors (e.g., misspellings) existing in the data, will only be taken into account if the model has learned to cope with such errors. For this reason, examples of such errors should be present in training data. Deep models rely on techniques such as *data augmentation* [] and *adversarial training* in order to enhance the training data set with data patterns that are not typically part of the original training data.

TLMs are trained over very large text collections (e.g., Wikipedia) in order to gain language understanding. In this paper we seek to conduct a detailed investigation of how sensitive TLMs are to input noise when deployed for EM tasks. We adopt intuitive noise models seeking to understand how sensitive transformer based EM approaches are to increasing noise. We deploy techniques from the literature to enhance transformer models with noise understanding, deploying both data augmentation and adversarial training and highlighting the benefits and limitations of each utilizing real data sets.

We then propose an encoding of the input, applicable to both training and test data, that aims to alleviate the impact noise has in the operation of the transformer models. Our proposed encoding named *Noise Resistant Encoding* (NRE) operates on training data as a pre-processing step. It employs a specific type of grouping into the token representation of each attribute domain, employing similarity measures from the literature between tokens, essentially encoding similar tokens to a common input representation. In that sense, NRE acts as a bridge between the vast techniques developed over the years on similarity measures and learning based deep EM. We present NRE analyzing its performance and accuracy when deployed on real data sets. The overall architecture of where NRE fits in an EM pipeline is depicted as Figure 2.

Proceedings of the VLDB Endowment, Vol. , No.
ISSN 2150-8097.
DOI:

title	manufacturer	price		title	manufacturer	price
adobe acrobat 7.0 professional	adobe	449.99	unmatched	Adobe premiere pro cs3 upgrade	Adobe	299.00
ulead videostudio 11 plus	corel	129.99	matched	Video studio 11 plus	Corel corporation	103.99
partition commander 10	avanquest	49.95	matched	Avanquest usa llc partition commander 10	NULL	43.32

Figure 1: Entity Matching example in a relational setting

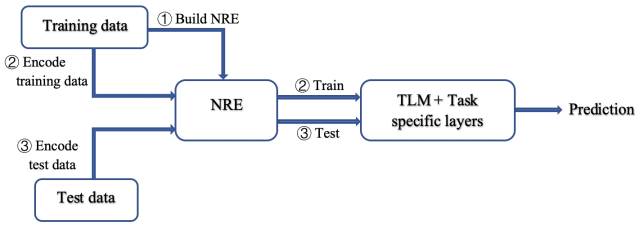

Figure 2: Matching Architecture. ① Training data are used to build our NRE encoding, ② NRE encodes training data for training downstream matcher, and ③ NRE encodes test data for downstream matcher to make prediction.

More specifically in this paper we make the following contributions:

- We investigate the impact of noise on the accuracy of state of the art TLMs for EM for a variety of benchmark data sets.
- We present the impact and the effect data augmentation and adversarial training have for noisy inputs on state of the art TLMs for EM on real data sets.
- We present NRE an algorithm to encode the input to TLM utilized for EM models (both during fine tuning and testing), that essentially standardizes noisy input into a common representation. We evaluate the impact of NRE on EM tasks using real data sets and highlight the benefits of the approach, varying parameters of interest. NRE utilizes similarity measures across attribute values which have been prevalent in non-learning based EM, thus in a way aiming to bring the best of both worlds in the EM process.
- Our results indicate that the proposed NRE methodology is a promising way to encode noisy inputs in a common representation, significantly increasing accuracy of TLM applied for EM especially when inputs become increasingly noisy.
- NRE is an additional step that can be easily adopted, in any existing TLM for EM pipeline. It can be adapted to different types of noisy inputs and can significantly improve the accuracy of the matching process.

This paper is organized as follows.

2. BACKGROUND

We present a brief overview of TLM and their adaptation for EM. Past learning-based EM solutions rely on word embeddings and custom RNN architectures to train the EM models []. In contrast TLMs for EM train the models by fine-tuning pre-trained language models. Pre-trained models such as BERT and GPT-2/3 have demonstrated good performance on many NLP tasks. They are typically deep neural networks with multiple Transformer layers [], typically 24 layers, pre-trained on large text corpora such as Wikipedia articles in an unsupervised manner. During pre-training, the model is trained to perform various tasks such as missing token and next-sentence prediction. Prior work [] has demonstrated that the shallow layers capture lexical meaning while the deeper layers capture syntactic and semantic meanings of the input sequence after pre-training.

Pre-trained language models learn the semantics of words better than conventional word embeddings, such as word2vec, GloVe, or FastText. This is because the Transformer architecture calculates token embeddings from all the tokens in the input sequence and thus, the embeddings it generates are highly-contextualized capturing the semantic and contextual understanding of the words. Consequently, such embeddings can capture constructs such as polysemy, i.e., the same word may have different meanings in different phrases. For example, the word apple has different meanings in "apple computers" versus "apple picking". Pre-trained language models will embed "apple" differently depending on the context while traditional word embedding techniques such as FastText always produce the same vector independent of the context.

The basic approach to utilize TLM for EM in the literature [] is to fine tune a pre-trained language model (e.g., BERT []) for the specific task of entity matching. More specifically, following [], a pre-trained language model is trained for the EM task with a labeled training data set consisting of positive and negative pairs of matching and non-matching entries as follows. First one adds task-specific layers after the final layer of the language model. For EM, one typically adds a simple fully connected layer and a softmax output layer for binary classification. Second, we initialize the modified network with parameters from the pre-trained language model (e.g., BERT). Last, we train the modified network on the training set until it converges. This basic approach is independent of the underlying language model adopted (e.g., BERT [], RoBERTa [], DistilBERT []).

A typical EM pipeline accepts as input two collections of data entries D_1 and D_2 (e.g., rows of relational tables, JSON files, raw text) and outputs a set $M \subset D_1 \times D_2$ of pairs where each pair $(e_1, e_2) \in M$ represents the same real-world entity (e.g., person, product, etc). A data entry e is a set of key-value pairs $e = \{(a_i, v_i)\} 1 \leq i \leq k$ where a_i is the attribute name and v_i is the attribute value in a text representation. This definition of data entries captures both structured and semi-structured data such as JSON.

Since language models accept token sequences as input, deep transformer based EM networks [] convert the candi-

date pairs into token sequences so that they can effectively be processed. The EM network serializes data entries as follows []: for each data entry $e = \{(a_i, v_i)\} 1 \leq i \leq k$, let serialize(e) ::= [COL] a_1 [VAL] v_1 ... [COL] a_k [VAL] v_k, where [COL] and [VAL] are special tokens for indicating the start of attribute names and values respectively.

To serialize a candidate pair (e_1, e_2), we let serialize(e_1, e_2) ::= [CLS] serialize(e_1) [SEP] serialize(e_2) [SEP], where [SEP] is the special token separating the two sequences and [CLS] is the special token necessary for BERT to encode the sequence pair into a 768-dimensional vector which will be fed in to the fully connected layers for classification [].

Given a collection of training pairs with labels $(e_1, e_2, y) \sim P$, a TLM is learned (typically by fine tuning an existing language model such as BERT with the training pairs provided) essentially implementing a function f mapping pairs of data entries to labels. We evaluate the $accuracy(f) = \mathrm{E}_{(e_1,e_2,y)\sim P} 1[f(e_1, e_2) = y)]$.

We will evaluate the impact that noise, injected on training and test data, has when deploying TLM for the EM problem. Next we will propose a data encoding technique that is applied both on training data before the TLM is fine tuned for EM as well as while inference. The encoding aims to mitigate the impact of noise in the EM process while using TLM. We will evaluate in detail the impact of increasing noise on a standard TLM adapted for EM [] as well as the impact that data augmentation and adversarial training have on the EM process.

3. NOISE RESISTANT ENCODING

Let D a collection of data entries of the form $e_j = (a_i, u_i^j)$, $1 \leq i \leq k$. Let $D(a_i)$ the domain of attribute a_i. An string attribute value u_i^j for entry e_j can be decomposed into a number of *tokens* $u_i^{jk}, 1 \leq k \leq L_j$, where L_j the number of tokens in attribute value u_i^j. A token could be a word, a qgram or any other predefined and semantically meaningful sub-string of u_i^j. We assume that in the training data any token can be perturbed in arbitrary ways. For example, in the simplest case, if a token is a natural language word it can be misspelled (essentially allowing character typos or transpositions). Similarly if a token is part of a product description (e.g., SLR, referring to lens types as part of a camera product description) it may be erroneously represented, etc.

Let e be a data entry and $\hat{e}$ a data entry with at least one token in one or more of the attribute values of e perturbed. We will refer to $\hat{e}$ as a perturbation of e. Our goal is to learn a TLM, f implementing a mapping between pairs of data entries to labels in a way that $accuracy(f) = \mathrm{E}_{(e_1,e_2,y)\sim P} 1[f(\hat{e_1}, \hat{e_2}) = y)]$ is high. Notice that $\hat{e_1}, \hat{e_2}$ are perturbations of e_1, e_2 for which $(e_1, e_2, y) \sim P$.

In order to achieve this, we aim to design *encoding functions* m_i for each attribute a_i, $1 \leq i \leq k$. Ideally. each encoding function should have the property of mapping possible perturbations of an attribute value (in which one or more of its tokens have been perturbed) into the same standardized representation. At the same time, the application of such encoding functions should not penalize prediction accuracy of the attribute value if not perturbed. Assuming that such encoding functions have been designed, our overall approach is to transform a data entry $e = (a_i, u_i)$ to $\mathcal{M}(e) = (a_i, m_i(u_i))$. The TLM is learned utilizing entries $(\mathcal{M}(e_1), \mathcal{M}(e_2), y)$. During inference we follow the same methodology applying $\mathcal{M}$ to each input entry before utilizing the TLM. Essentially, such encoding functions, should eliminate the impact of perturbations (noise) on the data entries being applied before fine tuning a TLM and before conducting inference.

There are two main properties that our design of the encoding functions m_i should possess. First, we desire that our encoding is *firm* namely that different perturbations of the same attribute value result to the same encoded representation. Second, we desire that the encoding is *loyal* namely that the encoded representations do not hamper prediction accuracy when there are no perturbations present in attribute values. The firmness and loyalty objectives are potentially conflicting. At one hand an encoding can map each attribute value to the same representation, thus achieving very high firmness and severely penalizing loyalty. Similarly, an encoding can map each attribute value to itself, thus achieving very high loyalty penalizing firmness.

We detail below how to derive *noise resistant encodings* for a single attribute[1]. The encodings are based on a preprocessing of the attributes values aiming to balance between loyalty and firmness in the set of attribute values present in the training data for each attribute. The TLM is build by fine tuning a pre-trained language model such as BERT [] utilizing the encoded data. At inference time, the data is encoded similarly before it is provided to the TLM as input.

3.1 Encoding Design

For a data entry $e_j = (a_i, u_i^j)$, $1 \leq i \leq k$, a string attribute value u_i^j is decomposed into a number of *tokens* $u_i^{jk} 1 \leq k \leq L_j$, where L_j the number of tokens in attribute value u_i^j. The encoding function m_i encodes e_j by encoding each individual token separately. Essentially m_i is defined by a token level function ρ, mapping each token $u_i^{jk}, 1 \leq k \leq L_j$ to $\rho(u_i^{jk})$, for each k. Thus, $m_i(u_i^j) = (\rho(u_i^{j1}), \ldots, \rho(u_i^{jL_j}))$. As a result the specification of $\rho()$ is important as it ultimately decide which tokens and associated perturbations have the same representation (encoding).

Let $\mathcal{U}$ be the set of unique tokens among all attribute values present in attribute a_i. We view the derivation of the encoding function $\rho()$ as a clustering problem. More specifically we seek to determine a number of clusters $C_1, \ldots C_c$, each a subset of U, such that all tokens belonging to a cluster share the same encoded representation. We will choose one cluster member (token) for each cluster to represent the encoded representation of the tokens in this cluster. To fully specify ρ, given c, we need to determine the contents of each cluster C_i from U, but also decide how to assign each token $u \in \mathcal{U}$ to a cluster. The encoding should balance between the properties of firmness and loyalty which we detail below.

Firmness: To measure firmness of the encoding, we have to quantify how frequently perturbations of the same token map to different clusters. For a token u let $\mathcal{T}(u)$ be the set of all allowable perturbations of u. Let $\mathcal{T}_\rho(u)$ the set of all encodings $\rho(\hat{u})$ for each perturbation $\hat{u} \in \mathcal{T}(u)$. The

[1]Similarly such encodings can be derived for any group of attributes jointly based on semantics or domain knowledge, for example grouping attributes knowing that noise is jointly present

firmness of a clustering $C = C_1 \dots C_c$ of U is defined as:

$$Firm(C) = -\sum_{i=1}^{|\mathcal{U}|} fr(u_i)|\mathcal{T}_\rho(u_i)| \quad (1)$$

where $fr(u_i)$ is the frequency of token u_i among all tokens in the attribute values of attribute a_i. Let $dist()$ be any similarity measure/distance function quantifying the closeness of two tokens. Typical measures include, edit distance and Jaccard coefficient, but numerous other measures are applicable []. Let $u_1, u_2 \in U$ be two tokens with $fr(u_1) > fr(u_2)$ and assume that they have been assigned to different clusters. Let u be a perturbation of both u_1 and u_2. That means that for a given function $dist()$, we have $dist(u_1, u_2) < \lambda$ for a suitably chosen threshold λ. For example if $u_1 =' excess'$ and $u_2 =' express$, then may be $u =' exess'$. In this case, assuming $dist$ is edit distance, we have $dist(u_1, u_2) < 2$. If we assign u to the cluster that u_1 belongs to then $|\mathcal{T}_\rho(u_2)|$ increases and vice-versa (since token u is a perturbation of both tokens u_1, u_2). We aim to maximize equation 1 and for this reason we choose to assign u to the same cluster as its most frequent neighbor (according to $dist$) belongs to, namely u_1.

Loyalty: Loyalty of the encoding is maximized when each token belongs to its own cluster. Since we have to work with a fixed number of clusters, loyalty will be compromised as various tokens will be assigned to the same cluster. We will have to determine how to assign tokens to clusters in a way that we minimize the negative impact on loyalty. Given a specific cluster C_i that already contains a number of tokens, let u be the cluster representative, namely the token that is utilized as the encoding for all members of the cluster. If we wish to augment this cluster with additional tokens, every time we add a token, loyalty is compromised. If we wish to minimize the impact on loyalty, it would make sense to include in the same cluster tokens u_i that have small distance to the cluster representative u. At the same time we would like to add tokens u_i with small $fr(u_i)$, since the impact of encoding u_i as u will be confined on fewer attribute values. Thus, it is evident that any measure of *loyalty* has to consider the frequency of tokens as well as their distance to the cluster representative. We define *Loyalty* of a clustering C as:

$$Loyal(C) = -\sum_{i=1}^{|\mathcal{U}|} fr(u_i) * dist(u_i, u_{c(i)}) \quad (2)$$

where u_i is a token and $c(i)$ is the index of the cluster u_i belongs to. Essentially we wish to stress that loyalty is high when frequent tokens are encoded together with less frequent tokens and is small when multiple frequent tokens are encoded jointly.

Clustering Objective: Since the clustering has to yield a specific number of clusters, we seek to optimize an objective that balances firmness and loyalty. More specifically we aim to optimize $O(C) = \alpha Loyal(C) + (1-\alpha)Firm(C)$. This objective aims to balance between these two measures utilizing a hyper-parameter α. For smaller values of α the overall objective favors loyalty, aiming to cluster tokens around the most frequent ones. As α increases, the objective aims to favor firmness, aiming to assign perturbations of tokens to clusters along with their most frequent neighbors.

3.2 The NRE Algorithm

3.2.1 Clustering Phase

We present a clustering algorithm for approximately maximizing our overall objective $O(C)$. Algorithm 1 presents the overall approach. We initialize a graph G such that it contains a vertex for each token in $\mathcal{U}$. We then examine each pair of vertices C_i, C_j; an edge is established (lines 3-5) by connecting C_i, C_j iff $dist(C_i, C_j) \leq \lambda$, where $dist()$ is any similarity measure/distance function applied on the tokens C_i, C_j and λ is the distance threshold. Notice that this initial state corresponds to initializing the encoding function $\rho(u) = u$ for each $u \in \mathcal{U}$, therefore, the Loyalty is maximized and Firmness is minimized. We start by considering each vertex in $\mathcal{U}$ as a cluster. We progressively merge vertices forming new clusters. We start clustering by examining every edge e. Let C_i, C_j be the two vertices connected by e, if replacing C_i, C_j by a new vertex $C_i \cup C_j$ improves $O(C)$, we commit this replacement (merging) in G. The modified graph G is returned after all edges have been examined. At that point each vertex in G is a cluster in C. The NRE algorithm is formally presented as Algorithm 1 (lines 10-21).

Algorithm 1: NRE algorithm

Input : $\mathcal{U}$, $dist()$, λ, α, $fr()$
Output: Graph G

1 $C \leftarrow \mathcal{U}$
2 $E \leftarrow \{\}$
3 **for** $C_i, C_j \in C$ **do**
4 **if** $dist(C_i, C_j) \leq \lambda$ **then**
5 $E \leftarrow E \cup \{C_i, C_j\}$
6 $G \leftarrow (C, E)$
7 $Firm(C)_{opt} \leftarrow Firmness(G, fr())$
8 $Loyal(C)_{opt} \leftarrow Loyalty(C, fr(), dist())$
9 $O(C)_{opt} \leftarrow \alpha Loyal(C)_{opt} + (1-\alpha)Firm(C)_{opt}$
10 **for** $e \in E$ **do**
11 $C_i, C_j \leftarrow$ two vertices connected by e
12 $Loyal(C)_{new} \leftarrow$ $Loyal(C)_{opt} - Loyalty(C_i, fr(), dist()) - Loyalty(C_j, fr(), dist()) + Loyalty(C_i \cup C_j, fr(), dist())$
13 $G_{Neighbors} \leftarrow$ a subgraph contains $C_i, C_j, Neighbors_{C_i}, Neighbors_{C_j}$
14 $G'_{Neighbors} \leftarrow$ replace C_i, C_j by $C_i \cup C_j$ in $G_{Neighbors}$
15 $Firm(C)_{new} \leftarrow$ $Firm(C)_{opt} - Firmness(G_{Neighbors}, fr()) + Firmness(G'_{Neighbors}, fr())$
16 $O(C)_{new} \leftarrow \alpha Loyal(C)_{new} + (1-\alpha)Firm(C)_{new}$
17 **if** $O(C)_{new} > O(C)_{opt}$ **then**
18 $G \leftarrow$ replace C_i, C_j by $C_i \cup C_j$ in G
19 $Loyal(C)_{opt} \leftarrow Loyal(C)_{new}$
20 $Firm(C)_{opt} \leftarrow Firm(C)_{new}$
21 $O(C)_{opt} \leftarrow O(C)_{new}$
22 **return** G

The crucial part in the NRE algorithm is to efficiently compute the two objectives $Loyal(C)$ and $Firm(C)$ for a fixed clustering C. In the initial state, each vertex/cluster in G contains only one token so the cluster representative is the token itself. It follows that initially $Loyal(C)$ is zero. After merging a pairs of nodes, we have to pick a single clus-

Algorithm 2: Loyalty

Input : C, $fr()$, $dist()$
Output: $Loyal(C)$

```
1 Loyal(C) ← 0
2 for C_i ∈ C do
3 |   ClustRep ← arg max_t fr(t) for t ∈ C_i
4 |   for t ∈ C_i do
5 |   |   Loyal(C) ←
  |   |    Loyal(C) + fr(t) * dist(t, ClustRep)
6 return −1 × Loyal(C)
```

Algorithm 3: Firmness

Input : $G = (C, E)$, $fr()$
Output: $Firm(C)$

```
1 Firm(C) ← 0
2 for C_i ∈ C do
3 |   for t ∈ C_i do
4 |   |   HN_t ←
  |   |    a set of all cluster n ∈ Neighbors_{C_i} with ∃u ∈ n
  |   |    s.t. dist(u, t) ≤ λ and fr(u) ≥ fr(t)
5 |   |   Firm(C) ← Firm(C) + fr(t) * |HN_t|
6 return −1 × Firm(C)
```

ter representative and update the measure $Loyal(C)$. The maximization of equation 2 implies that one should choose the token with highest frequency in a cluster as the new cluster representative. Also, the measure $Loyal(C)$ equals to the sum of *loyalty* values of each cluster, therefore, we are able to update $Loyal(C)$ in constant time by subtracting the *loyalty* of the two old clusters and adding the *loyalty* of the new merged cluster. This is what Algorithm 1 line 12 does and Algorithm 2 provides the computation of *loyalty* of one or more clusters. Algorithm 2, presents the computation of *loyalty* and the designation of the cluster representative for the cluster.

For a token u and an encoding function ρ, $\mathcal{T}(u)$ is the set of all allowable perturbations of u and $\mathcal{T}_\rho(u)$ is the set of all encodings $\rho(\hat{u})$ for each perturbation $\hat{u} \in \mathcal{T}(u)$. The size of $\mathcal{T}_\rho(u)$ ($|\mathcal{T}_\rho(u)|$) dictates the $Firm$ value of the encoding function ρ at u. We want ρ to have higher $Firm$ value at tokens with higher frequency while tolerate ρ to have a smaller $Firm$ value at tokens with lower frequency. $|\mathcal{T}_\rho(u_i)|$ can be computed efficiently utilizing graph G. Suppose $u_i \in C_i$ where u_i is a token and C_i is the cluster that u_i belongs to. Then if there exists a u_j such that $fr(u_j) > fr(u_i)$ and $dist(u_i, u_j) \leq \lambda$, this u_j can only appear in C_i or the neighbors of C_i. Therefore, we set $|\mathcal{T}_\rho(u_i)|$ to be the number of C_i's neighbors that contain contain such u_j. This is what Algorithm 3 and it runs in linear time (computing $Firm(C)$) for a fixed clustering C. After merging a pair of nodes, we have to update $Firm(C)$. Since the value of $|\mathcal{T}_\rho(u_i)|$ is only affected by the neighboring clusters of u_i, we are able to update $Firm(C)$ in constant time by subtracting the $Firm$ value of every token in the subgraph which contains two merged clusters and their neighbors, followed by adding the $firmness$ of every token in the new merged subgraph (Algorithm 1 line 13 - 15).

The overall run time of the NRE algorithm (Algorithm 1) is $\mathcal{O}(|\mathcal{U}|^2)$ since edges are build in $\mathcal{O}(|\mathcal{U}|^2)$ (lines 3 - 5), computing the initial objectives is conducted in $\mathcal{O}(|\mathcal{U}|)$ (line 7 - 9). All objectives are updated in constant time at each of the iterations over $|E|$ edges (line 10 - 21).

3.2.2 Encoding Phase

During the encoding phase, we aim, given a tuple pair, to encode it utilizing the clusters that we obtain from the application of algorithm NRE, before we route the encoded pair downstream for further processing by the TLM both during training as well as testing. Our overall approach during training is to transform each data entry $e = (a_i, u_i)$ to $\mathcal{M}(e) = (a_i, m_i(u_i))$, $1 \leq i \leq k$. Then train the TLM using the all (encoded) training pairs and associated labels, $(\mathcal{M}(e_1), \mathcal{M}(e_2), y)$. After the TLM has been trained, during inference, we first encode the provided pairs of entries, before providing it to the TLM for inference.

We now describe how to encode each corresponding attribute value utilizing the computed clustering. Suppose u is a token of attribute value a_i in e_1. The token level encoding function ρ accepts u as its input. If $u \in \mathcal{U}_{a_i}$, ρ outputs the cluster representative of the cluster u belongs to. Notice that if u was part of the training data, it may be a perturbation of some token. Since we expect that perturbations will have lower frequency in the training data, NRE would have assigned it to a cluster already. As a result, it will have a suitable encoded representation by the cluster representative.

If $u \notin \mathcal{U}_{a_i}$, we aim to identify a $u' \in \mathcal{U}_{a_i}$, for which $dist(u, u') \leq \lambda$. If such a u' is identified, we encode u utilizing the cluster representative of u'. If such a u' does not exist, we examine the corresponding attribute $a_i^{'}$ in data entry e_2 to determine if $u \in \mathcal{U}_{a_i^{'}}$ and if true we encode u using corresponding cluster representative. Failing that we seek $u' \in \mathcal{U}_{a_i^{'}}$ such that $dist(u, u') \leq \lambda$ and if such u' exists we encode u using the cluster representative for u'. Failing all these we maintain u as is. As a result encoding a token can be conducted in $\mathcal{O}(|\mathcal{U}|)$.

4. EXPERIMENTAL EVALUATION

In this section we present the results of a thorough experimental study evaluating the proposed NRE algorithm for data encoding, along other applicable approaches. We utilize the recently proposed TLM for EM, named *Ditto* [] as the main architecture and utilize the implementation publicly available for the authors []. In our study we evaluate all techniques using publicly available data sets that have been utilized previously in EM works utilizing Deep Networks [].

All of our experiments are conducted utilizing *Ditto* that in turns utilizes BERT []. Task specific layers (linear followed by softmax) are attached to the end of Bert to make binary predictions. We activate all three optimization techniques provided in *Ditto*. Specifically, (1) Span Typing domain knowledge is injected by an open-source Named Entity Recognition model []. (2) MixDA Data augmentation [] is used with $\lambda = 0.9$. (3) The maximum sequence length that BERT can access at input is 512 characters. For all datasets we used, there is no data entry longer than 512 words so we can directly input them to Bert. We set the maximum sequence length to be 256 in BERT. The training process runs a sufficient number of epochs (10, 20, or 80 depending on the dataset size) until convergence. For the case of

the *Company* data set (see below in data set description) the number of tokens is greater than 512. For this data set we use a TF-IDF-based summarization to reduce each of them to the maximum length allowed by BERT, setting the maximum sequence length to be 512. We conduct all experiments on a machine with 8 cores Intel Xeon Gold 5122 CPU at 3.60GHz, 128 GB DDR4 memory at 2666MHz, and 2 Titan Xp GPUs.

4.1 Noise Model

We focus on the analysis of the impact of noise on the EM process. There are numerous ways that noise can manifest in data entries. A robust way to model noise is as random edit errors on a data entry. It is easy to vary noise as a percentage of the entry length (in tokens) that have been modified. We assume that the noise model is uniform and random (each character has the same probability to be changed). When a character is changed, one of four operations is randomly applied. They are **permutation** of existing neighbor character, **substitution** of another randomly selected character, **deletion** of the character, and **insertion** of a randomly selected character beside it. Notice that this model is on purpose different than a model of typographical errors []. Since typographical errors take place due to mistyping on a keyboard, the errors are fairly correlated and depend on the relative position of neighboring characters on the keyboard. A random noise model is more general as we are interested on the impact of any possible error on any character and any position and not necessarily mistyping error only. We will include typographical error correction as part of our evaluation as well [].

4.2 Algorithms Compared

We include the following algorithm in our evaluation in addition to NRE.

- **Data Augmentation:** This is a well establish technique in training deep neural architectures [] and it is applied in the case of TLM for EM. The original *Ditto* architecture conducts some data augmentation by default during training. Our data augmentation is independent from *Ditto*'s default data augmentation. We augment training examples utilizing our noise model to the training set. For pairs of entries in the training data set selected for data augmentation, we apply our noise model, injecting noise to a fraction of the tokens in the data pairs. The augmented data set is then supplied to Ditto.

- **Adversarial Training:** We first train *Ditto* on given data set. We then select examples from the data set and utilize the noise model to generate a set of adversarial examples (injecting noise to a fraction of the tokens of the training data pair). We then let *Ditto* predict each adversarial pair and collect those erroneously classified examples and further fine-tune (retrain) *Ditto* with those adversarial examples.

- **Typographical Error Correction:** We utilize a state-of-the-art typo-corrector ScRNN []. We transform our data into an input that ScRNN accepts, by serializing each data entry (concatenating its attribute values in a comma separated representation). We then re-train a ScRNN typo-corrector utilizing the serialized data. ScRNN accepts the input and applies a set of perturbations on the training examples. At test time, the serialized test examples are corrected by ScRNN typo-corrector and converted back to the suitable data format to provide input for Ditto. We then enable *Ditto* trained on the clean training set to conduct testing.

  When ScRNN receives a token that is not a part of its training set, it will output UNK (also known as UNK-predictions). To handle UNK-predictions, we adopt Pass-through backoff (leave the token unchanged) for the Company data set and Backoff to a neutral word (replace the token by a neutral word like 'a') for other data sets since we empirically found that this setting achieves best results for this technique.

4.3 Experimental Results

We utilized numerous real data sets with diverse characteristics all previously utilized extensively in studies related to the EM problem utilizing deep neural architectures []. Since NRE processes each attribute domain in the training set in its entirety to derive encodings, we conduct data augmentation, adversarial training as well as typographical error correction, utilizing each tuple of the training data set.

Noise is injected in the tuples of the training data set. For each data set utilized in our experiments, we compute the average number of tokens across all data pairs. Then we vary the percentage of errors injected in tuples pairs (relative to the average number of tokens) so we can modify the amount of noise in a methodological way. The noise injected data pairs in each case, enhance the original training data set with additional data pairs, yielding an enhance training set. Ditto is fully re-trained in each experiment with the enhance data set, as we vary the percentage of errors injected.

We utilize *edit distance* as the *dist*() function between tokens in NRE during our experiments. We also deployed other functions (e.g., Jaccard similarity) and the results are qualitatively the same. In each case we vary the threshold λ. We treat parameter α in NRE as a hyper-parameter which is optimized using grid search.

We report the $F1$ score utilizing test data. We vary the percentage of entry pairs that errors have been injected and observe the overall accuracy score in each algorithm. In each case we utilize the same level of noise (e.g., 5% of the average number of tokens in the pairs of the training data set) in both the training data and test data in each experiment.

Dataset	K	α	% perturbed tokens	F1-score				
				Original *Ditto*	DA	AT	Typo Corrector	NRE
Structured iTunes-Amazon	1	1	0.00	93.10	91.53	94.74	89.80	92.31
			9.94	89.23	90.40	91.04	86.64	**92.15**
			19.87	79.93	89.50	86.47	83.75	**91.55**
			29.81	74.85	83.95	81.46	77.69	**89.96**
			39.74	70.53	77.76	73.89	72.28	**88.52**
			49.68	61.00	73.36	68.59	64.56	**86.45**
Structured Beer	1	0.7	0.00	84.85	77.78	82.35	69.23	82.35
			13.22	78.14	75.46	80.80	63.16	**81.94**
			26.43	63.33	72.73	66.75	57.97	**80.57**
			39.65	51.76	67.03	60.28	53.65	**76.22**
			52.87	44.74	61.10	53.88	50.37	**75.93**
			66.09	33.65	54.52	46.64	48.05	**74.40**
Structured Walmart-Amazon	1	1	0.00	85.87	81.20	78.86	58.76	76.12
			9.59	78.56	79.79	78.37	56.13	74.00
			19.19	66.62	72.78	72.73	54.79	72.66
			28.78	56.11	69.53	68.61	53.64	**72.59**
			38.37	45.20	63.71	65.19	53.20	**70.95**
			47.97	36.28	56.56	62.46	52.59	**68.69**
Structured Fodors-Zagat	1	0	0.00	97.78	97.78	97.78	86.36	97.67
			11.27	96.28	97.06	97.33	84.30	**97.43**
			22.55	88.25	92.72	97.06	77.60	**97.42**
			33.82	81.61	87.52	93.90	73.98	**97.19**
			45.09	69.34	77.58	89.71	68.64	**96.94**
			56.37	54.98	65.13	79.42	61.92	**94.41**
Structured Amazon-Google	1	0.95	0.00	71.36	66.51	73.52	66.67	70.66
			5.48	59.48	64.28	69.74	64.71	69.01
			10.97	49.21	60.28	66.18	62.39	**67.93**
			16.45	39.14	58.53	63.69	61.65	**67.45**
			21.93	32.34	53.60	53.64	57.69	**64.70**
			27.41	23.99	51.65	49.12	57.08	**63.95**
Structured DBLP-ACM	1	0.4	0.00	98.43	98.65	97.57	96.81	97.77
			4.79	94.32	98.20	97.42	92.70	97.65
			9.57	87.34	97.53	97.09	90.32	97.46
			14.36	76.88	96.56	95.76	88.54	**96.95**
			19.15	63.84	93.36	94.19	88.07	**96.73**
			23.94	51.21	88.50	91.04	84.78	**95.71**
Structured DBLP-Scholar	1	0.2	0.00	95.00	94.81	94.48	93.63	94.42
			2.76	93.20	94.36	93.73	93.30	94.32
			5.51	90.09	94.21	92.40	92.60	**94.25**
			8.27	86.38	93.74	90.89	91.71	**94.18**
			11.02	82.39	93.09	89.97	91.05	**93.97**
			13.78	77.45	92.22	89.47	90.68	**93.92**
Dirty iTunes-Amazon	1	1	0.00	90.00	91.53	90.92	89.80	88.52
			9.94	86.81	88.45	88.88	84.72	88.15
			19.87	74.50	83.74	84.12	77.69	**87.71**
			29.81	66.32	76.51	79.34	74.90	**86.25**
			39.74	60.12	72.90	72.17	69.68	**85.65**
			49.68	53.63	70.25	67.53	63.19	**85.20**
Dirty Walmart-Amazon	1	1	0.00	79.37	77.15	75.61	54.05	70.82
			9.59	70.85	74.10	73.83	52.06	68.95
			19.19	63.70	69.64	71.10	51.17	68.89
			28.78	55.73	62.82	67.95	49.88	67.42
			38.37	44.02	54.29	61.92	48.26	**65.99**
			47.97	35.85	46.67	57.32	46.16	**65.02**
Dirty DBLP-ACM	1	0.4	0.00	97.72	97.67	97.34	95.08	97.52
			4.79	93.94	97.37	97.10	92.18	97.29
			9.57	84.38	97.28	96.96	90.04	97.05
			14.36	71.58	95.10	95.45	88.31	**96.61**
			19.15	56.56	92.50	93.96	87.73	**96.06**
			23.94	42.43	88.00	90.82	83.68	**95.48**
Dirty DBLP-Scholar	1	0.2	0.00	94.79	94.74	94.28	93.38	94.39
			2.76	93.15	94.17	93.63	93.07	**94.19**
			5.51	90.05	94.07	92.25	92.51	**94.10**
			8.27	86.09	93.60	90.71	91.23	**93.83**
			11.02	81.79	92.36	89.62	90.81	**93.67**
			13.78	77.19	92.06	89.32	90.45	**93.33**
Textual Abt-Buy	1	0.2	0.00	84.42	82.61	82.33	80.10	82.65
			6.08	82.27	81.42	81.09	78.01	**82.36**
			12.16	78.60	79.19	80.15	77.98	**81.40**
			18.24	74.41	77.43	80.06	77.28	**80.94**
			24.32	72.44	75.41	78.59	77.11	**80.65**
			30.40	68.07	72.69	76.28	76.01	**79.82**
Textual Company	1	0.85	0.00	87.42	84.57	86.81	86.48	85.03
			5.11	85.08	83.21	85.07	85.33	84.59
			10.22	82.40	81.29	83.11	83.82	**84.24**
			15.32	79.63	79.37	80.02	82.36	**84.08**
			20.43	76.43	77.00	77.49	80.23	**83.34**
			25.54	73.66	74.21	74.65	78.69	**82.70**