ERBlox: Combining Matching Dependencies with Machine Learning for Entity Resolution

Zeinab Bahmani[1], **Leopoldo Bertossi**[1] and **Nikolaos Vasiloglou**[2]

[1] Carleton University, School of Computer Science, Ottawa, Canada
[2] LogicBlox Inc., Atlanta, GA 30309, USA

Abstract. Entity resolution (ER), an important and common data cleaning problem, is about detecting data duplicate representations for the same external entities, and merging them into single representations. Relatively recently, declarative rules called *matching dependencies* (MDs) have been proposed for specifying similarity conditions under which attribute values in database records are merged. In this work we show the process and the benefits of integrating three components of ER: (a) Classifiers for duplicate/non-duplicate record pairs built using machine learning (ML) techniques, (b) MDs for supporting both the blocking phase of ML and the merge itself; and (c) The use of the declarative language *LogiQL* -an extended form of Datalog supported by the *LogicBlox* platform- for data processing, and the specification and enforcement of MDs.

Keywords: Entity resolution, matching dependencies, support-vector machines, classification, Datalog

1 Introduction

Entity resolution (ER) is a common and difficult problem in data cleaning that has to do with handling unintended multiple representations in a database of the same external objects. Multiple representations lead to uncertainty in data and the problem of managing it. Cleaning the database reduces uncertainty. In more precise terms, ER is about the identification and fusion of database records (think of rows or tuples in tables) that represent the same real-world entity [8, 15]. As a consequence, ER usually goes through two main consecutive phases: (a) detecting duplicates, and (b) merging them into single representations.

For duplicate detection, one must first analyze multiple pairs of records, comparing the two records in them, and discriminating between: *pairs of duplicate records* and *pairs of non-duplicate records*. This classification problem is approached with machine learning (ML) methods, to learn from previously known or already made classifications (a training set for supervised learning), building a *classification model* (a classifier) for deciding about other record pairs [10, 15].

In principle, in ER every two records (forming a pair) have to be compared, and then classified. Most of the work on applying ML to ER work at the record level [22, 10, 11], and only some of the attributes, or their features, i.e. numerical values associated to them, may be involved in duplicate detection. The choice of relevant sets of attributes and features is application dependent.

ER may be a task of quadratic complexity since it requires comparing every two records. To reduce the large number two-record comparisons, *blocking techniques* are

used [2, 19, 24]. Commonly, a single record attribute, or a combination of attributes, the so-called *blocking key*, is used to split the database records into blocks. Next, under the assumption that any two records in different blocks are unlikely to be duplicates, only every two records in a same block are compared for duplicate detection.

Although blocking will discard many record pairs that are obvious non-duplicates, some true duplicate pairs might be missed (by putting them in different blocks), due to errors or typographical variations in attribute values. More interestingly, similarity between blocking keys alone may fail to capture the relationships that naturally hold in the data and could be used for blocking. Thus, entity blocking based only on blocking key similarities may cause low recall. This is a major drawback of traditional blocking techniques.

In this work we consider different and coexisting entities. For each of them, there is a collection of records. Records for different entities may be related via attributes in common or referential constraints. Blocking can be performed on each of the participating entities, and the way records for an entity are placed in blocks may influence the way the records for another entity are assigned to blocks. This is called "collective blocking". Semantic information, in addition to that provided by blocking keys for single entities, can be used to state relationships between different entities and their corresponding similarity criteria. So, blocking decision making forms a collective and intertwined process involving several entities. In the end, the records for each individual entity will be placed in blocks associated to that entity.

Example 1. Consider two entities, Author and Paper. For each of them, there is a set of records (for all practical purposes, think of database tuples in a single table). For Author we have records of the form $\mathbf{a} = \langle name, \ldots, \mathit{affiliation}, \ldots, \mathit{paper\ title}, \ldots \rangle$, with {*name*, *affiliation*} the blocking key; and for Paper, records of the form $\mathbf{p} = \langle title, \ldots, author\ name, \ldots \rangle$, with *title* the blocking key. We want to group Author and Paper records at the same time, in an entwined process. We block together two Author entities on the basis of the similarities of authors' names and affiliations.

Assume that Author entities $\mathbf{a}_1, \mathbf{a}_2$ have similar names, but their affiliations are not. So, the two records would not be put in the same block. However, $\mathbf{a}_1, \mathbf{a}_2$ are authors of papers (in Paper records) $\mathbf{p_1}, \mathbf{p_2}$, resp., which have been put in the same block (of papers) on the basis of similarities of paper titles. In this case, additional *semantic knowledge* might specify that if two papers are in the same block, then corresponding Author records that have similar author names should be put in the same block too. Then, $\mathbf{a}_1$ and $\mathbf{a}_2$ would end up in the same block.

In this example, we are blocking Author and Paper entities, separately, but collectively and in interaction. ■

Collective blocking is based on blocking keys and *the enforcement* of semantic information about the *relational closeness* of entities Author and Paper, which is captured by a set of *matching dependencies* (MDs). So, we propose "MD-based collective blocking" (more on MDs right below).

After records are divided in blocks, the proper duplicate detection process starts, and is carried out by comparing every two records in a block, and classifying the pair as "duplicates" or "non-duplicates" using the trained ML model at hand. In the end,

records in duplicate pairs are considered to represent the same external entity, and have to be *merged* into a single representation, i.e. into a single record. This second phase is also application dependent. MDs were originally proposed to support this task.

Matching dependencies are declarative logical rules that tell us under what conditions of similarity between attribute values, any two records must have certain attribute values merged, i.e. made identical [16, 17]. For example, the MD

$$Dept_B[dept] \approx Dept_B[dept] \;\rightarrow\; Dept_B[city] \doteq Dept_B[city] \qquad (1)$$

tells us that for any two records for entity (or relation or table) $Dept_B$ that have similar values for attribute $dept$ attribute, their values for attribute $city$ should be matched, i.e. made the same.

MDs as introduced in [17] do not specify how to merge values. In [6, 7], MDs were extended with *matching functions* (MFs). For a data domain, an MF specifies how to assign a value in common to two values. We adopt MDs with MFs in this work. In the end, the enforcement of MDs with MFs should produce a duplicate-free instance (cf. Section 2 for more details).

MDs have to be specified in a declarative manner, and at some point enforced, by producing changes on the data. For this purpose, we use the *LogicBlox* platform, a data management system developed by the LogicBlox[1] company, that is centered around its declarative language, *LogiQL*. *LogiQL* supports relational data management and, among several other features [1], an extended form of Datalog with stratified negation [9]. This language is expressive enough for the kind of MDs considered in this work. [2]

In this paper, we describe our *ERBlox* system. It is built on top of the *LogicBlox* platform, and implements entity resolution (ER) applying to *LogiQL*, ML techniques, and the specification and enforcement of MDs. More specifically, *ERBlox* has three main components: (a) MD-based collective blocking, (b) ML-based duplicate detection, and (c) MD-based merging. The sets of MDs are fixed and different for the first and last components. In both cases, the set of MDs are *interaction-free* [7], which results, for each entity, in the unique set of blocks, and eventually into a single, duplicate-free instance [7]. We use *LogicQL* to declaratively implement the two MD-based components of *ERBlox*.

The blocking phase uses MDs to specify the blocking strategy. They express conditions in terms of blocking key similarities and also relational closeness (the semantic knowledge) to assign two records to a same block (by making the block identifiers identical). Then, under MD-based collective blocking different records of possibly several related entities are simultaneously assigned to blocks through the enforcement of MDs (cf. Section 5 for details).

On the ML side, the problem is about detecting pairs of duplicate records. The ML algorithm is trained using record-pairs known to be duplicates or non-duplicates. We independently used three established classification algorithms: *support vector machines* (SVMs) [25], *k-nearest neighbor* (K-NN) [14], and *non-parametric Bayes classifier* (NBC) [4]. We used the Ismion[3] implementations of them due to the in-house expertise

[1] www.logicblox.com

[2] For arbitrary sets of MDs, we need higher expressive power [7], such as that provided by answer set programming [3].

[3] http://www.ismion.com

at LogicBlox. Since the emphasis of this work is on the use of *LogiQL* and MDs, we will refer only to our use of SVMs.

We experimented with our *ERBlox* system using as dataset a snapshot of Microsoft Academic Search (MAS)[4] (as of January 2013) including 250K authors and 2.5M papers. It contains a training set. The experimental results show that our system improves ER accuracy over traditional blocking techniques [18], which we will call *standard* blocking, where just blocking-key similarities are used. Actually, MD-based collective blocking leads to higher precision and recall on the given datasets.

This paper is structured as follows. Section 2 introduces background on matching dependencies and their semantics, and SVMs. A general overview of the *ERBlox* system is presented in Section 3. The specific components of *ERBlox* are discussed in Sections 4, 5, and 6. Experimental results are shown in Section 7. Section 8 presents conclusions.

2 Preliminaries

2.1 Matching dependencies

We consider an application-dependent relational schema $\mathcal{R}$, with a data domain U. For an attribute A, Dom_A is its finite domain. We assume predicates do not share attributes, but different attributes may share a domain. An instance D for $\mathcal{R}$ is a finite set of ground atoms of the form $R(c_1, \ldots, c_n)$, with $R \in \mathcal{R}$, $c_i \in U$.

We assume that each entity is represented by a relational predicate, and its tuples or rows in its extension correspond to records for the entity. As in [7], we assume records have unique, fixed, global identifiers, *rids*, which are positive integers. This allows us to trace changes of attribute values in records. Record ids are placed in an extra attribute for $R \in \mathcal{R}$ that acts as a key. Then, records take the form $R(r, \bar{r})$, with r the rid, and $\bar{r} = (c_1, \ldots, c_n)$. Sometimes we leave rids implicit, and sometimes we use them to denote whole records: if r is a record identifier in instance D, $\bar{r}$ denotes the record in D identified by r. Similarly, if $\mathcal{A}$ is a sublist of the attributes of predicate R, then $r[\mathcal{A}]$ denotes the restriction of $\bar{r}$ to $\mathcal{A}$.

MDs are formulas of the form: $R_1[\bar{X}_1] \approx R_2[\bar{X}_2] \ \rightarrow \ R_1[\bar{Y}_1] \doteq R_2[\bar{Y}_2]$ [16, 17]. Here, $R_1, R_2 \in \mathcal{R}$ (and may be the same); and $\bar{X}_1, \bar{X}_2$ are lists of attribute names of the same length that are *pairwise comparable*, that is, X_1^i and X_2^i, and also $\bar{Y}_1, \bar{Y}_2$, share the same domain.[5] The MD says that, for every pair of tuples (one in relation R_1, the other in relation R_2) where the LHS is true, the attribute values in them on the RHS have to be made identical. Symbol $\approx$ denotes generic, reflexive, symmetric, and application/domain dependent similarity relations on shared attribute domains.

A *dynamic, chase-based semantics* for MDs with matching functions (MFs) was introduced in [7]. Given an initial instance D, the set Σ of MDs is iteratively enforced until they cannot be be applied any further, at which point a *resolved instance* has been produced. In order to *enforce* (the RHSs of) MDs, there are binary *matching functions* (MFs) $\mathtt{m}_A : Dom_A \times Dom_A \rightarrow Dom_A$; and $\mathtt{m}_A(a, a')$ is used to replace two values $a, a' \in Dom_A$ that have to be made identical. MFs are idempotent, commutative, and

[4] http://academic.research.microsoft.com. For comparison, we also tested our system with data from DBLP and Cora.

[5] A more precise notation for the MD would be: $\forall x_1^1 \cdots \forall y_2^m (\bigwedge_j R_1[x_1^j] \approx_j R_2[x_2^j] \ \longrightarrow \bigwedge_k R_1[y_1^k] \doteq R_2[y_2^k])$.

associative, and then induce a partial-order structure $\langle Dom_A, \preceq_A \rangle$, with: $a \preceq_A a' :\Leftrightarrow \mathtt{m}_A(a, a') = a'$ [6, 5]. It always holds: $a, a' \preceq_A \mathtt{m}_A(a, a')$. In this work, MFs are treated as built-in relations.

There may be several resolved instances for D and Σ. However, when (a) MFs are similarity-preserving (i.e., $a \approx a'$ implies $a \approx \mathtt{m}_A(a', a'')$); or (b) Σ is interaction-free (i.e., each attribute may appear in either the RHS or LHS of MDs in Σ), there is a unique resolved instance that is computable in polynomial time in $|D|$ [7].

2.2 Support vector machines

The SVMs technique [25] is a form of kernel-based learning. SVMs can be used for classifying vectors in an inner-product vector space $\mathcal{V}$ over $\mathbb{R}$. Vectors are classified in two classes, with a label in $\{0, 1\}$. The algorithm learns from a training set, say $\{(\mathbf{e}_1, f(\mathbf{e}_1)), (\mathbf{e}_2, f(\mathbf{e}_2)), (\mathbf{e}_3, f(\mathbf{e}_3)), \ldots, (\mathbf{e}_n, f(\mathbf{e}_n))\}$. Here, $\mathbf{e}_i \in \mathcal{V}$, and for the *feature* (function) f: $f(\mathbf{e}_i) \in \{0, 1\}$.

SVMs find an optimal hyperplane, $\mathcal{H}$, in $\mathcal{V}$ that separates the two classes where the training vectors are classified. Hyperplane $\mathcal{H}$ has an equation of the form $\mathbf{w} \bullet \mathbf{x} + b$, where $\bullet$ denotes the inner product, $\mathbf{x}$ is a vector variable, $\mathbf{w}$ is a weight vector of real values, and b is a real number. Now, a new vector $\mathbf{e}$ in $\mathcal{V}$ can be classified as positive or negative depending on the side of $\mathcal{H}$ it lies. This is determined by computing $h(\mathbf{e}) := sign(\mathbf{w} \bullet \mathbf{e} + b)$. If $h(\mathbf{e}) > 0$, $\mathbf{e}$ belongs to class 1; otherwise, to class 0.

It is possible to compute real numbers $\alpha_1, \ldots, \alpha_n$, such that the classifier h can be computed through: $h(\mathbf{e}) = sign(\sum_i \alpha_i \cdot f(\mathbf{e}_i) \cdot \mathbf{e}_i \bullet \mathbf{e} + b)$ (cf. Figure 3).

3 Overview of *ERBlox*

A high-level description of the components of *ERBlox* is given in Figure 1. It shows the workflow supported by *ERBlox* when doing ER. *ERBlox*'s three main components are: (1) MD-based collective blocking (path $\mathbf{1}, \mathbf{3}, \mathbf{5}, \{\mathbf{6}, \mathbf{8}\}$), (2) ML-based record duplicate detection (the whole initial workflow up to task $\mathbf{13}$, inclusive), and (3) MD-based merging (path $\mathbf{14}, \mathbf{15}$). In the figure, all the boxes in light grey are supported by *LogiQL*. As just done, in the rest of this section, numbers in boldface refer to the edges in this figure.

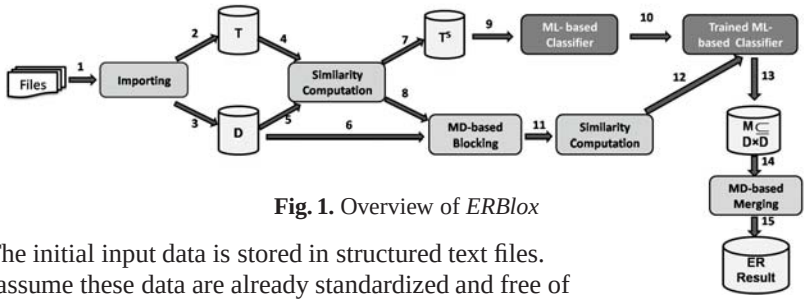

Fig. 1. Overview of *ERBlox*

The initial input data is stored in structured text files. (We assume these data are already standardized and free of misspellings, etc., but duplicates may be present.) Our general *LogiQL* program that supports the whole workflow contains some rules for importing data from the files into the extensions of relational predicates (think of tables, this is edge $\mathbf{1}$). This results in a relational database instance T containing the training data (edge $\mathbf{2}$), and the instance D on which ER will be performed (edge $\mathbf{3}$).

The next main task is blocking, which requires similarity computation of pairs of records in D (edge $\mathbf{5}$). For record pairs $\langle r_1, r_2\rangle$ in T, similarities have to be computed as well (edge $\mathbf{4}$). Similarity computation is based on similarity functions, Sf_i : $Dom_{A_i} \times Dom_{A_i} \rightarrow [0, 1]$, each of which assigns a numerical value, called similarity weight, to the comparisons of values for a record attribute A_i (from a pre-chosen subset of attributes) (cf. Figure 2). A weight vector $w(r_1, r_2) = \langle \cdots, Sf_i(r_1[A_i], r_2[A_i]), \cdots \rangle$ is formed by similarity weights (edge $\mathbf{7}$). For more details on similarity computation see Section 4.

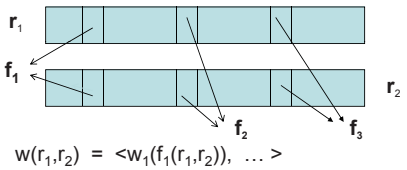

Fig. 2. Feature-based similarity

Since some pairs in T are considered to be duplicates and others non-duplicates, the result of this process leads to a "similarity-enhanced" database T^s of tuples of the form $\langle r_1, r_2, w(r_1, r_2), L\rangle$, with label $L \in \{0, 1\}$ indicating if the two records are duplicates ($L = 1$) or not ($L = 0$). The labels are consistent with the corresponding weight vectors. The classifier is trained using T^s, leading to a classification model (edges $\mathbf{9}, \mathbf{10}$).

For records in D, similarity measures are needed for blocking, to decide if two records r_1, r_2 go to the same block. Initially, every record has its rid assigned as block (number). To assign two records to the same block, we use matching dependencies that specify and enforce (through their RHSs) that their blocks have to be identical. This happens when certain similarities between pairs of attribute values appearing in the LHSs of the MDs hold. For this reason, similarity computation is also needed before blocking (workflow $\mathbf{5}, \mathbf{6}, \mathbf{8}$). This similarity computation process is similar to the one for T. However, in the case of D, this does not lead directly to the same kind of weight vector computation. Instead, the computation of similarity measures is only for the similarity predicates appearing in the LHSs of the blocking-MDs. (So, as the evaluation of the LHS in (1) requires the computation of similarities for $dept$-string values.)

Notice that these blocking-MDs may capture semantic knowledge, so they could involve in their LHSs similarities of attribute values in records for different kinds of entities. For example, in relation to Example 1, there could be similarity comparisons involving attributes for entities Author and Paper, e.g.

$$Author(x_1, y_1, bl_1) \wedge Paper(y_1, z_1, bl_3) \wedge Author(x_2, y_2, bl_2) \wedge Paper(y_2, z_2, bl_4) \wedge x_1 \approx_1 x_2 \wedge z_1 \approx_2 z_2 \rightarrow bl_1 \doteq bl_2, \quad (2)$$

expressing that when the similarities on the LHS hold, the blocks bl_1, bl_2 have to be made identical.[6] The similarity comparison atoms on the LHS are considered to be true when the similarity values are above predefined thresholds (edges $\mathbf{5}, \mathbf{8}$).[7]

This is the *MD-based collective blocking* stage that results in database D enhanced with information about the blocks to which the records are assigned. Pairs of records with the same block form *candidate duplicate record pairs*, and any two records with different blocks are simply not tested as possible duplicates (of each other).

[6] These MDs are more general than those introduced in Section 2.1: they may contain regular database atoms, which are used to give context to the similarity atoms in the same antecedent.

[7] At this point, since all we want is to do blocking, and not yet decisions about duplicates, we could, in comparison with what is done with pairs in T, compute less similarity measures and and even with low thresholds.

After the records have been assigned to blocks, pairs of records $\langle r_1, r_2 \rangle$ in the same block are considered for the duplicate test. As this point we proceed as we did for T: the similarity vectors $w(r_1, r_2)$ have to be computed (edges $\mathbf{11}, \mathbf{12}$).[8] Next, tuples $\langle r_1, r_2, w(r_1, r_2) \rangle$ are used as input for the trained classification algorithm (edge $\mathbf{12}$).

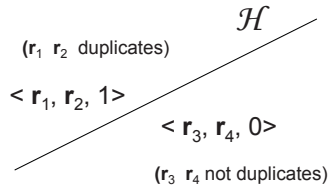

Fig. 3. Classification hyperplane

The result of the trained ML-based classifier, in this case obtained through SVMs as a separation hyperplane $\mathcal{H}$, is a set M of record pairs $\langle r_1, r_2, 1 \rangle$ that come from the same block and are considered to be duplicates (edge $\mathbf{13}$).[9] The records in these pairs will be merged on the basis of an *ad hoc* set of MDs (edge $\mathbf{15}$), different from those used in edges $\mathbf{6}, \mathbf{8}$.

Informally, the merge-MDs are of the form: $r_1 \approx r_2 \rightarrow r_1 \doteq r_2$, where the antecedent is true when $\langle r_1, r_2, 1 \rangle$ is an output of the classifier. The RHS is a shorthand for: $r_1[A_1] \doteq r_2[A_1] \wedge \cdots \wedge r_1[A_m] \doteq r_2[A_m]$, where m is the total number of record attributes. Merge at the attribute level uses the matching functions m_{A_i}.

We point out that MD-based merging takes care of transitive cases provided by the classifier, e.g. if it returns $\langle r_1, r_2, 1 \rangle$, $\langle r_2, r_3, 1 \rangle$, but not $\langle r_1, r_3, 1 \rangle$, we still merge r_1, r_3 (even when $r_1 \approx r_3$ does not hold). Actually, we do this by by merging all the records r_1, r_2, r_3 into the same record. Our system is capable of recognizing this situation and solving it as expected. This relies on the way we store and manage -via our *LogiQL* program- the positive cases obtained from the classifier (details can be found in Section 6). In essence, this makes our set of merging-MDs *interaction-free*, and leads to a unique resolved instance [7].

The following sections provide more details on *ERBlox* and our approach to ER.

4 Initial Data and Similarity Computation

We describe now some aspects of the MAS dataset, highlighting the input for- and output of each component of the *ERBlox* system. The data is represented and provided as follows. The Author relation contains authors names and their affiliations. The Paper relation contains paper titles, years, conference IDs, journal IDs, and keywords. The PaperAuthor relation contains papers IDs, authors IDs, authors names, and their affiliations. The Journal and Conference relations contain short names, full names, and home pages of journals and conferences, respectively. By using *ERBlox* on this dataset, we determine which papers in MAS data are written by a given author. This is clear case of ER since there are many authors who publish under several variations of their names. Also the same paper may appear under slightly different titles, etc.[10]

From the MAS dataset, which contains the data in structured files, extensions for intentional, relational predicates are computed by *LogiQL*-rules of the general program, e.g.

[8] Similarity computations are kept in appropriate program predicates. So similarity values computed before blocking can be reused at this stage, or whenever needed.

[9] The classifier also returns pairs or records that come from the same block, but are not considered to be duplicate. The set thereof in not interesting, at least as a workflow component.

[10] For our experiments, we independently used two other datasets: DBLP and Cora Citation.

Author	*AID*	*Name*	*Affiliation*	*Bl#*
	659	*Jean-Pierre Olivier de*	*Ecole des Hautes*	659
	2546	*Olivier de Sardan*	*Recherche Scientifique*	2546
	612	*Matthias Roeckl*	*German Aerospace Center*	612
	4994	*Matthias Roeckl*	*Institute of Communications*	4994

Paper	*PID*	*Title*	*Year*	*CID*	*JID*	*Keyword*	*Bl#*
	123	*Illness entities in West Africa*	1998	179		*West Africa, Illness*	123
	205	*Illness entities in Africa*	1998	179		*Africa, Illness*	205
	769	*DLR Simulation Environment m3*	2007	146		*Simulation m3*	769
	195	*DLR Simulation Environment*	2007	146		*Simulation*	195

PaperAuthor	*PID*	*AID*	*Name*	*Affiliation*
	123	659	*Jean-Pierre Olivier de*	*Ecole des Hautes*
	205	2546	*Olivier de Sardan*	*Recherche Scientifique*
	769	612	*Matthias Roeckl*	*German Aerospace Center*
	195	4994	*Matthias Roeckl*	*Institute of Communications*

Fig. 4. Relation extensions from MAS using LogiQL rules

$$\textit{\_file\_in}(x1, x2, x3) \rightarrow \textit{string}(x1), \textit{string}(x2), \textit{string}(x3). \quad (3)$$

$$\textit{lang} : \textit{physical} : \textit{filePath}[\text{`}\textit{\_file\_in}] = \text{"}\textit{author.csv}\text{"}. \quad (4)$$

$$+\textit{author}(id1, x2, x3) \leftarrow \textit{\_file\_in}(x1, x2, x3), \textit{string} : int64 : \textit{convert}[x1] = id1. \quad (5)$$

Here, (3) is a predicate schema declaration (metadata uses "$\rightarrow$"), in this case of the "_file_in" predicate with three string-valued attributes,[11] which is used to store the contents extracted from the source file, whose path is specified by (4). Derivation rules, such as (5), use the usual "$\leftarrow$". In this case, it defines the author predicate, and the "$+$" in the rule head inserts the data into the predicate extension. The first attribute is made an identifier [1]. Figure 4 illustrates a small part of the dataset obtained by importing data into the relational predicates. (There may be missing attributes values.)

As described above, in *ERBlox*, similarity computation generates similarity weights, which are used to: (a) compute the weight vectors for the training data T and the data in D under classification; and (b) do the blocking, where similarity weights are compared with predefined thresholds for the similarity conditions in the LHSs of blocking-MDs. [12]

We used three well-known similarity functions [13], depending on the attribute domains. "TF-IDF cosine similarity" [23] used for computing similarities for text-valued attributes, whose values are string vectors. It assigns low weights to frequent strings and high weights to rare strings. It was used for attribute values that contain frequent strings, such as affiliation. For attributes with short string values, such as author name, we applied "Jaro-Winkler similarity" [26]. Finally, for numerical attributes, such as publication year, we used "Levenshtein distance" [21], which computes similarity of two numbers on the basis of the minimum number of operations required to transform one into the other.

Similarity computation for *ERBlox* is supported by *LogiQL*-rules that define similarity functions. In particular, similarity computations are kept in extensions of program predicates. For example, if the similarity weight of values a_1, a_2 for attribute $Title$ is above the threshold, a tuple $TitleSim(a_1, a_2)$ is created by the program.

[11] In *LogiQL*, each predicate has to be declared, unless it can be inferred from the rest of the program.

[12] As described at the end of Section 3, these similarity computations are *not* used with the MDs that support the final merging process (cf. Section 6).

5 MD-Based Collective Blocking and Duplicate Detection

Since every record has an identifier, *rid*, initially each record uses its *rid* as its block number, in an extra attribute $Bl\#$. In this way, we create the *initial blocking instance* from the initial instance D, also denoted with D. Now, blocking strategies are captured by means of (blocking) MDs of the form:

$$R_i(\bar{X}_1, Bl_1) \wedge R_i(\bar{X}_2, Bl_2) \wedge \psi(\bar{X}_3) \;\rightarrow\; Bl_1 \doteq Bl_2. \tag{6}$$

Here Bl_1, Bl_2 are variables for block numbers, and R_i is a database (record) predicate. The lists of variables $\bar{X}_1, \bar{X}_2$ stand for all the attributes in R_i, but $Bl\#$. Formula ψ is a conjunction of relational atoms and comparison atoms via similarity predicates; but it does not contain similarity comparisons of blocking numbers, such as $Bl_3 \approx Bl_4$.[13] The variables in the list $\bar{X}_3$ appear in R_i or in another database predicate or in a similarity atom. It holds that $(\bar{X}_1 \cup \bar{X}_2) \cap \bar{X}_3 \neq \emptyset$. For an example, see (2), where R_i is Author.

In order to enforce these MDs on two records, we use a binary matching function $m_{Bl\#}$, to make two block numbers identical: $m_{Bl\#}(i, j) := i$ if $j \leq i$. More generally, for the application-dependent set, Σ^{Bl}, of blocking-MDs we adopt the chase-based semantics for entity resolution [7]. Since this set of MDs is interaction-free, its enforcement results in a single instance D^{Bl}, where now records may share block numbers, in which case they belong to the the same block. Every record is assigned to a single block.

Example 2. These are some of the blocking-MDs used for the MAS dataset:

$$Paper(pid_1, x_1, y_1, z_1, w_1, v_1, bl_1) \wedge Paper(pid_2, x_2, y_2, z_2, w_2, v_2, bl_2) \wedge x_1 \approx_{Title} x_2 \wedge y_1 = y_2 \wedge z_1 = z_2 \rightarrow bl_1 \doteq bl_2. \tag{7}$$

$$Author(aid_1, x_1, y_1, bl_1) \wedge Author(aid_2, x_2, y_2, bl_2) \wedge x_1 \approx_{Name} x_2 \wedge y_1 \approx_{Aff} y_2 \rightarrow bl_1 \doteq bl_2. \tag{8}$$

$$Paper(pid_1, x_1, y_1, z_1, w_1, v_1, bl_1) \wedge Paper(pid_2, x_2, y_2, z_2, w_2, v_2, bl_2) \wedge PaperAuthor(pid_1, aid_1, x_1', y_1') \wedge PaperAuthor(pid_2, aid_2, x_2', y_2') \wedge Author(aid_1, x_1', y_1', bl_3) \wedge Author(aid_2, x_2', y_2', bl_3) \wedge x_1 \approx_{Title} x_2 \rightarrow bl_1 \doteq bl_2. \tag{9}$$

$$Author(aid_1, x_1, y_1, bl_1) \wedge Author(aid_2, x_2, y_2, bl_2) \wedge x_1 \approx_{Name} x_2 \wedge PaperAuthor(pid_1, aid_1, x_1, y_1) \wedge PaperAuthor(pid_2, aid_2, x_2, y_2) \wedge Paper(pid_1, x_1', y_1', z_1', w_1', v_1', bl_3) \wedge Paper(pid_2, x_2', y_2', z_2', w_2', v_2', bl_3) \rightarrow bl_1 \doteq bl_2. \tag{10}$$

Informally, (7) tells us that, for every two Paper entities $\mathbf{p}_1, \mathbf{p}_2$ for which the values for attribute $Title$ are similar and with same publication year, conference ID, the values for attribute $Bl\#$ must be made the same. By (8), whenever there are similar values for name and affiliation in Author, the corresponding authors should be in the same block. Furthermore, (9) and (10) collectively block Paper and Author entities. For instance, (9) states that if two authors are in the same block, their papers $\mathbf{p}_1$, $\mathbf{p}_2$ having similar titles must be in the same block. Notice that if papers $\mathbf{p}_1$ and $\mathbf{p}_2$ have similar titles, but they do not have same publication year or conference ID, we cannot block them together using (7) alone. ■

We now show how these MDs are represented in *LogiQL*, and how we use *LogiQL* programs for declarative specification of MD-based collective blocking.[14] In *LogiQL*, an MD takes the form:

[13] Actually, this natural condition makes the set of blocking-MDs interaction-free, i.e. for every two blocking-MDs m_1, m_2, the set of attributes on the RHS of m_1 and the set of attributes on the LHS of m_2 on which there are similarity predicates, are disjoint [7].

[14] Notice that since we have interaction-free sets of blocking-MDs, stratified Datalog programs are expressive enough to express and enforce them [3]. *LogiQL* supports stratified Datalog.

$$R_i[\bar{X}_1]{=}Bl_2,\ \ R_i[\bar{X}_2]{=}Bl_2 \ \longleftarrow\ \ R_i[\bar{X}_1] = Bl_1,\ R_i[\bar{X}_2] = Bl_2,\ \psi(\bar{X}_3),\ Bl_1 < Bl_2, \tag{11}$$

subject to the same conditions as in (6). An atom $R_i[\bar{X}]{=}Bl$ states that predicate $R_i^{(11)}$ is functional on $\bar{X}$ [1]. It means each record in R_i can have only one block number $Bl\#$.

Given an initial instance D, a *LogiQL* program $\mathcal{P}^B(D)$ that specifies MD-based collective blocking contains the following (kind of) rules:

1. For every atom $R(rid, \bar{x}, bl) \in D$, the fact $R[rid, \bar{x}] = bl$. (Initially, $bl := rid$.)

2. For every attribute A of R_i, facts of the form *A-Sim*(a_1, a_2), with $a_1, a_2 \in Dom_A$, the finite attribute domain. They are obtained by similarity computation.

3. The blocking-MDs as in (11).

4. Rules to represent the consecutive versions of entities during MD-enforcement:

$$R\text{-}OldVersion(r_1, \bar{x}_1, bl_1) \ \leftarrow\ R[r_1, \bar{x}_1] = bl_1,\ R[r_1, \bar{x}_1] = bl_2,\ bl_1 < bl_2.$$

For each *rid*, r, there could be several atoms of the form $R[r, \bar{x}] = bl$, corresponding to the evolution of the record identified by r due to MD-enforcement. The rule specifies that versions of records with lower block numbers are old.

5. Rules that collect the latest versions of records. They are used to form blocks:

$$R\text{-}MDBlock[r_1, \bar{x}_1] = bl_1 \ \leftarrow\ R[r_1, \bar{x}_1] = bl_1,\ !\ R\text{-}OldVersion(r_1, \bar{x}_1, bl_1).$$

In *LogiQL*, "!", as in the body above, is used for negation [1]. The rule collects R-records that are not old versions.

Programs $\mathcal{P}^B(D)$ as above are stratified (there is no recursion involving negation). Then, as expected in relation to the blocking-MDs, they have a single model, which can be used to read the final block number for each record.

Example 3. (ex. 2 cont.) Considering only MDs (7) and (9), the portion of $\mathcal{P}^B(D)$ for blocking Paper entities has the following rules:

2. Facts such as: $TitleSim(Illness\ entities\ in\ West\ Africa, Illness\ entities\ in\ Africa).$
$TitleSim(DLR\ Simulation\ Environment\ m3, DLR\ Simulation\ Environment).$

3. $Paper[pid_1, x_1, y_1, z_1, w_1, v_1] = bl_2, Paper[pid_2, x_2, y_2, z_2, w_2, v_2] = bl_2 \ \leftarrow$
$Paper[pid_1, x_1, y_1, z_1, w_1, v_1] = bl_1, Paper[pid_2, x_2, y_2, z_2, w_2, v_2] = bl_2,$
$TitleSim(x_1, x_2), y_1 = y_2, z_1 = z_2, bl_1 < bl_2.$

$Paper[pid_1, x_1, y_1, z_1, w_1, v_1] = bl_2, Paper[pid_2, x_2, y_2, z_2, w_2, v_2] = bl_2 \ \leftarrow$
$Paper[pid_1, x_1, y_1, z_1, w_1, v_1] = bl_1, Paper[pid_2, x_2, y_2, z_2, w_2, v_2] = bl_2, TitleSim(x_1, x_2),$
$PaperAuthor(pid_1, aid_1, x_1', y_1'), PaperAuthor(pid_2, aid_2, x_2', y_2'),$
$Author[aid_1, x_1', y_1'] = bl_3, Author[aid_2, x_2', y_2'] = bl_3, bl_1 < bl_2.$

4. $PaperOldVersion(pid_1, x_1, y_1, z_1, w_1, v_1, bl_1) \leftarrow Paper[pid_1, x_1, y_1, z_1, w_1, v_1] = bl_1,$
$Paper[pid_1, x_1, y_1, z_1, w_1, v_1] = bl_2, bl_1 < bl_2.$

5. $PaperMDBlock[pid, \bar{x}_1] = bl_1 \ \leftarrow\ Paper[pid_1, x_1, y_1, z_1, w_1, v_1] = bl_1,$
$PaperOldVersion(pid_1, x_1, y_1, z_1, w_1, v_1, bl_1).$

Restricting the model of the program to the relevant attributes of predicate *PaperMD-Block* returns: $\{\{123, 205\}, \{195, 769\}\}$, i.e. the papers with *pids* 123 and 205 are blocked together; similarly for those with *pids* 195 and 769. ■

As described above, the input to the trained classifier is a set of tuples of the form $\langle r_1, r_2, w(r_1, r_2)\rangle$, with $w(r_1, r_2)$ the computed weight vector for records (with ids) r_1, r_2 in a same block.[15]

Example 4. (ex. 3 cont.) Consider the blocks for entity Paper. If the "journal ID" values are null in both records, but not the "conference ID" values, "journal ID" is not considered for a feature. Similarly, when the conference ID values are null. However, the values for "journal ID" and "conference ID" are replaced by "journal full name" and "conference full name" values, found in Conference and Journal records, resp. In this case then, attributes *Title*, *Year*, *ConfFullName* or *JourFullName* and *Keyword* are used for corresponding feature for weight vector computation.

Considering the previous Paper records, the input to the classifier consists of: $\langle 123,\ 205, w(123, 205)\rangle$, with $w(123, 205) = [0.8, 1.0, 1.0, 0.7]$, and $\langle 195, 769, w(195, 769)\rangle$, with $w(195, 769) = [0.93, 1.0, 1.0, 0.5]$ (actually the contents of the two square brackets only). ■

Several ML techniques are accessible from *LogicBlox* platform through the *BloxML-Pack* library, that provides a generic Datalog interface. Then, *ERBlox* can call an ML-based record duplicate detection component through the general *LogiQL* program. In this way, the SVMs package is invoked by *ERBlox*.

The output is a set of tuples of the form $\langle r_1, r_2, 1\rangle$ or $\langle r_1, r_2, 0\rangle$, where r_1, r_2 are ids for records of entity (table) R. In the former case, a tuple $R\text{-}Duplicate(r_1, r_2)$ is created (as defined by the *LogicQL* program). In the previous example, the SVMs method return $\langle[0.8, 1.0, 1.0, 0.7], 1\rangle$ and $\langle[0.93, 1.0, 1.0, 0.5], 1\rangle$, then $PaperDuplicate(123, 205)$ and $PaperDuplicate(195, 769)$ are created.

6 MD-Based Merging

When $EntityDuplicate(r_1, r_2)$ is created, the corresponding full records $\bar{r}_1, \bar{r}_2$ have to be merged via record-level merge-MDs of the form $R[r_1] \approx R[r_2] \longrightarrow R[\bar{r}_1] \doteq R[\bar{r}_2]$, where $R[r_1] \approx R[r_2]$ is true when $R\text{-}Duplicate(r_1, r_2)$ has been created according to the output of the SVMs classifier. The RHS means that the two records are merged into a new full record $\bar{r}$, with $\bar{r}[A_i] := m_{A_i}(\bar{r}_1[A_i], \bar{r}_2[A_i])$ [7].

Example 5. (ex. 4 cont.) We merge duplicate Paper entities enforcing the MD: $Paper\ [pid_1] \approx Paper[pid_2] \longrightarrow Paper[Title, Year, CID, Keyword] \doteq Paper[Title, Year, CID, Keyword]$. ■

The portion, $\mathcal{P}^M$, of the general *LogiQL* program that represents MD-based merging contains rules as in **1.-4.** below:

1. The atoms of the form $R\text{-}Duplicate$ mentioned above, and those representing the matching functions (MFs) m_A.

2. For an MD $R[r_1] \approx R[r_2] \longrightarrow R[\bar{r}_1] \doteq R[\bar{r}_2]$, the rule:

$$R[r_1, \bar{x}_3] = bl,\ R[r_2, \bar{x}_3] = bl \longleftarrow R\text{-}Duplicate(r_1, r_2),\ R[r_1, \bar{x}_1] = bl,\ R[r_2, \bar{x}_2] = bl,\ m(\bar{x}_1, \bar{x}_2) = \bar{x}_3,$$

[15] The features considered in a weight vector computation depend on whether they have a strong discrimination power, i.e. do not contain missing values.

which creates two records (one of them can be purged afterwards) with different ids but all the other attribute values the same, and computed componentwise according to the MFs for m. Here, $\bar{x}_1, \bar{x}_2, \bar{x}_3$ stand each for all attributes of relation R, except for the id and the block number (represented by bl). (Block numbers play no role in merging.)

3. As for program $\mathcal{P}^B(D)$ given in Section 5, rules specify the old versions of a record:

$$R\text{-}OldVersion(r_1, \bar{x}_1) \leftarrow R[r_1, \bar{x}_1] = bl,\ R[r_1, \bar{x}_2] = bl,\ \bar{x}_1 \prec \bar{x}_2.$$

Here, $\bar{x}_1$ stands for all attributes other than the id and the block number; and on the RHS $\bar{x}_1 \prec \bar{x}_2$ means componentwise comparison of values according to the partial orders defined by the MFs.

4. Finally, rules to collect the latest version of each record, building the final resolved instance: $R\text{-}ER(r_1, \bar{x}_1) \leftarrow R[r_1, \bar{x}_1] = bl,\ !\, R\text{-}OldVersion(r_1, \bar{x}_1)$.

Notice that the derived tables $R\text{-}Duplicate$ that appear in the LHSs of the MDs (or in the bodies of the corresponding rules) are all computed before (and kept fixed during) the enforcement of the merge-MDs. In particular, a duplicate relationship between any two records is not lost. This has the effect of making the set of merging-MDs interaction-free, which results in a unique resolved instance.

7 Experimental Evaluation

We now show that our approach to ER can improve accuracy in comparison with standard blocking. In addition to the MAS, we used datasets from DBLP and Cora Citation.

In order to emphasize the importance of semantic knowledge in blocking, we consider standard blocking and two different sets of MDs, (1) and (2), for MD-based collective blocking. Under (1), we define blocking-MDs for all the blocking keys used for standard blocking, but under (2) we have MDs for only some of the used blocking keys. In both cases, in addition to properly collective blocking MDs.

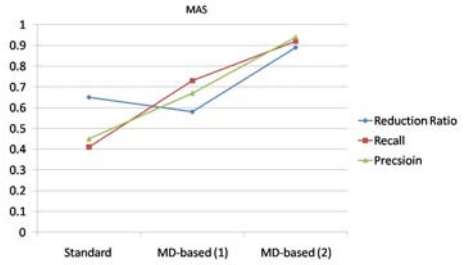

Fig. 5. The experiments (MAS)

We use three measures for the comparisons of blocking techniques. One is *reduction ratio*, which is the the ratio (minus 1) of the number of candidate record-pairs over the initial number of records. The higher this value, the less candidate record-pairs are being generated, but the quality of the generated candidate record pairs is not taken into account. We also use recall and precision measures. The former is the number of true duplicate candidate record-pairs divided by the number of true duplicate pairs, and precision is the number of true candidate duplicate record-pairs divided by the total number of candidate pairs [12].

Figures 5, 6 and 7 show the comparative performance of *ERBlox*. They show that standard blocking has higher reduction ratio than MD-based collective blocking version (1). This means that less candidate record-pairs are being generated by standard blocking. However, the precision and recall of MD-based blocking version (1) are higher than

standard blocking, meaning that MD-based blocking version (1) can lead to improved ER results at the cost of larger blocks, and thus more candidate record pairs that need to be compared.

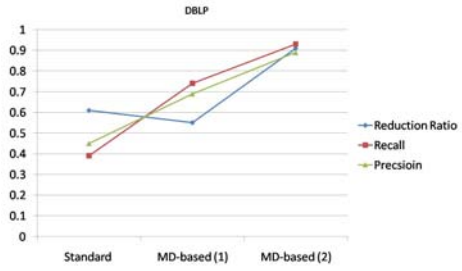

Fig. 6. The experiments (DBLP)

In blocking, this is a common trade-off that needs to be considered. On the one hand, having a large number of smaller blocks will result in fewer candidate record-pairs that will be generated, probably increasing the number of true duplicate record-pairs that are missed. On the other hand, blocking techniques that result in larger blocks generate a higher number of candidate record-pairs that will likely cover more true duplicate pairs, at the cost of having to compare more candidate pairs [12]. The experiments are all done before MD-based merging.

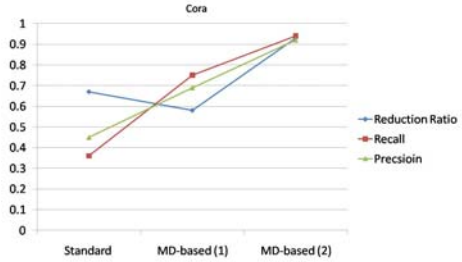

Fig. 7. The experiments (Cora)

Interestingly, MD-based blocking version (2) has higher reduction ratio, recall, and precision than standard blocking. This emphasizes the importance of MDs supporting collective blocking, and shows that blocking based on string similarity alone fails to capture the relationships that naturally hold in the data.

As expected, the experiments show that different sets of MDs for MD-based collective blocking have different impact on reduction ratio, so as standard blocking depends on the choice of blocking keys. However, the quality of MD-based collective blocking, in its two versions, dominates standard blocking for the three datasets.

8 Conclusions

We have shown that matching dependencies, a new class of data quality/cleaning semantic constraints in databases, can be profitably integrated with traditional ML-methods, in our case for entity resolution. They play a role not only in the intended goal of merging duplicate representations, but also in the record blocking process that precedes the learning task. At that stage they allow to declaratively capture semantic information that can be used to enrich the blocking activity. MDs declaration and enforcement, data processing in general, and machine learning can all be integrated using the *LogiQL* language.

Acknowledgments: Part of this research was funded by an NSERC Discovery grant and the NSERC Strategic Network on Business Intelligence (BIN). Z. Bahmani and L. Bertossi are very much grateful for the support from LogicBlox during their internship and sabbatical visit.

References

[1] Aref, M., ten Cate, B. , Green, T.J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T.L. and Washburn, G. Design and Implementation of the LogicBlox System. Proc. SIGMOD 2015, pp. 125-141.

[2] Baxter, R., Christen, P. and Churches, T. A Comparison of Fast Blocking Methods for Record Linkage. Proc. *ACM SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Identification* 2003, pp. 234-256.
[3] Bahmani, Z., Bertossi, L., Kolahi, S. and Lakshmanan, L. Declarative Entity Resolution via Matching Dependencies and Answer Set Programs. Proc. KR 2012, pp. 380-390.
[4] Baudat G. and Anouar, F. Generalized Discriminant Analysis using a Kernel Approach. *Neural Computation*, 2000, 12(3):2385-2404.
[5] Benjelloun, O., Garcia-Molina, H., Menestrina, D., Su, Q., EuijongWhang, S. and Widom, J. Swoosh: A Generic Approach to Entity Resolution. *VLDB Journal*, 2009, 18(1):255-276.
[6] Bertossi, L., Kolahi, S. and Lakshmanan, L. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. Proc. ICDT 2011, ACM Press, 2011.
[7] Bertossi, L., Kolahi, S. and Lakshmanan, L. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. *Th. Comp. Systems*, 2013, 52(3):441-482.
[8] Bleiholder, J. and Naumann, F. Data Fusion. *ACM Computing Surveys*, 2008, 41(1).
[9] Ceri, S., Gottlob, G. and Tanca, L. *Logic Programming and Databases*. Springer, 1989.
[10] Christen, P. and Goiser, K. Quality and Complexity Measures for Data Linkage and Deduplication. In *Quality Measures in Data Mining*, ser. Studies in Computational Intelligence, (Guillet, F. and Hamilton, H., Eds.), 2007, 43:127-151.
[11] Christen, P. Automatic Record Linkage using Seeded Nearest Neighbour and Support Vector Machine Classification. Proc. SIGKDD 2008, pp. 151-159.
[12] Christen, P. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Transactions in Knowledge and Data Engineering*, 2011, 19(1):1-16.
[13] Cohen, W., Ravikumar, P. and Fienberg, S. A Comparison of String Metrics for Matching Names and Records. Proc. *Workshop on Data Cleaning and Object Consolidation* 2003, pp. 123-134.
[14] Cover, T.M. and Hart, P.E. Nearest Neighbor Pattern Classification. *IEEE Transactions on Information Theory*, 1967, 13(1): 21-27.
[15] Elmagarmid, A., Ipeirotis, P. and Verykios, V. Duplicate Record Detection: a Survey. *IEEE Transactions in Knowledge and Data Engineering*, 2007, 19(1):1-16.
[16] Fan, W. Dependencies Revisited for Improving Data Quality. Proc. PODS 2008.
[17] Fan, W., Jia, X., Li, J. and Ma, S. Reasoning about Record Matching Rules. *PVLDB*, 2009, 2(1):407-418.
[18] Fellegi, I.P. and Sunter, A.B. A Theory for Record Linkage. *Journal of the American Statistical Society*, 1969, 64(1):328-339.
[19] Herzog, T.N., Scheuren,F.J. and Winkler, W.E. *Data Quality and Record Linkage Techniques*. Springer, 2007.
[20] Jaro, M.A. UNIMATCH: A Record Linkage System: User's Manual, Technical Report, U.S. Bureau of the Census, 1976.
[21] Navarro, G. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 2001, 33(1): 31-88.
[22] Rastogi, V., Dalvi, N.N. and Garofalakis, M.N. Large-scale Collective Entity Matching. *PVLDB*, 2011, 4(4):208-218.
[23] Salton, G. and Buckley, C. Term-weighting Approaches in Automatic Text Retrieval. *Information Processing and Management*, 1988, 24(5): 513-523.
[24] Euijong Whang, S., Menestrina, D., Koutrika, G., Theobald, M. and Garcia-Molina, H. Entity Resolution with Iterative Blocking. Proc. SIGMOD 2009, pp. 219-232.
[25] Vapnik, V.N. *Statistical Learning Theory*. Wiley, 1998.
[26] Winkler, W. E. The State of Record Linkage and Current Research Problems. Technical Report, U.S. Census Bureau, 1999.