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
1s.f.cardosodearaujo@tudelft.nl
3a.p.devries@tudelft.nl

*Karlsruher Institute of Technology, Germany
2ducthanh.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 $IR : U \times \mathbb{G} \to \mathbb{G}$ 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 - Computing Instance Matches and Instance 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.

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. 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.

More precisely, instance matching can be formulated as the problem of finding a (weighted) combination of similarity function predicates, $\sum_i w_i s_i(p_m, p_n) > \alpha$, which can be used to decide if a given pair of instances match (when the similarity exceeds the threshold α), or not. Every $s_i(p_m, p_n)$ is a similarity function, which given the values of the predicates p_m and p_n, returns a similarity score. This problem entails the subproblems of finding the comparable predicates p_m and p_n (schema matching), choosing and weighting these predicate pairs, as well as determining the similarity function s_i (e.g. Jaccard distance) and threshold α.

Often, instance matching is preceded by a 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. $\bigwedge s_i(p_m, p_n)$, where s_i is a binary function that returns whether the two values of p_m and p_n match or not. Here, the predicates p_m and p_n constitutes the pair of comparable blocking keys while their values are called *blocking key values* (BKV). Usually, the similarity function is fixed to be exact value matching or value overlap. That is, two instances form candidates if their blocking key values are the same or overlap on some tokens. In this step, there is no need of tuning any threshold. However, quite often, candidate sets are refined in a intermediary matching step by a matcher based on on approximate string matching over the BKVs.

Thus, candidate selection can be seen as sub-problem of instance matching, i.e. finding comparable predicate pairs and choosing the most selective as blocking key pairs. Fig. 1 depicts the whole instance matching process described here. This paper focus on the candidate selection part of the problem.

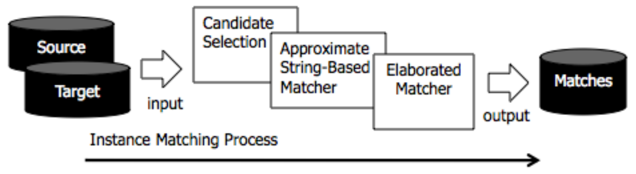

Fig. 1. Instance matching process overview.

C. Existing Solutions

Most of the approaches that tackle the candidate selection problem over heterogeneous data, where there are no overlaps between schemas, i.e. predicates in different datasets do not matches, uses an approximate string matching-based matcher to resolve the ambiguity(imprecision) on candidate sets that contains instances that share overlapping strings. One main class of solutions for this problem comprises supervised learning-based approaches, which uses training examples to learn the comparable predicates, the similarity functions and their thresholds.

For instance,[1] exploits the availability of positive and negative examples to search through this space and suggest an initial record matching query. Although they do not learn the comparable predicates, their algorithm find the combination of $s_i(p_m, p_n)$ and their respective threshold that produces the best candidate sets, i.e., those that include all positive matches and avoid the negative ones. As a limitation of this approach, it requires positive and negative examples that are hard to obtain, specially in the heterogeneous settings; moreover, this approach can be problematic when applied over remote data endpoints.

Another class of solutions for this problem comprises unsupervised learning-based approaches, which without training examples, try to learn the comparable predicates, their mapping, the similarity functions and thresholds, by only inspecting the data and schema.

The problem of learning/finding comparable predicates in this setting is presented in [2]. To determined the comparable predicates, they compute the coverage and the discriminability of all predicates, by counting the frequency that each predicate occurs in the data; then predicates with high discriminating score that are above are threshold are considered as comparable predicates. When individual discriminabilities are below the threshold, predicates are combined together to become more discriminative. The threshold is manual defined, and in the worse case can lead to an exponential combination of keys. Beside of that, the mapping between those source and target predicates are manually defined (another limitation of their work). Then, after the comparable predicates are manually mapped, the instances are indexed by the value of those predicates and the candidate sets are formed by searching on this index for overlapping tokens. Because candidate selection are generally imprecise, a approximate string-based matcher is applied to prune the candidates that are above a threshold. In this step, both the similarity function and the threshold are manually defined. Finally, the candidates sets generated are delegated to an arbitrary more elaborated matcher that finds the exact matches.

Another unsupervised solution is described in [3]. They describe a schema-agnostic approach to solve the sub-problem of candidate selection. In their approach, the learning of comparable predicate is unnecessary. They based on the idea that instances that refer to the same entity have at least one value in common, independently of the corresponding predicate names. Therefore, all instances that contain the same token in any predicate are placed in one candidate set (block, in their terminology). The approach is thus very robust against heterogeneity, noise, and loose schema binding. However, the candidate sets produced are highly redundant, because each instance is placed in multiple candidate sets. Consequently, a lot of processing is required to efficiently build precise candidate sets. Although, the authors states that in this setting the use of schema knowledge improves the precision but decreases considerably the coverage of the correct matches, we will show that the use of the predicates in the blocking keys can considerably improve the performance of both measures.

D. Our Solution

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. Second, due to the heterogeneity on the source and target data, where there are multiple instances from different classes, represented in different schemas that do not overlap, the learning of a general instance matching schema to deal with all those diversity is certainly not enough. 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:

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. Evaluation

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

IV. Constructing Template Queries

- Describe the template queries are compose of two type of clauses
- Describe the motivation for the use of attribute clauses
- Describe the problem with clauses with multiple predicates (to specific then recall is penalized)
- Describe the algorithm to find key predicates
- Describe how we sample the source and target data to find the key predicates
- Describe heuristic for sampling.
- Describe how processing highly discriminative instance first help to find better keys.
- Describe how the assumption that the sources instance are belong to the same class can speed up the generation of attribute clauses.
- Describe the algorithm to map comparable keys
- Describe the motivation for use class clauses (because the ambiguity at class level)
- Define what is a class clause. (a predicate/value pair that select positive and no negative examples.)
- Describe that we have to assume that instances belong to a same class.
- Then we case use the class based disambiguation to generate examples and those examples are used to find class clauses.
- Describe the algorithm to find those class clauses.

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.

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.