MDedup: Duplicate Detection with Matching Dependencies

Ioannis Koumarelas Thorsten Papenbrock Felix Naumann
Hasso Plattner Institute, University of Potsdam, Germany
firstname.lastname@hpi.de

ABSTRACT

Duplicate detection is an integral part of data cleaning and serves to identify multiple representations of same real-world entities in (relational) datasets. Existing duplicate detection approaches are effective, but they are also hard to parameterize or require a lot of pre-labeled training data. Both parameterization and pre-labeling are at least domain-specific if not dataset-specific, which is a problem if a new dataset needs to be cleaned.

For this reason, we propose a novel, rule-based and fully automatic duplicate detection approach that is based on *matching dependencies (MDs)*. Our system uses automatically discovered MDs, various dataset features, and known gold standards to train a model that selects MDs as duplicate detection rules. Once trained, the model can select useful MDs for duplicate detection on any new dataset. To increase the generally low recall of MD-based data cleaning approaches, we propose an additional boosting step. Our experiments show that this approach reaches up to 94% F-measure and 100% precision on our evaluation datasets, which are good numbers considering that the system does not require domain or target data-specific configuration.

PVLDB Reference Format:
Ioannis Koumarelas, Thorsten Papenbrock, Felix Naumann. MDedup: Duplicate Detection with Matching Dependencies. *PVLDB*, 13(5): 712 - 725, 2020.
DOI: https://doi.org/10.14778/3377369.3377379

1. MD-BASED DUPLICATE DETECTION

Data cleaning is a multivariate process that identifies and repairs various issues in given datasets. Duplicates, which specify multiple representations of same real-world entities in a database, are among the most addressed and harmful data quality issues. Hence, their detection plays an important role in data cleaning processes. The general problem has been studied under different names, such as entity resolution, record linkage, and duplicate detection with slight variations in the application approach [1, 2]. Because duplicates are considered a very important problem with large negative implications if not treated properly, a plethora of approaches exist that effectively detect and merge duplicate records [1, 2]. A large portion of these approaches is based on machine learning (ML) techniques [2, 3]; several of them employ deep learning methods [4, 5]. Most other duplicate detection approaches are rule-based and follow systematic detection strategies. Irrespective of whether ML is used or not, all known solutions require either careful manual configuration by domain experts and/or exhaustive pre-labeling of training data, which are both difficult requirements that we want to avoid. To this end, we use automatically discovered *matching dependencies (MDs)* as rules and devise a system called *MDedup*, which automatically selects the best discovered MDs for the duplicate detection process, without needing any special domain-specific configuration or training data for the dataset at hand. Other, non-MD-based rule languages for duplicate detection, such as [6,7], allow to formulate more expressive rules than MDs, but these rules are not discoverable without domain knowledge.

Table 1 shows two example duplicates (pairs with ids 1 and 2) and two example non-duplicates (pairs with ids 3 and 4) from the real-world *restaurants* dataset (see Section 6.1). Most duplicate classifiers effectively distinguish true and false pairs, but many – especially ML-based ones – cannot provide a human understandable reasoning for their decision. Our approach, in contrast, does this by offering the rules, i.e., MDs that marked the pairs as true duplicates.

Intuitively, a matching dependency is a functional dependency $X \rightarrow A$ with a set of left-hand-side (LHS) attributes X and a right-hand-side (RHS) attribute A where attribute values do not need to be exactly equal but *similar* w.r.t. some attribute-specific similarity measure and some attribute- and dependency-specific similarity threshold (more details in Section 3.1). The similarities in these rules are expressed in the range of [0.0, 1.0] with 0.0 describing fully dissimilar values and 1.0 equal values.

For example, given the restaurants dataset, our *MDedup* system detects (among others) the second pair of Table 1 as duplicate providing the rule $\texttt{name}_{0.7}, \texttt{address}_{0.7} \rightarrow \texttt{phone}_{0.75}$ as an explanation for this decision. This rule is interpreted as follows: All record pairs in restaurants with a name similarity of at least 0.7 and an address similarity of at least 0.7 have a phone similarity of at least 0.75. This *if-this-then-that* rule is true for all record pairs, but the LHS matches only a few records, which are those that are considered to be duplicates. In our example, the LHS is true only for record

Proceedings of the VLDB Endowment, Vol. 13, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3377369.3377379

Table 1: A sample of duplicate (<163,164> and <165,166>) and non-duplicate record pairs (<180,823> and <676,811>) from the restaurants dataset. The floating point numbers indicate value pair similarities.

pair id	record id	name		phone		address		city		type	
	163	georgia grille		404 352 3517		2290 peachtree rd peachtree square shopping center		atlanta		american	
1	164	georgia grille	1.0	404 352 3517	1.0	2290 peachtree rd	0.36	atlanta	1.0	southwestern	0.17
	165	hedgerose heights inn		404 233 7673		490 e paces ferry rd		atlanta		international	
2	166	hedgerose heights inn the	0.84	404 233 7673	1.0	490 e paces ferry rd ne	0.91	atlanta	1.0	continental	0.3
	180	ritz carlton cafe buckhead		404 237 2700		3434 peachtree rd ne		atlanta		american new	
3	823	ritz carlton cafe atlanta	0.74	404 659 0400	0.58	181 peachtree st	0.6	atlanta	1.0	american new	1.0
	676	johnny rockets la		213 651 3361		7507 melrose ave		la		american	
4	811	johnny rockets at	0.89	770 955 6068	0.33	2970 cobb pkwy	0.18	atlanta	0.29	american	1.0

pair 2 so the MD classifies it as a duplicate; the pairs 1, 3, and 4 also meet the MD, but they do not fulfill the LHS condition. Note that matching dependencies can be used also in other ways for data cleaning, e.g., to automatically correct RHS attribute values [8, 9], but we focus on their duplicate detection ability in this paper.

On the entire restaurants dataset, the matching dependency $\texttt{name}_{0.7}, \texttt{address}_{0.7} \rightarrow \texttt{phone}_{0.75}$ *alone* is able to achieve an F-measure of 72% while maintaining a perfect precision of 100%. To also capture record pair 1 as a duplicate, a second MD-rule, namely $\texttt{name}_{0.95}, \texttt{city}_{0.82} \rightarrow \texttt{phone}_{0.75}$, is necessary. For this reason, our duplicate detection system needs to identify good *sets* of MDs rather than a single MD. Both MDs together achieve an F-measure of 78% with still 100% precision, which is the best possible result that can be achieved with any combination of MD-rules on restaurants – the remaining duplicates cannot be described by automatically discoverable MDs. Because the set of discovered MDs might also contain many MDs that match non-duplicates, it is crucial to pick only those for duplicate detection that are appropriate *duplicate classifiers*. In this paper, we propose a machine learning approach that learns to distinguish good from bad MD-rules based on several novel features.

More specifically, our *MDedup* duplicate detection system works in two phases: *training* and *application* (see Figure 1). The training phase first discovers all minimal MDs in possibly many pre-annotated datasets, i.e., ones with a gold standard of duplicates. The discovery itself is fully automatic and does not require the gold standard (Section 4.1). The gold standard is then used after the discovery to find the optimal subsets of discovered MDs, which are those that achieve the highest F-measure scores. With the scored MD combinations and certain characteristic features (Section 4.4.1), *MDedup* trains a regression model to predict effectiveness scores for arbitrary sets of MDs (Section 4.4).

The application phase then takes this general *prediction model* to predict the scores of MD combinations from a different, dirty dataset (without own gold standard). Afterwards, the best MD combination is used as a classifier for a first round of duplicate detection. Because the result usually has high precision but low recall, we propose a final *boosting* step, in which the high precision duplicates are used to train a binary classifier, which is a typically used support vector machine (SVM) model, to find further duplicates (Section 5). This final boosting step clearly improves the system's recall in our experiments (Section 6.3.3).

Intuitively, this paper addresses the transfer learning problem in duplicate detection (*train on known datasets, apply on a new dataset*) with a non-transfer learning solution (*learn how to judge MDCs regardless of the domains of their datasets*) by solving the problem on a different, domain-agnostic level. Our contributions are summarized as follows:

MDedup system. A fully automatic end-to-end duplicate detection system that is based on discovered matching dependencies. The system is trained on a few datasets with a gold standard, but can then be applied to arbitrary datasets. It adapts several well known techniques and combines them into a novel, domain-agnostic duplicate detection solution.

Feature definition. A set of novel features and heuristics that serve to train a model on how to distinguish accurate from inaccurate MDs for duplicate classification tasks.

MD selection. An efficient algorithm to select the best subset of MDs for duplicate detection either w.r.t. some gold standard or a pre-trained quality prediction ML model.

Evaluation. Various experiments that demonstrate the effectiveness of *MDedup* on eight real-world datasets with different domains and sizes.

We provide our MD combinations, the trained ML models, and the source code of *MDedup* online[1] to be reused in other cleaning projects. The rest of the paper is organized as follows: Section 2 provides a summary of related work. Section 3 then introduces the most relevant concepts before Section 4 describes our *MDedup* system's training methodology. Afterwards, Section 5 presents the application methodology, used to obtain duplicates in a new dataset. In Section 6, we evaluate *MDedup* and finally conclude in Section 7 with further remarks and future work.

2. RELATED WORK

In this section, we first discuss related work in the broad area of *duplicate detection*. We then concentrate on existing approaches for *automatic duplicate classification*, which is the focus of this research. In the end, we discuss *matching dependencies* and their applications in data cleaning. In

[1] https://hpi.de/naumann/projects/repeatability/duplicate-detection/mdedup.html

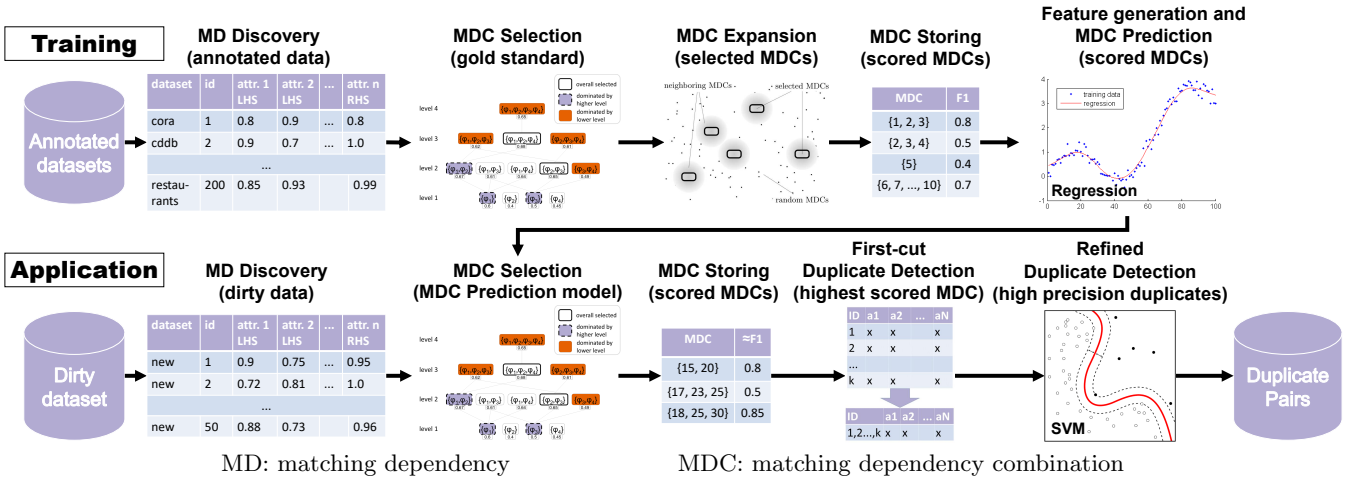

Figure 1: Overview of our *MDedup* duplicate detection system; see larger figures in Figures 2 and 3.

summary, we argue that our system is the first fully automatic, domain-agnostic system that is able to learn duplicate classification characteristics on datasets with a gold standard and use that knowledge on any other dataset.

The duplicate detection process. Research on identifying duplicates has been present for decades, as the problem appears in many applications. The commonly applied steps are (1) data preparation, (2) candidate indexing and retrieval, (3) candidate comparison and scoring, (4) candidate classification as either matches or non-matches, and (5) process evaluation [1, 2]. In this paper, we focus mostly on the fourth part, which is the *classification of record pairs*. For all the other steps, we use standard techniques (Sections 6.1 and 6.2). For our evaluation, we use the common metrics F-measure, precision, and recall (Section 6.3).

Several tools exist that implement the previously discussed duplicate detection phases [10]. Febrl [11] and JedAI [12] are two examples developed in Python and Java, respectively. Rule-based approaches, such as AJAX [13], let domain expert users specify duplicate detection rules in a declarative way. In cases where gold standard duplicates are available for the target dataset, duplicate detection rules can be generated automatically [6, 7]. In this work, though, we do not assume pre-labeled data for the target dataset and, hence, solve a more difficult problem.

Automatic duplicate classification. To date, no algorithm exists that *automatically* discovers duplicates without domain-specific parameterization and pre-labeled data (on the target dataset). In contrast, classification in duplicate detection is, in general, a well-researched area. Swoosh [14] and Metablocking [15] are two example threshold-based approaches, but they are hard to parameterize without domain expertise. The same is true for unsupervised approaches, such as [16] and [17]: In [16], the blocking strategy, the duplicate probability parameters, the similarity measures, and the configuration of the SVMs are all selected manually; similarly, [17] needs to configure its SVM and requires thresholds to be set for the Threshold and Nearest-based approaches. Similarly to our approach, however, both [16] and [17] also use an SVM classifier in a final boosting step.

A variety of works focus on ML models using either custom features or having artificial neural networks learn their features [4, 5, 17]; the resulting models are always tailored to the data they were trained on, because they learn how to classify domain-specific record pairs rather than general data cleaning rules. Decision Trees [18], Support Vector Machines (SVM) [17], and Deep Learning [4,5], are examples of such ML systems. They all require pre-labled training data from the data that needs to be cleaned. All these approaches can be used in our system for the final boosting step. In our implementation, though, we have chosen an SVM approach, as it was also successfully used for boosting by [16] and [17].

Matching dependency based approaches have also been used in duplicate detection [8, 9, 19]. However, they all start with some trusted, manually picked MDs and do not automatically select them – which is a major part of our contribution. These approaches also primarily aim at data correction rather than duplicate detection [8, 9].

The Snorkel system [20] is similar to our approach in that it also starts without pre-labeled data, but it requires some starting rules or functions whose definition, again, requires domain knowledge in the target dataset. The record linkage system of Negahban et al. [21] is able to match records across datasets without needing example matches between these two datasets, which is similar to our setup. The system, however, assumes that the two datasets share the same domain and that gold standard matches are available that linked both datasets to same other datasets – two very restrictive assumptions that are hardly met in practice.

Matching dependencies. Functional dependencies (FDs) are one of the most recognized types of data dependencies, also due to their capabilities in data cleaning when used in relaxed forms [22]. Matching dependencies (MDs) are one such relaxed form that was first introduced by Fan [23]. MDs extend FDs by also matching similar and not only strictly equal record values (see Section 3.1).

Song and Chen [24, 25] proposed the first discovery algorithm for MDs. The most recent algorithm for automatic MD discovery is HyMD [26]. This algorithm is the to date most efficient approach and we, therefore, use it in our duplicate detection system (see Section 4.1). However, any MD profiling algorithm can be used to serve the MDs.

Matching dependencies have been used in various works for data cleaning purposes. In [8,9], MDs are used in a query

answering environment. Here Minimally Resolved Instances (MRIs) define the final result set of records for a given question and they are produced by iteratively enforcing right-hand-sides of MDs in a repair step. Another approach by Bahmani et al. uses MDs for candidate blocking and to merge duplicate records [19]. For the actual duplicate classification, however, they use standard ML techniques, such as SVMs and K-Nearest Neighbors (KNN). In summary, all these approaches use MDs to identify duplicate candidates and to repair right-hand-side values. In our work, we focus on the detection of duplicates rather than their correction or resolution. In contrast to all MD-based previous works, we use MDs for the *classification* only and solve the problem of *selecting* proper MDs in unsupervised scenarios where *no gold standard* with labeled duplicate pairs is available.

3. BACKGROUND AND NOTATIONS

In this section, we first give a more detailed definition of MDs and explain how we use them for duplicate detection (Section 3.1). We then introduce the concept of matching dependency combinations (Section 3.2).

3.1 Matching dependencies

Let R be a relational schema and r an instance of R. We identify the attributes of R by index, i.e., $\mathrm{R} = \langle A_1, A_2, ..., A_y \rangle$. A functional dependency (FD) on R is defined as $X \rightarrow A_i$ with $X \subseteq \mathrm{R}$ and $A_i \in \mathrm{R}$. An FD denotes that all pairs of records with same X values also have same A_i values. X and A_i are also known as LHS and RHS, respectively.

Matching dependencies are a relaxation of functional dependencies as they introduce three extensions: First, they relax the value comparisons, which is strictly equal ($=$) in FDs, by incorporating similarity metrics; this makes MDs useful for duplicate detection. Second, records can in theory be matched on different attributes, although this is not particularly useful for duplicate detection purposes. Third, they can match records across different relations, which is useful in the scenario of record linkage where the similarity join is between two relations ($\mathrm{R} \bowtie \mathrm{S}$); for the sake of simplicity and without loss of generality, we focus in this paper on single relation joins, i.e., self-joins ($\mathrm{R} \bowtie \mathrm{R}$).

The fuzzy matching of MDs is accomplished using a set of *similarity measures* ($\approx$), such as Levenshtein [27] and Jaccard [28]. These similarity measures calculate similarities in the range of [0.0, 1.0]. To classify two values as match or non-match, MDs also specify a *decision boundary*, which is defined in [0.0, 1.0]. Similarities greater than or equal to this boundary are considered matches and lower similarities non-matches. The decision boundaries of the LHS are denoted as $\lambda = \{\lambda_1, \lambda_2, ..., \lambda_m\}$ whereas the RHS has only one decision boundary ρ. Due to their strong connection, we call the combination of a similarity measure ($\approx_i$) and a decision boundary λ_i (or ρ) a *similarity classifier* $\approx_{i,\lambda_i}$ (or $\approx_{i,\rho}$). This leads us to the following definition [26]:

Definition 1 (Matching dependency)**.** *Given a relational schema R with attributes $A_i, A_j \in R$, its instance r, and similarity classifiers $\approx_{i,\lambda_i}$ and $\approx_{i,\rho}$, a matching dependency (MD) φ is defined as follows:*

$$\forall r_s, r_t \in r\colon \left(\bigwedge_{i=1}^{w-1} r_s[A_i] \approx_{i,\lambda_i} r_t[A_i] \right) \rightarrow r_s[A_w] \approx_{w,\rho} r_t[A_w]$$

In other words, a matching dependency states that if two records r_s and r_t match in all their A_i values (attribute-specific similarity calculated by $\approx_i$ greater than or equal to λ_i) then their A_w values need to be at least ρ similar w.r.t. $\approx_w$. Note that in the broader definition of MDs the two A_i attributes and the two A_w attributes can be different attributes and even attributes from different relations.

For practical reasons and because there is usually only one reasonable similarity measure per attribute, we usually use the following short notation to specify MDs:

$$\left(\bigwedge_{i=1}^{w-1} A_{i,\lambda_i} \right) \rightarrow A_{w,\rho}$$

The MD $\texttt{address}_{0.7}, \texttt{name}_{0.7}, \texttt{type}_{0.71} \rightarrow \texttt{phone}_{1.0}$, which is a true MD in the restaurants dataset (see example records in Table 1), follows this short notation. If for two records all LHS similarities match, they match the RHS similarity.

To use an MD for *duplicate detection*, we simply consider its LHS as a classifier: All record pairs that match the MD's LHS are labeled as *duplicate*. Intuitively, the LHS is the matching rule that we are looking for and the presence of a valid RHS is an indicator (for rule discovery and scoring) that the LHS is relevant. For the majority of MDs, this inference leads to poor results. The MD $\texttt{name}_{0.0} \rightarrow \texttt{phone}_{0.0}$, for instance, is true on any instance of the restaurants dataset and it matches *all* record pairs. Hence, the challenge is to identify such MDs that are *useful* duplicate classifiers. The rule $\texttt{address}_{0.7}, \texttt{name}_{0.7}, \texttt{type}_{0.71} \rightarrow \texttt{phone}_{1.0}$, for example, yields an F-measure of 72%, which is relatively good. We discuss indicators that hint towards useful MDs for duplicate classification in Section 4.4.1.

3.2 Matching dependency combinations

As we illustrated in Section 1, one MD might not be able to capture all duplicates. Our approach, therefore, considers multiple MDs for the classification. We refer to these sets of MDs as matching dependency combinations:

Definition 2 (Matching dependency combination)**.** *Given the set of all MDs Φ, a matching dependency combination (MDC) χ is any selection of MDs with $\chi \subseteq \Phi$.*

An MDC χ can be used as a *duplicate classifier* by considering a record pair as duplicate, iff it matches the LHS of at least one MD $\varphi \in \chi$. Technically, an MDC-based duplicate classifier combines a set of MD-based duplicate classifiers via logical *or*-operations. In this way, different MDs can be used to classify different kinds of duplicates. The goal of MD-based duplicate detection in general is therefore to predict the best MDC for the duplicate classification step.

To illustrate, consider again our running example of Table 1, which offers 5 MDs. Our MDC selection strategy (see Section 4.2) then selects 17 MDCs in total from the $2^5 - 1 = 31$ possible combinations. One of these MDCs consists of the two MDs $\texttt{name}_{0.7}, \texttt{address}_{0.7} \rightarrow \texttt{phone}_{0.75}$ and $\texttt{name}_{0.95}, \texttt{city}_{0.82} \rightarrow \texttt{phone}_{0.75}$, which offer the best possible F-measure of 78% (and a precision of 100%).

4. MDEDUP TRAINING

The MDedup process is comprised of two phases, namely training and application. We explain the training phase now and the application phase in Section 5. As already shown in the MDedup overview of Figure 1, the training phase takes

several datasets and their labelled duplicates as input and, then, runs five basic steps: At first, the pipeline *discovers* all minimal MDs (Section 4.1). Using our MDC selection algorithm, it then *selects* MDCs that produce possibly high quality results when used as duplicate classifiers (Section 4.2). In the third step, this set of high quality MDCs is enriched with further MDCs of varying quality via systematic *expansion* (Section 4.3). Afterwards, MDedup's training pipeline *stores* all scored MDCs in one training set that integrates the results calculated on different annotated datasets. The fifth and last step of the pipeline *generates* a set of features for the scored MDCs to then use both the MDCs and their features to *train* a machine learning (ML) model on how to *predict* the F-measure of a given MDC without a gold standard of duplicate pairs (Section 4.4).

4.1 MD discovery

The main asset of our system are the MDs that we use for duplicate classification, because they prescribe the optimal recall and precision that we can achieve. It is therefore essential to discover many and good MDs – in our implementation of MDedup, we discover *all minimal, non-trivial* MDs. Although any set of MDs can be used as input for our duplicate detection system, the complete set of minimal and non-trivial MDs is promising, because these MDs are by definition very close to core data patterns; it is also what most existing dependency profiling algorithms discover.

An MD φ is *minimal* if no other MD φ' with same LHS and RHS exists, such that only one LHS threshold is smaller (i.e., more general) or the RHS threshold is larger (i.e., more restrictive); given a valid MD, raising LHS thresholds or decreasing the RHS threshold will always generate a valid MD. Additionally, an MD φ is *trivial* if its RHS attribute is contained in its LHS attributes with either the same or a higher similarity threshold; if the LHS matches only those records with at least similarity x in attribute A_i, then these records are trivially also at least x (or less) similar in A_i.

MD profiling algorithms, such as those published by Song and Chen [24, 25], can effectively serve our pipeline with MDs. The implementation of MDedup that we developed for this paper, however, uses our own algorithm *HyMD* [26] — the most efficient MD algorithm to date. It discovers all minimal, non-trivial MDs using several indexes to identify similar records, along with a number of pruning rules that effectively tame the exponentially large search space.

4.2 MDC selection

The goal of MDedup's MDC selection component is to find MDCs that produce possibly high F-measure scores when used as duplicate classifiers. Identifying which combinations of MDs are the best is a challenging task, because the complexity of checking all possible combinations of a set of m discovered MDs is exponential, i.e, in $\mathcal{O}(2^m)$. More specifically, there are $\sum_{k=1}^{m} \binom{m}{k}$ candidates to be evaluated. The problem is particularly hard to solve, because our system often deals with thousands of MDs. For this reason, we propose a *greedy selection algorithm* that systematically searches the combination space for MDCs with high F-measures. The greediness of the algorithm decides which combinations are worth being further investigated, pruning all those MDCs on the way that offer sub-optimal F-measure scores. Because optimizing for F-measure is a non-convex problem [29] and we propose a linear, bottom-up greedy search, our approach does not guarantee optimality. It however always found the optimal combination in our experiments.

Before we discuss our lattice traversal-based search process, we first define certain preliminaries. To keep the search within reasonable time constraints, but at the same time consider a variety of possible solutions, we define a parameter k, which controls the fan-out of the process. In each level of the lattice, a maximum of *top* k MDCs produce further candidate MDCs to be considered in the next higher level of the lattice, with each candidate MDC being one MD larger than its predecessor. To consider more variations as we go up the lattice levels, we employ another trick. Instead of just combining the top k candidates of each level with each other, which would produce a narrower fan-out, we combine every candidate with every MD of the first level. This grows the search space linearly, but at the same time allows us to consider some variation in our candidates. Nevertheless, the main focus is kept on the combinations that produce good scores. The process could also be characterized as "hill climbing with top-k". Although this approach is heuristic, an exhaustive, i.e., complete candidate testing approach produced the same results on smaller datasets in our experiments. We label the greedily selected MDCs as "selected MDCs" and define them as follows:

Definition 3 (Selected MDC)**.** *An MDC χ is defined as "selected" if $\exists$ MD φ where $\chi - \{\varphi\}$ reduces its score and $\nexists$ φ' where $\chi \cup \{\varphi'\}$ improves its score.*

Based on this definition, Algorithm 1 solves MDedup's *selection* process. The parameters of this process are the *HyMD* algorithm that discovers the MDs, the Oracle algorithm that calculates the F-measure score given an MDC, and the search scope k of the greedy approach. For the training phase, the Oracle algorithm uses the given duplicate gold standards; the calculations follow the principles that we described in Section 3.2. In the application phase, the Oracle algorithm uses the trained MDC scoring model.

The MDC selection algorithm starts by calling HyMD to discover the MDs for this dataset (line 2). Then, it scores the initial MDCs, where every MDC contains exactly one MD (line 3). These MDCs are shown in Figure 2 at level 1. Subsequently, it filters out MDs with a score of zero to reduce the search space (line 4). Next, two sets are initialized (lines 5 and 6): the best MDCs for the currently examined level (*levelSelected*) and the overall best MDCs (*overallSelected*). None of the overall best MDCs may be dominated by another MDC of the *levelSelected* set. The algorithm then starts to iterate all search space levels from level one upwards as long as there are more non-dominated MDCs in the current level (line 7). For each level, the algorithm creates the respective candidate MDCs, based on the MDCs of the previous level (lines 8 to 11). For each generated set, the algorithm iterates over all its candidates, calculates their score, and if that score is higher than the scores of the MDCs that created them, they are added to the *levelSelected* set (lines 12 to 17). It proceeds by keeping only the k best MDCs (line 18), while these k MDCs (*levelSelected*) are added to the *overallSelected* set (line 19). Consequently, it removes the creator MDCs of the selected k from the *overallSelected* set (lines 20 and 21). We do not remove MDCs that create higher scored MDCs if these higher scored MDCs are not selected in the top k of their level to maintain a higher diversity in the final set. Finally, the *overallSelected*

Algorithm 1: MDC selection

```
1  function MDC_selection(HyMD, ORACLE, k)
       /* Discover, score, and filter MDs */
2      mds ← HyMD.discoverMDs()
3      mds ← ORACLE.score(mds)
4      mds ← mds.filter(lambda md : md.score > 0)
       /* Initialize MD combinations sets */
5      levelSelected ← topk(mds, k)
6      overallSelected ← {levelSelected}      /* Copy */
7      while |levelSelected| > 0 do
           /* Combine with single MDs */
8          candidates ← {∅}
9          for mdc ∈ levelSelected do
10             for md ∈ mds \ mdc do
                   // mdc ∪ md : {1,2} ∪ {3} = {1,2,3}
11                 candidates.add(mdc ∪ md)
           /* Score MDCs and keep those with
              improved score */
12         levelSelected ← {∅}
13         for mdc ∈ candidates do
14             mdc.score = ORACLE.score(mdc)
15             creatorsScore = max(mdc.creators.scores)
16             if mdc.score > creatorsScore then
17                 levelSelected.add(mdc)
           /* Filter and save topk, and remove
              their creators */
18         levelSelected ← topk(levelSelected, k)
19         overallSelected.addAll(levelSelected)
20         for mdc ∈ levelSelected do
21             overallSelected.removeAll(mdc.creators)
22     return overallSelected
```

set containing all the best, non-dominated MDCs across all levels is returned (line 22).

To better understand the selection process, consider the example in Figure 2 with $k = 2$. It shows a lattice of MDCs with the top MDC describing the combination of all MDs and the bottom holding all single-MD MDCs. The execution begins at the bottom by considering the single-MD MDCs, which are given by the MD discovery algorithm (see Section 4.1). As explained previously, we consider only the top k candidates with highest F-measure scores to reduce the search space. After selecting the top k (*levelSelected*), which are $\{\varphi_1\}$ and $\{\varphi_3\}$, from the first level, the algorithm combines them with every other possible MD to produce the candidates for the next level. Moving to level 2, notice that the combination $\{\varphi_2, \varphi_4\}$ is not considered, as neither $\{\varphi_2\}$ nor $\{\varphi_4\}$ were selected in the previous *levelSelected*. By calculating the MDC scores, we find that all MDCs of size 2 have a better score than their creators, i.e., $\{\varphi_1\}$ and $\{\varphi_3\}$, except one, which is MDC $\{\varphi_3, \varphi_4\}$ with a worse score than $\{\varphi_3\}$; thus it is eliminated and not considered for the *levelSelected* selection. Because $\{\varphi_1\}$ and $\{\varphi_3\}$ created descendants with better scores, they are removed from the final result set (*overallSelected*). The same process now repeats at level 2: We calculate the scores for each MDC and keep the top k, which are $\{\varphi_1, \varphi_2\}$ and $\{\varphi_2, \varphi_3\}$. Proceeding to level 3, we find only one MDC, which is $\{\varphi_1, \varphi_2, \varphi_4\}$, that is not dominated by its creators. In the end, we select

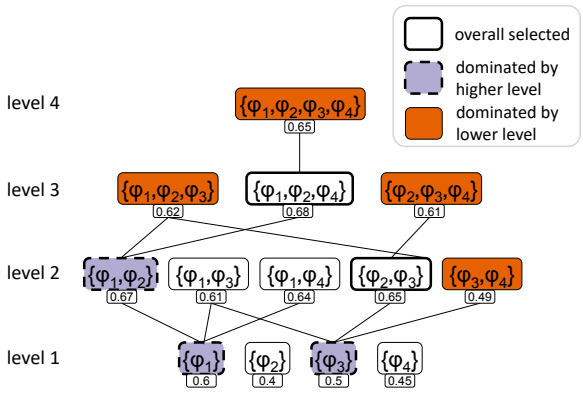

Figure 2: An example for the selection algorithm on an MDC lattice with four input MDs and $k = 2$. The finally selected MDCs are $\{\varphi_1, \varphi_2, \varphi_4\}$ and $\{\varphi_2, \varphi_3\}$.

$\{\varphi_1, \varphi_2, \varphi_4\}$ and $\{\varphi_2, \varphi_3\}$ in *overallSelected* as final result.

4.3 MDC expansion

The goal of MDedup's training phase is to create a model that can predict F-measure scores for MDCs. Because most discovered MDCs have a low F-measure score, the MD selection step finds a set of k MDCs with high scores. This set is, therefore, biased towards top performing MDCs and it is rather small. Most ML models, however, behave better if their training data is more diverse and they are given more data [30]. So to ensure that enough training instances with a large variety in their properties exist, we propose an MDC *expansion* process that consists of two strategies: neighbor expansion and random sampling.

Given the top k MDCs from the selection step, the *neighbor expansion* strategy considers neighboring MDCs around these selected MDCs. This is achieved by an iterative process where we randomly select an input MDC and then *add*, *remove*, or *replace* some MDs using a Gaussian distribution, thus ensuring a focus on neighboring MDCs. Figure 3 visualizes this concept: Rectangular nodes represent the initially selected MDCs and their shaded neighborhoods show the Gaussian distribution from which the additional MDCs are generated. By taking neighboring MDCs of top performing MDCs into the training set, we enable a machine learning algorithm to learn the precise characteristics of why certain MDCs perform well. The result of this strategy therefore produces a larger set of MDCs that describes the few top performing MDCs well.

Because the neighbor expansion strategy is still biased around top scored MDCs, the *random sampling* strategy injects MDCs of larger variety: It randomly creates (and scores) MDCs by considering the entire search space up to the maximum level reached by the MDC selection, disregarding selected and neighboring MDCs. In Figure 3's visualization, the random MDCs are scattered all over the surface. We control the overall size of the expansion set (relative to the number of selected MDCs), equally for both strategies, with the parameter *expansion_factor*.

4.4 Feature generation and MDC prediction

In this section, we describe the machine learning model that we train to predict F-measure scores for MDCs. MDedup needs this model to score the MDCs in the application

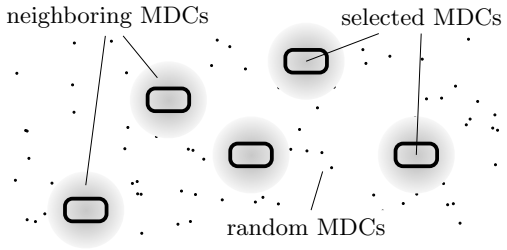

Figure 3: A visualization for two expansion strategies: neighbor expansion and random sampling.

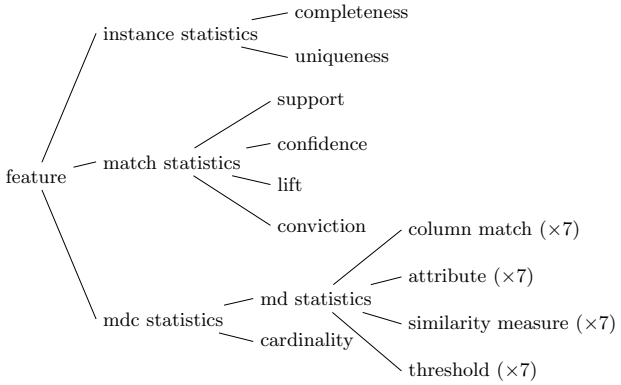

Figure 4: Taxonomy of the features based on the data required for their calculation.

phase where no gold standard is available to calculate the exact F-measure values (see Section 5). To build the ML model, we need to specify a set of features and the prediction strategy. For the features, we collected thirty-five metrics of MDC characteristics that are possibly relevant for the duplicate classification performance of the MDC instances (see Section 4.4.1). For the prediction strategy, we propose a Gaussian regression process (see Section 4.4.2).

4.4.1 Assembling features for MDCs

To predict the F-measure an MDC would achieve when used as a duplicate classifier, we determine several features describing the MDC itself and the data for which it holds. These features need to be available on any relational input dataset and do not need a gold standard.

In total, we define thirty-five features, which are outlined in the taxonomy presented in Figure 4. Based on this taxonomy, we have the following main categories: instance statistics, match statistics, and MDC statistics. First, the *instance statistics* metrics describe basic statistics of the relational input dataset, such as the median completeness or uniqueness of attributes w.r.t. all attributes used in an MDC. Second, the *match statistics* metrics describe how well an MDC is supported by the data. Finally, the *mdc statistics* metrics describe the structure of an MDC and its MDs. In particular, these are metrics based on MDC characteristics, such as their size, and the similarity measures, attributes, and thresholds that are used in the MDs' LHSs and RHSs. Having explained the main feature categories, we now describe the individual features.

Completeness. Completeness is the percentage of records with no null values in the attributes of the MDC. Null values are not useful for record matching and, therefore, this feature can help to disregard MDCs that match on sparse or empty data. With '$\perp$' representing null values and $attr(\chi)$ the union of all attributes used by the MDC's MDs, the completeness feature is calculated as follows:

$$compl(\chi) = \frac{|\{r_i \in r \mid \forall A_j \in attr(\chi) \ : \ r_i[A_j] \neq \perp\}|}{|r|}$$

Uniqueness. The uniqueness of an MDC is the fraction of unique value combinations in the MDC's attributes, relative to the overall number of records. A high uniqueness is a good indication that the given set of attributes is suitable as a key, in that it can uniquely identify a record when comparing it with others. If we represent the projection of the relational instance r to all the attributes of the MDC χ as $r[attr(\chi)]$, the uniqueness of χ is defined as follows:

$$uniq(\chi) = \frac{|\{r_i \in r[attr(\chi)]\}|}{|r|}$$

Support. The support of an MDC is the percentage of record pairs that match at least one MD of the MDC on the LHS ($suppLHS(\chi)$), the RHS ($suppRHS(\chi)$), or both sides ($supp(\chi)$). In this way, the support features encode how strong the MDC is and what duplicate detection recall we can expect. For exact MDs, which are all MDs that we discover, $suppLHS(\chi) = supp(\chi)$. For definition purposes, let $p = r \times r$ be the set of all record pairs and $match(p_i, X)$ a function that returns true if the record pair p_i matches the match conditions X, i.e., it matches $\bigwedge_{i=1}^{w-1} r_s[A_i] \approx_{i,\lambda_i} r_t[A_i]$ for LHSs and $r_s[A_w] \approx_{w,\rho} r_t[A_w]$ for RHSs. Then the support features are defined by the following three formulas:

$$supp(\chi) = \frac{|\{p_i \in p \mid \exists \varphi \in \chi : match(p_i, \varphi.LHS \cup \varphi.RHS)\}|}{|p|}$$

$$suppLHS(\chi) = \frac{|\{p_i \in p \mid \exists \varphi \in \chi : match(p_i, \varphi.LHS)\}|}{|p|}$$

$$suppRHS(\chi) = \frac{|\{p_i \in p \mid \exists \varphi \in \chi : match(p_i, \varphi.RHS)\}|}{|p|}$$

The formulas are a translation of $supp(X) = \frac{|\{t \in T; X \subseteq t|}{|T|}$ [31]. Caching these scores after their calculation allows us to re-use them for the calculation of the following metrics.

Confidence. The confidence feature describes the portion of the data that satisfies at least one MD of the MDC. If all MDs are correct, i.e., no approximation was used during MD discovery, the confidence is always 100%; otherwise, the feature allows the algorithm to trust correct MDCs more than partially violated MDCs. Using our exact MD discovery algorithm, the confidence measure is basically irrelevant. However, it might be useful in alternative setups. Hence, the feature definition is as follows:

$$conf(\chi) = \frac{supp(\chi)}{suppLHS(\chi)}$$

Lift. The lift feature is a measure for correlation and represents the ratio of record pairs where both LHS and RHS are satisfied, divided by the product of independent percentages for LHS and RHS. A number larger than 1 indicates that

the two sides tend to co-occur, meaning that if one appears the other will appear as well. On the other hand, a number smaller than one means the opposite. The calculation uses the following formula:

$$lift(\chi) = \frac{supp(\chi)}{suppLHS(\chi) \cdot suppRHS(\chi)}$$

Conviction. Conviction is a metric that can be interpreted as the fraction of the expected frequency that a LHS co-occurs with its RHS. Conviction values higher than 1 indicate to what extent the associations of the MDs' LHSs and RHSs are random. This should help the model to prefer meaningful dependencies over spurious ones. The feature is defined as follows:

$$conv(\chi) = \frac{1 - suppRHS(\chi)}{1 - conf(\chi)}$$

MD statistics. The MD statistics are set of features that describe significant characteristics of a given MDC by considering only its definition and not the data is was discovered from. The features use the following elements of the MDs: attributes, similarity measures, and thresholds, as well as their column matches, which is the composition of the previous three, i.e., A_i, $\approx_i$, and λ_i. Each MD in an MDC offers a set of these elements. We define a feature for each of these four element sets by calculating the following set operations:

- *Intersection size*: The number of elements that exist among all MDs (1).
- *Union size*: The number of elements that exist in at least one MD (2).
- *Jaccard distance*: The distance between common attributes and specific attributes, which is calculated by dividing the intersection size with the union size; as a result, we get a normalized value in the range of $[0.0, 1.0]$ (3).
- *Descriptive statistic*: Significant and extreme elements over all sets, calculated with the basic metrics of min (4), median (5), max (6), and stdev (7).

Cardinality. When increasing the size of an MDC by adding more MDs, its recall increases monotonically and its precision decreases monotonically. The cardinality of an MDC, which is simply defined as $card(\chi) = |\chi|$, is therefore an important feature for estimating the MDC's F-measure. Small MDCs would be preferable for human readability, whereas large MDCs can cover more special cases.

4.4.2 Regression for MDC score prediction

Using the feature definitions discussed above, we can now generate an instance set of scored MDCs with their MDC-specific features. With this instance set, we can train a machine learning model to predict the F-measure score for arbitrary MDs and their datasets. Although different types of models could be considered for this task, *regression models* seem to be a natural fit. They learn to estimate the relationship between a feature set that for us is MDC-specific, and a target variable, which in this case is the F-measure. Although many different regression models exist, we propose a Gaussian Process [31] for two main reasons: First, it is a capable of learning complex data patterns, which is necessary considering the many factors that influence the performance of an MD as a duplicate classifier. Second, it has a relatively small set of parameters, which is important, because our MDedup target system should be a largely automatic system and a small parameter space simplifies the hyperparameter tuning phase. We describe the configuration in further detail in Section 6.2.

5. MDEDUP APPLICATION

This section describes the application phase of the MDedup duplicate detection system. This phase is shown as the lower pipeline in Figure 1 and starts with a dirty dataset for which no gold standard records exist. Instead of using pre-labeled data for the scoring of MDCs, the application phase takes the regression model that was constructed on other datasets in the training phase and is able to predict F-measure scores for MDCs w.r.t. their relational instance data (see Section 4). Because most of the application pipeline uses the same components as the training pipeline, we focus on the differences in the following descriptions.

At first, the application pipeline *discovers* all minimal MDs on the dirty input dataset with the same algorithm as for the training phase (Section 4.1). In the next step, the pipeline uses our MDC selection algorithm (Section 4.2) to greedily *select* the top k MDCs with the highest F-measure scores. Instead of scoring each MDC using the gold standard, the selection algorithm in the application phase uses the regression model provided by the training phase (Section 4.4). The results of the selection are then passed to the MDC store were MDedup *picks* only the highest scored MDC as the duplicate classifier. The next step of the application pipeline then *detects* all record pairs that this MDC classifier matches as duplicates. As a final step, MDedup uses the discovered duplicate pairs to *boost* the overall F-measure of the process: With the known duplicates, the pipeline trains a state-of-the-art SVM model to then classify record pairs as duplicates or non-duplicates; this model is applied to the input data to refine and complement the duplicates.

MDC selection. A benefit of using a regression model for the MDC scoring is that we can easily convert our MDC selection approach for the application phase without any gold standard. Looking back at Algorithm 1, the score of every MDC is given by the ORACLE algorithm. In the absence of a gold standard, this ORACLE now needs to use the regression model for the F-measure calculation.

To predict the F-measure for an MDC, the ORACLE first generates a feature vector with the features discussed in Section 4.4.1. This feature vector is then provided to the regression model, which returns an estimate for the F-measure. The lattice traversal and top-k selection strategy in the application phase is identical to the training phase. The result of the MDC selection step is a list of MDC associated with their predicted F-measure scores.

First-cut duplicate detection. Given the highest scored MDC of the selection step, MDedup executes its first duplicate detection pass. This pass involves the three commonly applied steps of candidate indexing, candidate matching, and candidate classification. In the following, we briefly describe our implementations for these steps.

The *candidate indexing* method is responsible for the selection of record pairs that should be tested. The duplicate detection could, in theory, be applied to all pairs of records in the input relation, but this large number of candidates is, in practice, usually reduced by blocking methods [1, 2] that

divide the quadratic candidate space into smaller subsets (i.e., blocks) according to a predefined partitioning key. To make sure that similar records fall into at least one common block and, hence, can get matched, MDedup takes every attribute as a partitioning key once; records with same value in that attribute are put into the same block. For large values, i.e., ones that are comprised of more than four words, the algorithm extracts word n-grams, with $n = 4$, from the value where each word n-gram defines a bucket; records with long values are, in this way, placed into multiple buckets. The result of indexing are several, overlapping blocks of records.

For each block, the *candidate matching* step creates all pairs of records within the block. Because the blocks share large overlaps, MDedup places all pairs from all blocks into a common set that removes redundant pairs, reducing the matching effort significantly. All remaining candidate pairs are then compared by calculating their attribute-wise similarities, for all attributes and similarities specified in MDs. We finally store the candidate set to use it not only for this first-cut duplicate detection process, but also for the refined duplicate detection step later on.

For the *candidate classification*, MDedup applies the highest scored MDC as a duplicate classifier to every record pair, as described in Section 3.2. The result are two sets, a set of duplicates and a set of non-duplicates.

Refined duplicate detection. The evaluation of the first-cut duplicate detection results (in Section 6.3.2) shows that the discovered duplicate sets have high precision but rather low recall. For this reason, we propose a final *boosting* step to expand the result set.

Boosting is typically used to sequentially connect learning models in a way that each model can enhance the prediction of the previous models and reduce the overall bias and variance of the results. Because the first-cut duplicate detection pass provided us with a labeled set of record pairs and their similarities, we can use them to train a typical binary classification model. The overall rationale of this boosting approach is that MDCs identify duplicates based on very specific MD rules that cannot capture all kinds of similarity patterns. SVMs, in contrast, learn schema-specific similarity patterns and are, therefore, able to match records more precisely. Combining both approaches results in an overall schema independent but still flexible classification approach.

Although many complex boosting approaches, such as AdaBoost [32], have been proposed that combine multiple models on an instance level, we propose a simple Support Vector Machine (SVM) approach, because SVMs have already proven to be effective as a boosting duplicate detection step [16, 17]. Since training a binary classification model requires labels of both classes, the application pipeline takes the entire set of *duplicates* and a subset of the *non-duplicates*, which is ten times larger than the duplicates set. With the record pairs, their similarities and duplicate labels, we finally train our SVM model. After training, we apply the model as a duplicate classifier to our candidate set to retrieve the final duplicates. We provide more details on the configuration of the boosting step in Section 6.2.

6. EVALUATION

In this section, we present our evaluation to understand the advantages and drawbacks of using matching dependencies for duplicate detection. We begin in Section 6.1 by describing our experimental setup and the data that we used for training and testing. Section 6.2 then proposes robust configuration parameters for MDedup based on several smaller experiments. In Section 6.3, we present the results of three larger experiments that evaluate the effectiveness of the MDC selection, MDC prediction, and boosting step.

6.1 Preliminaries

In this section, we first introduce our experimental setup. Then, we discuss our datasets and how we complement their gold standards of duplicates with challenging non-duplicates.

Experimental setup. All experiments were conducted on a machine with 4x Intel Xeon E7-8837 (2.67GHz, Octa-Core) and 256GB of RAM. All 16 hyperthreaded cores were used for parallelization. The system uses Ubuntu 18.04 LTS and Oracle Java 1.8. The source code, datasets, and evaluation results are available on our website[2].

Datasets. To train and test MDedup, we use eight datasets of heterogeneous domains and their known gold standards: Amazon-Walmart (products), CDDB (audio), Census (census), Cora (bibliography), DBLP-Scholar (bibliography), Hotels (lodging), NCVoters (voter registrations), and Restaurants (restaurants). Table 2 lists the datasets' number of records, duplicates (DPL) and non-duplicates (NDPL). For all datasets, we select only those attributes for the MD discovery that have a completeness >10%, i.e., at least 10% of the values are non-null, because sparse columns make the discovery more difficult and are not useful for duplicate classification. We also exclude IDs, because they often encode the gold standard and, hence, make the task trivial. Only for NCVoters, we applied a stricter attribute filtering due to the schema size of that dataset. For more details and information about which attributes were used exactly, we refer to our website.

Gold standards. The gold standards of our evaluation datasets contain lists of record pairs that uniquely identify *duplicates*. In case these lists are not transitively closed (CDDB, Census, and Hotels), we add all transitive pairs as duplicates to the gold standard in a pre-processing step.

To train and test a classifier, we also require a number of negative examples, i.e., non-duplicate pairs. When creating these pairs, we made sure that records are not paired up randomly, because most random pairs have a very low similarity and are, therefore, trivial to classify. More useful for training and more realistic for testing are non-duplicate pairs that are similar and, hence, harder to classify. To this end, we apply blocking as described in Section 5 on the input relation and pick non-duplicate record pairs from within the blocks; in this way, the non-duplicates are guaranteed to be similar in at least the blocking key. When selecting non-duplicates, we also enforce a ratio of 1:10 (DPLs to NDPLs) to not over-represent the class of non-duplicates. For Cora and Hotels, we took all non-duplicates form the blocking, which resulted in about the ratio 1:4 for both datasets.

6.2 MDedup configuration

In this section, we summarize MDedup's configuration per pipeline step and propose robust default values so that the system can be deployed without parameter tuning.

[2] https://hpi.de/naumann/projects/repeatability/duplicate-detection/mdedup.html

Table 2: Dataset statistics: number of records, attributes, duplicates (DPL), non-duplicates (NDPL), matching dependencies (MDs), matching dependency combinations (MDCs), and execution times across phases.

Dataset	Records	Attr.	DPL	NDPL	MDs	MDCs		Execution time (hh:mm:ss)		
						Selection	Prediction	HyMD	Selection	Prediction
Census	841	5	376	3,760	15	31	31	00:00:07	00:00:01	00:00:19
Restaurants	864	5	112	1,120	5	17	9	00:00:07	00:00:10	00:00:39
DBLP-Scholar	66,879	5	5,347	53,470	5	1	10	04:54:15	00:00:03	00:00:21
CDDB	9,763	6	300	3,000	11	37	11	04:21:44	00:00:01	00:00:15
Hotels	364,965	10	94,677	368,002	159	144	48	03:10:43	00:00:42	03:29:30
Cora	1,879	13	64,578	268,082	74	288	80	00:01:18	00:00:19	00:11:42
Amazon-Walmart	24,583	13	1,154	11,540	132,732	96	17	14:03:42	00:00:06	01:10:11
NCVoters	14,183	25	9,819	98,142	149,950	400	60	58:21:14	00:04:46	07:21:19

MD Discovery For the MD discovery, we use the HyMD algorithm and its default parameterization in both the training and application phase [26]. Regarding the similarity measures, we consider all attributes as alphanumeric and use the following principles: In general, all attributes are compared with the Levenshtein similarity measure [27]; attributes with a mean value length of more than 100 characters, however, use the Jaccard similarity measure [28] that compares tokens instead of characters. Jaccard is beneficial for attributes with long values, because comparing long values on character basis is slow and, in general, in-effective: They often contain descriptions, comments, and other free-text fields that are less structured and words can be on different positions across two duplicate records.

For both measures, we set the minimum similarity threshold for discovered MDs to 0.7 – lower thresholds have not shown better results in our experiments, but the discovery time increases significantly with decreasing thresholds.

Table 2 contains additional information about the execution of HyMD on our evaluation datasets. The datasets Amazon-Walmart and NCVoters produce a large number of MDs and, hence, also require the longest execution times.

MDC Selection. The MDC selection steps in both the training and application phase require the parameter k. This parameter specifies the number of MDCs with the highest score that are selected in each level of the lattice and passed to the following; it is also the number of MDCs that are returned in the end. To investigate the impact of k on MDedup's performance, we trained the system with different k values and measured the following average F-measure scores across all datasets: $1 \rightarrow 34.1\%$, $2 \rightarrow 39.5\%$, $4 \rightarrow 32.5\%$, $8 \rightarrow 47.3\%$, $16 \rightarrow 55.2\%$, $32 \rightarrow 55.4\%$, $64 \rightarrow 55.4\%$. From this, we conclude that a certain set size is needed to learn how useful MDs look like, but at some point the set captures all relevant information and larger training sets do not improve the performance. Hence, MDedup is robust against large k value, i.e., overly large values impact its runtime performance but not its effectiveness. In our experiments, we set the parameter to 16.

Because the MDC selection step has exponential complexity in the number of MDs, its execution time can be very high. For this reason, we introduced a *maximum execution time* that, when exceeded, triggers MDedup to gracefully stop the lattice traversal: the current level is finished but the next level is not started. In our experiments, we set a maximum execution time of ten hours. Hence, this time limit is essentially irrelevant for the training phase, but the feature generation and predictions in the application phase are so expensive that our two largest datasets exceed a selection time of 10 hours. Table 2 lists the numbers of selected MDCs together with the actual selection times.

To evaluate the effectiveness of the thirty-five MDC score prediction features, we executed two experiments: The first experiment is a leave-one-out experiment, where we trained the system leaving out each feature once to see how this feature impacts the F-measure. In a second experiment, we trained the system once with every feature exclusively, i.e., with only that one feature to measure its performance. With the results of these two experiments, we could not identify a particularly important or harmful feature, because every feature increases the performance on at least one dataset and/or performs well on its own. For this reason, we use all proposed features in our system.

MDC Expansion. The parameter *expansion_factor* ensures that enough MDCs exist to capture the datasets' properties, especially for datasets with an inherently small number of MDCs. It also serves to represent some non-optimal MDCs, i.e., negative examples in the training set. Both the neighbor expansion and the random sampling are configured to enlarge the set of selected MDCs by a factor of 10 each. To evaluate the impact of the expansion, we trained MDedup once with an expansion factor of 0.0, which is expansion switched off, and once with a factor of 10. Without expansion, the F-measure for DLBP-Scholar increased from 0% to 83% (the one effective MD was selected now), but this effect appears to be incidental, because the F-measure decreases for all other datasets without expansion – on average by 2%. For this reason, we propose to use the expansion step. A default factor of 10 worked well in our experiments, because the F-measure did not improve for larger values.

MDC Prediction. The *Gaussian Process*, which is our regression model for predicting F-measures, requires a kernel and a regularization λ as parameters. For the kernel, we select a Gaussian kernel [31] that in turn needs σ to control the width of Gaussian distributions. For σ, we set the value range $[2^{-3}, 2^{-1}, \ldots, 2^{7}]$ (6 values) and for λ the value range $[2^{-5}, 2^{-3}, \ldots, 2^{5}]$ (6 values). For the implementation of the Gaussian Process, we use the Smile library [33].

The experimental process for the prediction step as described in Section 6.3 follows a leave-one-out cross-validation approach on dataset level: For every individual dataset, we use all other datasets as the annotated training datasets (running the training pipeline on them) and the target dataset as the test dataset (running the application pipeline on this dataset). The best parameters, i.e., λ and σ, are cho-

sen automatically by the models through an 80:20 train-validation split approach on the annotated datasets. The selected parameters are then applied to the tested dataset. Finally, results for both MDCs and execution time are given in Table 2. Recall that datasets with execution times longer than ten hours are a result of expensive MDC score prediction; at ten hours, we terminate the selection process, returning the results obtained until that point.

Boosting classifier. Support Vector Machines (SVMs) are a widely accepted binary classifier and have already been used for boosting in duplicate detection [16, 17]. We pair SVMs with a Gaussian kernel, which is a standard radial basis function (RBF) kernel, and configure two parameters: Gaussian kernel width σ and soft margin penalty Cs. Following the *loose grid search* recommendation [34], we set the values $[2^{-3}, 2^{-1}, \ldots, 2^{7}]$ (6 values) for σ and the values $[2^{-5}, 2^{-3}, \ldots, 2^{15}]$ (11 values) for C. The results are reported upon an 80:20 split of train-test on the duplicates reported by the first-cut duplicate detection step on the new dataset with a 10-fold cross-validation on the train part to select the best parameters. The experiments on all datasets lasted from a few minutes to less than an hour at most.

6.3 MDedup performance evaluation

In this section, through a set of experiments, we show the potential of using MDCs as a duplicate detection classifier. Figure 5 serves as our evaluation summary and guideline for the three experiments discussed afterwards. The figure shows for each dataset four different effectiveness results that correspond to different steps in the MDedup algorithm. Each result is evaluated with the classification metrics recall, precision, and F-measure.

MDC Selection and *MDC Selection with boosting* are two variations of the training pipeline. Hence, they both utilize the gold standard to obtain their results. Because MDedup's training pipeline identifies the highest scored MDC for duplicate detection, the idea is the following: If we consider the selected MDC as a duplicate detection classifier, this classifier should provide us with the optimal F-measure *any* matching dependency combination (from the set of discovered MDs) can achieve; it, hence, represents an upper bound for our experiments and describes the capabilities of discoverable MDs in the domain of duplicate detection – we cannot predict any better results using MDs alone. To improve the recall of the best MDC, we can also apply the boosting step to its results. The version with boosting follows the principles we discussed for the application pipeline, i.e., the first-cut duplicate detection produces a set of duplicate pairs, which are then fed into an SVM classifier for training and a refined duplicate detection pass.

Analogous to the results of training pipeline, *MDC Prediction* and *MDC Prediction with boosting* present the results of the application pipeline. They describe MDedup's effectiveness when predicting good MDC classifiers rather than testing them on a gold standard. Training and testing of the regression model to predict the MDC scores is done in a leave-one-out cross-validation manner: Given one dataset, we train on all other datasets (and their gold standards) and test the prediction effectiveness on the given dataset (for which we omit the annotations).

As an attempt to create a baseline, we first considered possible methods that can also exploit the knowledge gathered on annotated datasets to classify record pairs in another target dataset that lacks an annotation. Transfer learning, however, for typical binary classification is not trivial, as different datasets have different attribute domains. Thus, although some literature exists on transductive transfer learning using SVMs [35], it is not clear how to perform such a schema matching across datasets in an automatic way. Thus, we experimentally enforced a schema mapping by splitting the same dataset into multiple parts and, then, ran both MDedup and [21] on these perfectly matching parts. We did this for all our datasets and measured on average 51% F-measure for us and 68% for [21]. So for same domain datasets with a transitive gold standard and a schema mapping, more effective approaches exist.

We then applied a traditional classification technique that directly uses the gold standards to train and apply SVM models. The results of training directly on high-quality and large sets of similarity pairs yielded, as expected, much higher precision and recall numbers. But such training approaches are incomparable to our approach and, in particular, inapplicable to our no-gold-standard scenario.

Subsequently, we discuss three experiments in more detail: First, in Section 6.3.1, we test the general effectiveness of (discovered) MDCs when being used for duplicate detection. Second, Section 6.3.2 compares the classification effectiveness of the predicted MDCs against best selected MDCs. Third and finally, Section 6.3.3 shows the F-measure scores that MDCs can achieve with boosting.

6.3.1 MDC Selection

First, we evaluate the limits of MDs for classifying duplicates in the presence of a gold standard, using a top-k approximation. This corresponds to the training phase of our pipeline in Figure 1, which approaches the best-case scenario for a given dataset. More specifically, we discuss the results shown in Figure 5 with the label *MDC Selection*.

Overall, the F-measure for most datasets is low. This is primarily due to poor recall – precision is near perfect on all datasets except Census and Hotels. The poor recall is mainly due to some error patterns that are not described by the discovered, minimal matching dependencies. However, the fact that the selection favors precision over recall has interesting implications: First, it shows that properties of good MDCs are learnable and transferable, because the majority of MDCs in general has a very poor precision. Second, a high precision is important for automatic approaches, because although they might not find all duplicates, their actual findings are reliable. The results are, finally, also good for boosting, because training subsequent models requires reliable test records (and not all records).

6.3.2 MDC Prediction

Our second experiment evaluates how well MDedup can predict the MDC classifiers for duplicate detection, which is its ability to detect duplicates in new datasets. For this purpose, we discuss the measurements labeled as *MDC Prediction* in Figure 5 and compare them to the target values of *MDC Selection*, which are the best F-measure values the system can possibly achieve.

The measurements in Figure 5 show that the predicted MDCs do not achieve the F-measure scores of the selected MDCs; in particular, MDedup never predicted the best selected MDC. Considering the large search space and the fact that most MDCs perform poorly as duplicate classifiers, the

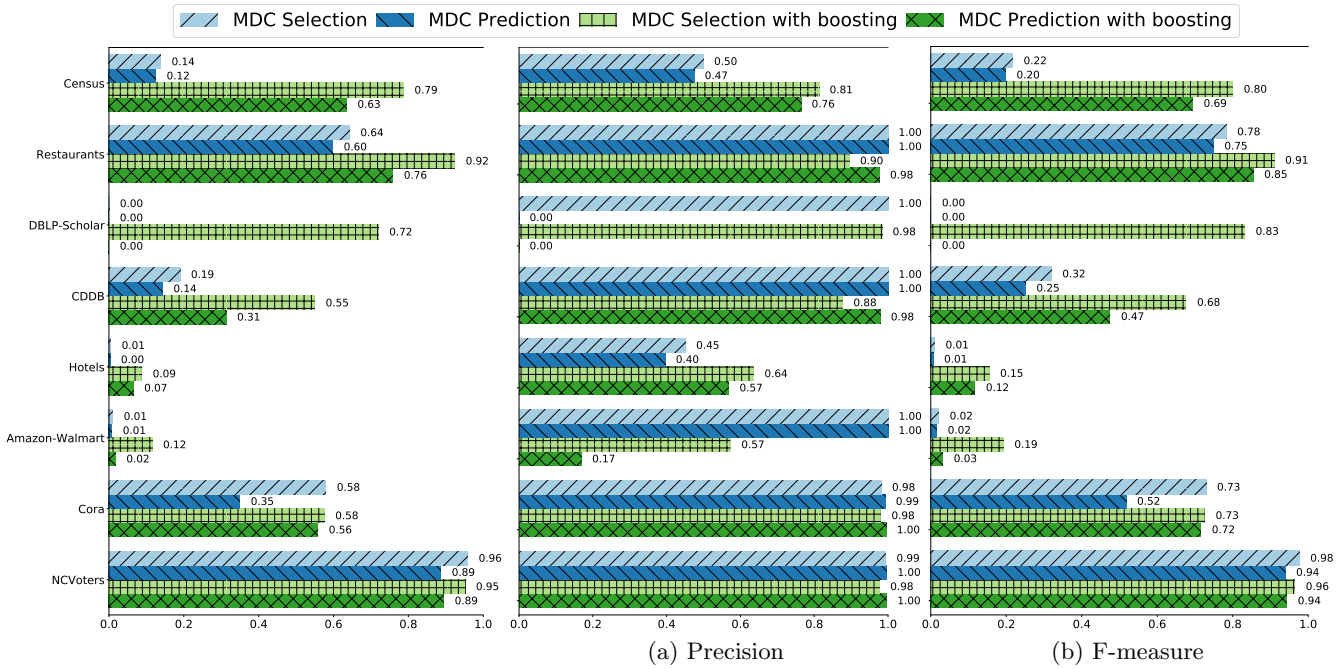

Figure 5: Effectiveness of MD-based duplicate classifiers using the best MDC according to a gold standard (Selection) or a Gaussian process (Prediction); in both cases, no-boosting and boosting is considered.

results are still very good. The predicted MDCs have, in particular, high precision and are, therefore, like their selected counterparts well suited for *automatic* cleaning processes and *boosting*. Overall, we primarily lose recall when using the MDC scoring model instead of a gold standard.

Unfortunately, MDedup failed to predict a useful MDC for the DBLP-Scholar dataset, because the dataset offers only one useful MD. When selected and boosted, it achieves 83% F-measure. The MD does not look promising on its own and all interesting insights are derived from the correctly identified duplicates in the boosting. Hence, it is no surprise that our system cannot identify the MD reliably.

6.3.3 Boosting

In the last experiment, we evaluate the boosting step of MDedup. The boosting step aims to improve the overall F-measure with an additional SVM model. This model is trained on the high-precision duplicates that the system generated with the selected/predicted MDCs and should be able to learn additional, domain-specific duplicate properties. To evaluate the boosting effect, we apply this step on top of both *MDC Selection* and *MDC Prediction*. The results, which are presented in Figure 5 with the *with boosting* suffix, show considerable improvements in F-measure for both selected and predicted MDCs; only Cora's and NCVoters' selected results lost one percent F-measure, which is due to slightly worse recall values. For this reason, boosting is a generally viable technique when using discovered MDs for duplicate detection.

Remarkably, datasets that performed poorly in the previous phases, have the largest improvements. More specifically, Amazon-Walmart, DBLP-Scholar, and Hotels were able to match only a few duplicate pairs beforehand. However, the combination of these duplicates, along with a few non-duplicates (as discussed in Section 5) was enough for an SVM-based approach to identify differences in the set of evaluated pairs. As expected *MDC Selection* provided SVMs with a noticeably better set of duplicates, which resulted in higher improvements and higher values of F-measure overall.

7. CONCLUSION

We proposed a system called MDedup that uses discovered matching dependencies for the *fully automatic* detection of duplicates, which means that no domain knowledge about or training data for a given target dataset is needed to detect duplicates in it. Because the duplicates discovered with only MDs usually have high precision but comparatively low recall, we proposed an SVM-based boosting ensemble step that in many cases greatly improves recall and, hence, the overall F-measure. Our experiments highlight the capabilities of discovered MDs when being used for duplicate detection, thus closing a gap in research. They also show that an algorithm can, to a certain extent, domain-independently learn how to score MDs for being good duplicate classifiers.

The achieved F-measure scores are certainly not competitive with those produced by algorithms that can learn on pre-labeled data or gold standards, but they are very useful for scenarios where no training data is available. The domain-independent properties of MDCs allow us to recommend their application, independent of a gold standard. No other duplicate detection approach is able to process arbitrary datasets without a human providing domain-specific input, such as initial classifier rules, data labels, or similarity thresholds. The usage of MDs in our approach, finally, also allows the user to interpret the detected duplicates.

8. REFERENCES

[1] Peter Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer-Verlag Berlin Heidelberg, 2012.

[2] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(1):1–16, 2007.

[3] Mikhail Bilenko and Raymond J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, Knowledge Discovery and Data Mining, pages 39–48, 2003.

[4] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. Distributed representations of tuples for entity resolution. *PVLDB*, 11(11):1454–1467, 2018.

[5] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. Deep learning for entity matching: A design space exploration. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 19–34, 2018.

[6] Jiannan Wang, Guoliang Li, Jeffrey Xu Yu, and Jianhua Feng. Entity matching: How similar is similar. *PVLDB*, 4(10):622–633, 2011.

[7] Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. Synthesizing entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017.

[8] Jaffer Gardezi, Leopoldo Bertossi, and Iluju Kiringa. Matching dependencies: semantics and query answering. *Frontiers of Computer Science*, 6(3):278–292, 2012.

[9] Zeinab Bahmani, Leopoldo E Bertossi, Solmaz Kolahi, and Laks VS Lakshmanan. Declarative entity resolution via matching dependencies and answer set programs. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 380–390, 2012.

[10] Hanna Köpcke and Erhard Rahm. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering*, 69(2):197–210, 2010.

[11] Peter Christen. Febrl: a freely available record linkage system with a graphical user interface. In *Proceedings of the Australasian Workshop on Health Data and Knowledge Management (HDKM)*, pages 17–25, 2008.

[12] George Papadakis, Leonidas Tsekouras, Emmanouil Thanos, George Giannakopoulos, Themis Palpanas, and Manolis Koubarakis. The return of JedAI: end-to-end entity resolution for structured and semi-structured data. *PVLDB*, 11(12):1950–1953, 2018.

[13] Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. Ajax: An extensible data cleaning tool. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, page 590, 2000.

[14] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. Swoosh: a generic approach to entity resolution. *VLDB Journal (VLDBJ)*, 18(1):255–276, 2009.

[15] George Papadakis, Georgia Koutrika, Themis Palpanas, and Wolfgang Nejdl. Meta-blocking: Taking entity resolution to the next level. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 26(8):1946–1960, 2013.

[16] Patrick Lehti and Peter Fankhauser. Unsupervised duplicate detection using sample non-duplicates. In *Journal of Data Semantics (JoDS)*, pages 136–164. Springer, 2006.

[17] Peter Christen. Automatic record linkage using seeded nearest neighbour and support vector machine classification. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 151–159. ACM, 2008.

[18] Munir Cochinwala, Verghese Kurien, Gail Lalk, and Dennis Shasha. Efficient data reconciliation. *Information Sciences*, 137(1):1–15, 2001.

[19] Zeinab Bahmani, Leopoldo Bertossi, and Nikolaos Vasiloglou. ERBlox: combining matching dependencies with machine learning for entity resolution. In *International Conference on Scalable Uncertainty Management (SUM)*, pages 399–414. Springer, 2015.

[20] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid training data creation with weak supervision. *PVLDB*, 11(3):269–282, 2017.

[21] Sahand Negahban, Benjamin I. P. Rubinstein, and Jim Gemmell. Scaling multiple-source entity resolution using statistically efficient transfer learning. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2224–2228, 2012.

[22] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. Relaxed functional dependencies: a survey of approaches. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(1):147–165, 2016.

[23] Wenfei Fan. Dependencies revisited for improving data quality. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 159–170, 2008.

[24] Shaoxu Song and Lei Chen. Discovering matching dependencies. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1421–1424. ACM, 2009.

[25] Shaoxu Song and Lei Chen. Efficient discovery of similarity constraints for matching dependencies. *Data and Knowledge Engineering (DKE)*, 87:146–166, 2013.

[26] Metanome algorithm repository. `https://github.com/HPI-Information-Systems/metanome-algorithms`. [Online; accessed 2-January-2020].

[27] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[28] Paul Jaccard. Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37:241–272, 1901.

[29] Ye Nan, Kian M. Chai, Wee S. Lee, and Hai L. Chieu.

Optimizing f-measure: A tale of two approaches. In John Langford and Joelle Pineau, editors, *International Conference on Machine Learning (ICML)*, pages 289–296, 2012.

[30] Michele Banko and Eric Brill. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the Annual Meeting on Association for Computational Linguistics (ACL)*, pages 26–33, 2001.

[31] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Springer-Verlag Berlin Heidelberg, 1st edition, 2001.

[32] Michael Collins, Robert E Schapire, and Yoram Singer. Logistic regression, adaboost and bregman distances. *Machine Learning*, 48(1-3):253–285, 2002.

[33] Smile - statistical machine intelligence and learning engine. `https://haifengl.github.io/smile/quickstart.html`. [Online; accessed 1-July-2019].

[34] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. *National Taiwan University, Taipei*, 2003.

[35] Zhaohong Deng and Shitong Wang. Novel inductive and transductive transfer learning approaches based on support vector learning. In *Support Vector Machines Applications*, pages 49–103. Springer, 2014.