Efficient and Effective On-line Learning-Based Instance Matching over Heterogeneous Data

Samur Araujo [#1], Duc Thanh Tran [*2], Arjen de Vries [#3]

[#]*Delft University of Technology, PO Box 5031, 2600 GA Delft, the Netherlands*
[1]s.f.cardosodearaujo@tudelft.nl
[3]a.p.devries@tudelft.nl

[*]*Karlsruher Institute of Technology, Germany*
[2]ducthanh.tran@kit.edu

***Abstract*— This document gives formatting instructions for authors preparing papers for publication in the Proceedings of an IEEE conference. The authors must follow the instructions given in the document for the papers to be published. You can use this document as both an instruction set and as a template into which you can type your own text.**

I. Overview

In this section, we introduce the data model, discuss the problem, and finally, present a brief overview of existing solutions as well as our approach.

A. Data

We focus on heterogeneous Web data including relational data, XML, RDF and other types of data that can be modeled as graphs. Closely resembling the RDF data model, we employ a graph-structured data model where every dataset is conceived as a graph $G \in \mathbb{G}$ comprising a set of triples:

Definition 1 (Data Model): A dataset set is a graph G formed by a set of triples of the form (s, p, o) where $s \in U$ (called subject), $p \in U$ (predicate) and $o \in U \cup L$ (object). Here, U denotes the set of Uniform Resource Identifiers (URIs) and L the set of literals. Every literal $l \in L$ is a bag of tokens, $l = \{t_1, \ldots, t_i, \ldots, t_n\}$, drawn from the vocabulary V, i.e. $t_i \in V$.

With respect to this model, *instances* are resources that appear at the subject position of triples. An instance representation can be obtained from the data graph as follows:

Definition 2 (Instance Representation): The instance representation $IR : U \times \mathbb{G} \rightarrow \mathbb{G}$ is a function, which given an instance $s \in U$ and a graph $G \in \mathbb{G}$, maps s to a set of triples in which s appears as the subject, i.e. $IR(s) = \{(s, p, o) | (s, p, o) \in G, o \in L\}$.

Thus, an instance is basically represented through a set of *predicate-value* pairs, where values are bags of tokens. @TODO: add a graph to provide example, also, provide one example for instance representation We will use the terms instance and instance representation interchangeably from now on. Note that for the sake of presentation, only the outgoing edges (s, p, o) of an instance s are considered while incoming edges (triples where s appears as the object) can also be added to the representation of s.

B. Problem - Find Instance Matches and Match Candidates

Instance matching is about finding instances that refer to the same real-world object based on their representations extracted from the data. In this paper, we tackle the problem of *instance matching across multiple datasets*: given instances of a *source dataset* G_s, the problem is to find instances in other datasets, collectively referred to as the *target dataset* G_t, which represents the same real world object.

Challenges. Compared to the single dataset setting, this problem entails additional challenges especially when the data is heterogeneous not only at the data but also schema level. In this regard, *data-level heterogeneity* means that for the same property, instances referring to the same object may have different values, or different syntactical representations of the same value. @TODO: For instance, the values "Michael Jackson" or "Jackson, Michael" can be both used as the value of the property name of an instance representing Michael Jackson. (1) *Schema-level heterogeneity* arises when there is only little or no schema overlaps between the datasets. That is, instances referring to the same object may be represented by different predicates, or different representations of the same predicates. Finding different representations of the same predicate is part of a problem also known as schema matching. Another challenge in this multiple dataset setting is (2) *efficiency*: for instance matching, a scheme is needed to determine how to compare two given instances; we will show that especially with schema heterogeneity, finding the best scheme for matching across heterogeneous datasets involves a greater search space and more feedback information (training data). Searching through all possible solutions and obtaining feedback information to evaluate them is expensive especially when we consider the online learning of instance matching schemes that requires access to data from remote endpoints. @TODO: discuss in intro that we need schemes for individual instances, and motivates and explain the concept of online learning of instance schemes: involves live access to remote endpoints that host fresh version of the datasets.

Computing Instance Matches. More precisely, an *instance matching scheme* is a (weighted) combination of similarity function predicates, $\sum_i w_i \sim (p_i) > \alpha$. Every $\sim (p_i)$ is a function, which given two instances s_i and s_j, returns the

similarity between these instances based on their similarity on the values of the predicate p_i. The scheme computes the overall similarity between s_i and s_j by combining the similarities obtain for individual predicates and determines them as a *match* when its exceeds the threshold α. Clearly, such a scheme focus on data-level heterogeneity. Given s_i and s_j are in the source and target datasets, respectively, and these datasets vary in schema, an extended scheme of the form $\sum_i w_i \sim (\langle p_m^s, p_n^t \rangle_i) > \alpha$ is needed to capture that the values of the source predicate p_m^s shall be compared with values of the target predicate p_n^t. In other words, when the predicates in the source and target are not the same, comparable pairs of predicates have to be found and incorporated into the scheme. In fact, this problem entails the subproblems of (A) finding the pair of comparable predicates $\langle p_m^s, p_n^t \rangle$ (schema matching) as well as (B) choosing and (C) weighting them, and determining the (D) similarity functions $\sim$ (e.g. Jaccard distance) and (E) thresholds α.

Computing Instance Match Candidates. Instead of solving all these problems, some instance matching solutions focus on the blocking step, which aims to quickly select candidates matches (hence also referred to as the *candidate selection* step). Instead of using a combination of similarity function predicates, candidate selection simply employs a (conjunction of) blocking key(s), i.e. $\bigwedge \sim (\langle p_m^s, p_n^t \rangle_i)$ (called *candidate selection scheme*), where $\sim$ is a binary function that returns whether the two values of p_m^s and p_n^t match or not. Here, the predicates p_m^s and p_n^t constitute the pair of comparable blocking keys while their values are called *blocking key values* (BKV). Usually, the similarity function is based on exact value matching or value overlap. That is, two instances form a *candidate match* if their blocking key values are the same or overlap on some tokens. Mostly, $\sim$ is defined manually such that candidate selection amounts to the problem of (A) finding comparable predicate pairs and (B) choosing the most selective ones as blocking key pairs. Often, candidate selection is performed as a preprocessing step, producing results that are further refined by a more effective instance matcher that also tunes the weights and threshold to obtain better results.

C. Existing Solutions

State-of-the-art matching systems are based on supervised learning, leveraging training data as feedbacks to evaluate candidate schemes [1]. Basically, optimal schemes found are those which maximize the coverage of positive examples while avoiding negative examples. They are geared towards homogeneous datasets, focusing on the learning of schemes of the type $\sum_i w_i \sim (p_i) > \alpha$ as discussed above. It has been shown that in fact, the underlying learning strategies can also be used to obtain the extended schemes $\sum_i w_i \sim (\langle p_m^s, p_n^t \rangle_i) > \alpha$. Instead of all combinations of individual predicates, the search space would have to include all combinations of all possible pairs of predicates.

Since obtaining representative training data across datasets is difficult, recent approaches that specifically target heterogeneous data derive schemes directly from the data. However, unsupervised approaches of this kind focus on the more simpler problem, namely the learning of the candidate selection schemes $\bigwedge \sim (\langle p_m^s, p_n^t \rangle_i)$. For instance, Song and Heflin [2] assume precomputed schema mappings such that the comparable predicate pairs are known. They propose to choose them based on their coverage and discriminability; two metrics derived from the data basically reflecting the number of instances a given predicate can be applied to and how well it distinguishes them. Based on manually defined coverage and discriminability threshold, the best pairs of comparable blocking keys are selected.

As an alternative, a schema-agnostic approach [] has been proposed for candidate selection. It does not use predicates for matching but treat instances simply as bags of value tokens. Instances form matches when they have some value tokens in common. Therefore, instances which share the same token (in any predicate) are placed in one candidate set. This approach does not require any effort for learning the scheme and is particularly suited when there is a lack of schema overlap such that only few or no comparable predicates exist. The problem with this is that the candidate sets produced are highly redundant because instances are often placed in multiple candidate sets. Consequently, this work employs much more additional processing to further refine these candidate sets.

D. Existing Solutions vs. Our Solution

In this work, we tackle the problem of instance matching. However, we focus on the problem of learning the candidate selection scheme and simply use the resulting scheme in combination with an existing matcher to refine candidate results.

It has been shown that the use of schema knowledge improves precision but considerably decreases the coverage of correct matches [] (recall). In this work, we propose a supervised learning strategy to incorporate predicate information into the candidate selection scheme that considerably improves both measures compared to this previous work [].

The main difference between this and both the existing supervised and unsupervised learning strategies lies in the granularity of the learned scheme. Instead of using one scheme for all instances, our solution may yield different schemes for different source instances. This separate treatment of instances is introduced to specifically deal with the heterogeneity problem: @TODO: concrete example, showing that different schemes are needed for two source instances. Using these more fine-grained schemes, we show that the quality of results produced by our approach is superior than those produced by existing supervised [1] and unsupervised approaches [2].

Another aspect that has been neglected so far is time efficiency. More precisely, works on finding the scheme as discussed above focus on the quality of matches. On the other hand, there are works on executing similarity joins and building blocking indexes, focusing on how to process the schemes efficiently. In other words, more emphasis is put on the efficiency of execution and less on the efficiency of learning. @TODO: stress the fact that efficiency is crucial

for online learning, minimize data access over remote endpoints. Further, how well a scheme can be optimized for time efficiency depends on the nature of the scheme itself. Some schemes are inherently expensive, requiring a large amount of data to be loaded and to be joined. Further, schemes that produce the same result quality may vary in terms of runtime efficiency. Thus, optimized execution performance cannot be achieved independent of learning. In this work, we consider the entire process of learning and execution. We consider time as an additional optimization criteria such that optimal schemes are those, which (a) can be learned quickly, (b) can be executed efficiently and (c) yield high quality candidates. We show that this holistic optimization of time efficiency leads to faster execution. In fact, the entire process of learning and execution is faster compared to the unsupervised approach, which requires almost no time in learning. We also compare the performance results for the entire process with the supervised approach, which requires training data to be locally available (i.e. offline learning instead of online learning over remote endpoints). Despites the overhead of retrieving data over endpoints, we show that our approach yields competitive performance.

E. Our Solution

To consider this instance specific schema, we do not consider the solution for instance matching as weight of predicates, but rather, we consider it as a query problem for every instance. Therefore, a candidate set is the results of candidate template query that we define next:

We noticed there are two problem with the existing approaches. First, they do not consider the time dimension of the problem, specially when we consider the scenario where there are remote endpoints. Basically, to learn the schema on these remote endpoints means that we have to execute a lot of queries, which takes a lot of time.

Definition 3 (Template Query): A template query Q is conjunction of n tuples (p, k), where p is a predicate in G_t and k is a token, defined as $\bigwedge_i^n (p_i, k_i) = \{t | (t, p_i, o) \in G_t \wedge o \sim k_i\}$. The similarity function $\sim$ is given. Without loss of generality we can also assume that can exist tuples where p are undefined, acting as a wild card.

Definition 4 (Candidate Set): A candidate set of an instance s in G_s is a instantiation of a template query Q where we have at least a tuple (p, k) where $(s, x, k) \in IR(s)$

OPTIMIZATION PROBLEM

- Describe the optimization goal. to be efficient and effective on building candidate sets.
- Describe the effective part of the problem
- Due to heterogeneity of the data, there is no unique template query that works for all instances. Therefore, we need to find a query template for each specific instance.
- Describe the optimal criteria for those template queries.
- For a specific instance, its optimal query template should avoid negative matches and include all positive matches in the candidate set.
- Describe the problem of building those optimal queries.
- Describe the efficiency part of the problem.
- Due to the large number of queries templates, we can not evaluate all queries because it takes too much time. Therefore, we need to be efficient on selecting only the queries that has the highest chance to retrieve a optimal candidate set. We should ignore queries that perform badly.

  We pose the problem of candidate selection as a optimization problem, where, for each source instance s, the goal is to find a template query that selects all positive matches for s, avoiding negative matches. Without lost of generality, we can assume that every source instance maps to a target instance (a 1-to-1 mapping), consequently, the optimal query for this problem would retrieve a candidate sets with one element. For now, we consider that an oracle can decide if the match is correct or not. Due the heterogeneity of the data, there is no unique template query that works for all instances. Therefore, to be effective, we need to find a template query for each specific source instance. As the number of those queries may be large, we do not want to evaluate all to find the optimal query, because it is time prohibitive; specially when we have to query a remote endpoint. Therefore, for every instance, we need to be time efficient by evaluating only the queries that has the highest change to be optimal.

SOLUTION

- Describe how we solve this optimization problem.
- Describe that we use a iterative process because we need to refined the templates queries during the process
- Describe how the process starts
- Describe how we solve the efficiency part of the problem
- Describe that we need a set of heuristic for efficiently select the best queries.
- Describe how this heuristic are used in the branch-and-bound framework
- Describe how we solve the effective part of the problem
- Describe that we learn the comparable predicates from the data
- Describe that we refined the queries during the process
- Describe that we use a matcher to generate positive and negative examples
- Describe how those examples are used to refined the queries.

  Basically, we tackle this problem in two stages. First, we build all possible effective template queries by sampling the source and target data. In this process we learn the most discriminative comparable predicates and each pair becomes one clause template query. Then, for each source instance, we use a branch-and-bound optimization algorithm that searches for those queries that have the highest chance to retrieve a optimal candidate set, through tree-structured space compose of all found template queries.
  We start the process with a initial set of template queries. Aiming to approximate our solution to the optimal solution, we approach this problem in an iterative fashion, where at each iteration we make use of a set of policies

to decide whether or not evaluate a query. Mainly, those policies are based on queries results obtained on previous iterations, and their aim is to select the queries that have the highest chance to be optimal for next iterations, therefore this process minimizes the number of queries performed. Generally, the most selective queries are selected, which are more efficient and effective, because they select less elements (and it takes less time); and they select less incorrect matches. To be effective, at each iteration we refine the template queries, building highly selective template queries with respect to our optimal criteria (maximize positive and minimize negative matches). Basically, this refining process adds another clause in the template queries, called class clauses. To build class clauses, we apply a matcher over the already generated candidate sets obtaining positive and negative matches that are input to an algorithm that output a set of class clauses, which are those predicate/value pairs that select only the positive matches. The matcher can be any approach that uses a more complex similarity measure to select the correct matches (positive examples) among the possible candidates.

This algorithm aims to find a best path of queries, representing a minimal set of time-efficient queries that produce high quality results.

II. Branch-and-Bound Optimization

- Describe the motivation to use the branch-and-bound to approach the efficiency issues (execute all queries are costly)
- The idea is to execute the minimum amount of queries.
- Describe the heuristic used to determined when evaluate a query
- Describe how those heuristic are used in the framework
- Describe how we tackle the time issue. (reordering queries).
- Describe the moment that attribute queries are generated. (at the beginning of the process)
- Describe the moment the queries with class clauses are generated. (they are generated when the cardinality of candidates are above a threshold)
- Describe how the predictor can increase the efficiency by skipping queries
- Describe how we train the classifier. it is done automatically after it converges.

III. Constructing Template Queries

In this section we describe how we construct the template queries in our approach. Basically, we want to avoid queries composed by too many clauses, because they are too selective, and they end up missing some positive matches. Therefore, in this work, we investigate template queries that are composed at most of two clauses; namely, an attribute clause and a class clause. The attribute clause helps us to find candidates based on the assumption that positive matches share a similar token on a pair of highly discriminative predicates. The class clause help us to disambiguate candidates that share the same token but belong to different classes. We only build template queries that contains an attribute clause or an attribute clause and an class clause.

An attribute clause denoted by $A(p_t, o_s)$ is derived from the comparable predicate pair (p_s, p_t) where the triple (s_i, p_s, o_s) exists in G_s. To build attribute clauses, we use known algorithms to determine the best pair of comparable predicates. Highly discriminative pairs are desired. The discriminative property guaranties that the clause will select a few candidates, impacting in the overall precision. Among those predicates, the set that cover all the positive match are used in process. The coverage property avoids that we miss positive matches, impacting the overall recall. To find the source predicates, we apply this algorithm over a subset of the source instances, which is quite straightforward. To find the target predicates and align them with the source predicates to form pairs is much less obvious; specially because we need to query the target endpoint to collect the data. A reasonable approach to get relevant data is to use the values of the selected source predicates to query the target endpoint. Then, we apply the algorithm over those candidates to determine the target predicates. Afterwards, we map the source and target predicates that their values are similar above a specific threshold. Notice that we do not use any schema information in this process, because in the heterogeneous setting, the schemas may not align. The final set of predicate pairs that we found is then transformed in a set of template queries containing one clause (one for each predicate pair).

In this work, we assume that the source instances to be matched belong to the same class (e.g. country, people, drugs, etc.). This assumption help us to find the best comparable predicates in two ways. First, because instances that share the same class are less diverse in the number of predicates; consequently, it is easier to find the ones with the highest coverage and discriminability. Second, the target instances for those source instances will also belong to a limited set of target class. Therefore, as both source and target sample are less diverse, we can with much less effort produce attribute clauses with the highest coverage and the maximal selectivity.

Beside of that, we can only produce class clauses if we consider this assumption. An class clause denoted by $C(p_t, o_t)$ is derived from a triples (s_j, p_t, o_t) that exists in G_t. To build class clauses, we use a set-cover based algorithm [4] to quickly retrieve the list of predicate/value pairs that select all positive elements but avoid the negative ones. The positive and negative matches are obtained during the candidate selection process that we will detail further.

IV. Evaluation

- Describe the datasets
- Data preparation (indexes)
- Describe the metrics
- Describe alternative approaches
- Results for candidate selection
- Results for instance matching

In this section, we discuss how we evaluate our approach and discuss about the results. Our system Sonda was implemented in Ruby and the queries were implemented as SPARQL queries issues over alive SPARQL endpoints. Sonda is available for download at GitHub as a command line tool [1], as well as all the results that we obtained.

A. Datasets

We evaluated our framework using the datasets and ground truth published by the instance-matching benchmark of the Ontology Alignment Evaluation Initiative (OAEI) [**?**]. We used the datasets provided in 2010 and 2011. We used the life science (LS) collection (which includes Sider, Drugbank, Dailymed TCM, and Diseasome) and the Person-Restaurant (PR) from the 2010 collection. We excluded LinkedCT from our experiments due to known quality problem in the ground truth. We used all datasets from the 2011 collection.

B. Querying Candidates

We implemented the queries in our algorithm as SPARQL queries (as discussed before) and directly query a SPARQL endpoint to obtain results (limit to 30 instances per query). For that, we loaded all datasets into the OpenLink Virtuoso Universal Server (Version 6.1.5.3127), except for DBPedia, which we queried its on-line sparql endpoint. We use the default S-P-O index created by this server, and created an inverted index for literal values using the following commands:

```
DB.DBA.RDF_OBJ_FT_RULE_ADD
(null, null, 'index_local');
DB.DBA.VT_INC_INDEX_DB_DBA_RDF_OBJ ();
```

We use the specific Virtuoso SPARQL implementation to have access to the index, and we limited all query results to 30 instances. This avoids the queries to retrieve too many data for non-discriminative queries. For example, in this syntax, the 4 query types EXACT, LIKE, AND, and OR are, respectively:

```
SELECT DISTINCT ?s WHERE {?s ?p
'eosinophilic pneumonia' .}
limit 30

SELECT DISTINCT ?s ?o WHERE {?s ?p ?o .
?o bif:contains '"eosinophilic pneumonia"'
. } limit 30

SELECT DISTINCT ?s ?o WHERE {?s ?p ?o .
?o bif:contains '"eosinophilic"AND"pneumonia"'
. } limit 30

SELECT DISTINCT ?s ?o WHERE {?s ?p ?o .
?o bif:contains '"eosinophilic"OR"pneumonia"'
. } limit 30
```

C. Evaluation metrics and alternative approaches

We used standard metrics, namely Reduction Ratio (RR), Pair-wise Completeness (PP) and F1. Basically, high RR means that the candidate selection algorithm helps to focus on a smaller number of candidates, while high PP means that it could preserve more of the correct candidates. Because RR is small given the number of all possible candidates is large in this scenario, we use a normalized version of RR. In particular, these metrics are computed as follows: PC = # Correctly Computed Candidates / # Ground Truth Candidates; RR = # Instances with Non-Empty Candidate Sets / # All Computed Candidates; $F1 = \frac{2*RR*PC}{RR+PC}$. Beside these metrics, we also count the average number of queries evaluated per instance as well as overall time for accomplishing the task of finding the candidate sets.

For comparison, we implemented the *S-agnostic* [**?**] and *S-based* [2] approaches as discussed in Sec. 2. S-based uses only an OR query and it does not feature the branch-and-bound optimization. It requires key pairs, which are generated as in Sec. 3.1. Further, S-based applies a similarity function on the keys to further prune incorrect candidates after that have been retrieved using the OR queries. For comparison purposes, we apply this strategy to all approaches, using the same similarity function. Sonda uses four types of queries for each key pair, and employs the proposed branch-and-bound optimization to select best queries.

D. Results

Table 1 shows the results. Comparing all approaches over all the 16 datasets pairs that we evaluated, Sonda achieves the best F1 score in all cases (in 96% of the cases, to be precise), except for the NYT-Geonames pair, where S-based has best F1 result (due to high RR).

@TODO: add table of results

In the NYTimes-Freebases case, Sonda achieves a considerable improvement in both RR and PC. The other approaches perform worse in this case because the comparable key pairs and queries they use are not discriminative, producing too many matching instances. Sonda also considers many comparable key pairs, thereby ensuring high PC (reflecting recall). However, not all queries generated from them are used to obtain candidates but only optimal ones found during the process. This helps to balance PC with RR (reflecting precision).

For Sider-Dailymed, we could not obtain the results for S-agnostic because it took more than 10,000 seconds to compute it. This may be attributed to the fact that the Dailymed dataset contains a large number of textual attributes and the queries generated by S-agnostic are evaluated over all these attributes, resulting in a large number of disk accesses.

We can see that in all cases, Sonda achieves a considerable reduction in the number of nodes evaluated per instance. This means that many comparable key pairs are considered, including incorrect ones. These ones are not considered further (resulting in reduction of evaluated query nodes) as Sonda iterates through the instances to learn the predictor and to apply the branching policy.

Regarding time performance, even though Sonda has more queries to evaluate, it is faster than S-based in 56% of the cases; it is faster than S-agnostic in 37% of the cases. Sonda could achieve these results because it uses different types of

[1] https://github.com/samuraraujo/Experiments/tree/master/experiments/ICDE2013

queries with different time performances. During the process, its branch-and-bound algorithm helps to select the ones that require less evaluation time (those that are more efficient than the ones used by the other approaches). In particular, we observe that less queries does not directly translate to less execution cost. For instance, the OR queries for one token is 10 times slower than the EXACT and LIKE queries together for the same token. Although queries performance time may vary among RDF store implementations, Sonda processes the fastest first, a decision that is based on experiences acquired during the process.

V. Conclusions

References

[1] S. Chaudhuri, B.-C. Chen, V. Ganti, and R. Kaushik, "Example-driven design of efficient record matching queries," in *VLDB*, 2007, pp. 327–338.

[2] D. Song and J. Heflin, "Automatically generating data linkages using a domain-independent candidate selection approach," in *International Semantic Web Conference (1)*, 2011, pp. 649–664.

[3] G. Papadakis, E. Ioannou, C. Niederée, and P. Fankhauser, "Efficient entity resolution for large heterogeneous information spaces," in *WSDM*, 2011, pp. 535–544.

[4] R. D. Carr, S. Doddi, G. Konjevod, and M. V. Marathe, "On the red-blue set cover problem," in *SODA*, 2000, pp. 345–353.