Complex Matching of RDF Datatype Properties

Bernardo Pereira Nunes[1,2], Alexander Mera[1], Marco Antônio Casanova[1], Besnik Fetahu[2], Luiz André P. Paes Leme[3], and Stefan Dietze[2]

[1] Department of Informatics - PUC-Rio - Rio de Janeiro, RJ, Brazil
{bnunes,acaraballo,casanova}@inf.puc-rio.br
[2] L3S Research Center - Leibniz University Hannover - Hannover, Germany
{nunes,fetahu,dietze}@l3s.de
[3] Computer Science Institute, Fluminense Federal University, Niterói, RJ, Brazil
lapaesleme@ic.uff.br

Abstract. Property mapping is a fundamental component of ontology matching, and yet there is little support that goes beyond the identification of single property matches. Real data often requires some degree of composition, trivially exemplified by the mapping of "first name" and "last name" to "full name" on one end, to complex matchings, such as parsing and pairing symbol/digit strings to SSN numbers, at the other end of the spectrum. In this paper, we propose a two-phase instance-based technique for complex datatype property matching. Phase 1 computes the Estimate Mutual Information matrix of the property values to (1) find simple, 1:1 matches, and (2) compute a list of possible complex matches. Phase 2 applies Genetic Programming to the much reduced search space of candidate matches to find complex matches. We conclude with experimental results that illustrate how the technique works. Furthermore, we show that the proposed technique greatly improves results over those obtained if the Estimate Mutual Information matrix or the Genetic Programming techniques were to be used independently.

Keywords: Ontology Matching, Genetic Programming, Mutual Information, Schema Matching.

1 Introduction

Ontology matching is a fundamental problem in many applications areas [10]. Using OWL concepts, *by datatype property matching* we mean the special case of matching datatype properties from two classes.

Concisely, an *instance* of a datatype property p is a triple of the form (s, p, l), where s is a resource identifier and l is a literal. A *datatype property matching* from a *source class* S to a *target class* T is a partial relation μ between sets of datatype properties from S and T, respectively. We say that a match ***(A,B)*** $\in \mu$ is m:n *iff* ***A*** and ***B*** contain m and n properties, respectively. A match ***(A,B)*** $\in \mu$ should be accompanied by one or more *datatype property mappings* that indicate how to construct instances of the properties in ***B*** from instances of the properties in ***A***. A match ***(A,B)*** $\in \mu$ is *simple* iff it is 1:1 and the mapping is a identity function; otherwise, it is complex.

H. Decker et al. (Eds.): DEXA 2013, Part I, LNCS 8055, pp. 195–208, 2013.
© Springer-Verlag Berlin Heidelberg 2013

In this paper, we introduce a two-phase, instance-based datatype property matching technique that is able to find complex n:1 datatype property matches and to construct the corresponding property mappings. The technique extends the ontology matching process described in [19] to include complex matches between sets of datatype properties and is classified as instance-based since it depends on sets of instances.

Briefly, given two sets, s and t, that contain instances of the datatype properties of the source class S and the target class T, respectively, the first phase of the technique constructs the Estimated Mutual Information matrix (EMI) [18,19] of the datatype property instances in s and t, which intuitively measures the amount of related information from the observed property instances. The scope of this phase is to identify simple datatype property matches. For example, it may detect that the "e-mail" datatype property of one class matches the "electronic address" datatype property of another class. Additionally, the first phase suggests, for the second phase, sets of candidate datatype properties that can be matched only under more complex relationships, thereby reducing the search space.

The second phase uses a genetic programming approach (GP) to find complex n:1 datatype property matches. For example, it discovers that the "first name" and "last name" datatype properties of the source class match the "full name" datatype property of the target class, and returns a property mapping function that concatenates the values of "first name" and "last name" (of the same class instance) to generate the "full name" value. The reason for adopting genetic programming is two-fold: it reduces the cost of traversing the search space; and it can be used to automatically generate complex mappings between datatype property sets.

The difficulty of the problem of finding complex matches between sets of datatype properties should not be underestimated since the search space is typically quite large. Therefore, our contribution towards a more accurate and efficient solution lies in proposing a two-phase technique, which deals with the problem of finding complex matches by: (a) using the Estimated Mutual Information matrix (in Phase 1) as a pre-processing stage, limiting the candidate sets of properties for complex matches; (b) adopting a genetic programming strategy to automatically generate complex property mappings. We also give empirical evidence that the combination of both approaches, EMI and GP, yields better results than using either technique in separate.

2 Background

2.1 Vocabulary Matching and Concept Mapping

We decompose the problem of OWL ontology matching into the problem of vocabulary matching and that of concept mapping. In this section, we briefly review these concepts and extend them to account for complex property matching. In what follows, let $\boldsymbol{S}$ and $\boldsymbol{T}$ be two OWL ontologies, and $\boldsymbol{V_S}$ and $\boldsymbol{V_T}$ be their vocabularies, respectively. Let $\boldsymbol{C_S}$ and $\boldsymbol{C_T}$ be the sets of classes and $\boldsymbol{P_S}$ and $\boldsymbol{P_T}$ be the sets of properties in $\boldsymbol{V_S}$ and $\boldsymbol{V_T}$, respectively.

An *instance* of a class c is a triple of the form $(s, rdf{:}type, c)$, an instance of an object property p is a triple of the form (s, p, o) and an instance of a datatype property d is a triple of the form (s, d, l), where s and o are resource identifiers and l is a literal.

A *vocabulary matching* between $\boldsymbol{S}$ and $\boldsymbol{T}$ is a finite set $\mu \subseteq \boldsymbol{V_S} \times \boldsymbol{V_T}$. Given $(v_1, v_2) \in \mu$, we say that (v_1, v_2) is a *match* in μ and that μ *matches* v_1 with v_2; a *property* (or *class*) *matching* is a matching defined only for properties (or classes).

A *concept mapping* from $\boldsymbol{S}$ to $\boldsymbol{T}$ is a set of transformation rules that map instances of the concepts of $\boldsymbol{S}$ into instances of the concepts of $\boldsymbol{T}$.

In this paper, we extend *vocabulary matchings* to also include pairs of the form ***(A,B)*** where $\boldsymbol{A}$ and $\boldsymbol{B}$ are sets of datatype properties in $\boldsymbol{P_S}$ and $\boldsymbol{P_T}$, respectively. We say that ***(A,B)*** is an $m{:}n$ match *iff* $\boldsymbol{A}$ and $\boldsymbol{B}$ contain m and n properties, respectively. In this case, a match ***(A,B)*** must be accompanied by *datatype property mappings*, denoted $\mu[\boldsymbol{A}, B_i]$, which are transformation rules that map instances of the properties in $\boldsymbol{A}$ into instances of the property B_i, for $i = 1, \ldots, n$, where $\boldsymbol{B}=\{B_1, \ldots, B_n\}$. Using "//" to denote string concatenation, the following transformation rule $(s, fullName, v) \leftarrow (s, firstName, n), (s, lastName, f), v = n//f$ indicates that the value of the "fullName" property is obtained by concatenating the values of properties "firstName" and "lastName". We will use the following abbreviated form for mapping rules with the above syntax:

$$\mu[\{firstName, lastName\}, fullName] = \\ \text{``}fullName \leftarrow firstName//lastName''$$

As an abuse of notation, when $\boldsymbol{A}$ is a singleton $\{A_1\}$, we simply write $\mu[A_1, B_i]$, rather than $\mu[\{A_1\}, B_i]$. Finally, a match ***(A,B)*** is *simple iff* it is 1:1, that is, of the form $(\{A_1\}, B_1)$, and the mapping $\mu[A_1, B_1]$ is the *identity transformation rule*, defined as "$(s, B_1, l) \leftarrow (s, A_1, l)$"; otherwise, the match is *complex*.

2.2 An Instance-Based Process for Vocabulary Matching

In this section, we very briefly summarize the instance-based process to create *vocabulary matchings* introduced in [19]. The outline of the process is as follows:

S1. Generate a preliminary property matching using similarity functions.
S2. Generate a class matching using the property matching obtained in S1.
S3. Generate an instance matching using the output from S1.
S4. Refine the property matching using the class matching generated in S2 and the instance matching from S3.

The final vocabulary matching is the result of the union of the class matching obtained in S2 and the refined property matching obtained in S4.

The intuition used in all steps of the process is that *"two schema elements match iff they have many values in common and few values not in common"*, i.e. *iff* they are similar above a given similarity threshold.

We obtain the following output from each individual step. S1 generates preliminary 1:1 property matchings based on the intuition that two properties match *iff* their instances share similar sets of values. In the case of string properties, their values are replaced by the tokens extracted from their values. S1 provides evidences on class and instance matchings, explored in the next two steps.

S2 generates class matchings based on the intuition that two classes match *iff* their sets of properties are similar. This step uses the property matchings generated in S1.

S3 generates instance matchings based on the intuition that two instances match *iff* the values of their properties are similar. However, equivalent instances from different classes may be described by very different sets of properties. Therefore, extracting values from all of their properties may lead to the wrong conclusion that the instances are not equivalent. Therefore, Leme et al. [19] propose to extract values only from the matching properties of the instances.

3 Two-Phase Property Matching Technique

In this section, we introduce a technique to partly implement and extend the ontology matching process of Section 2.2 to compute complex n:1 datatype property matches (note that the technique does not cover n:m matches). The technique comprises two phases: Phase 1 uses Estimated Mutual Information matrices, defined in Section 3.1, to compute 1:1 simple matches, while Phase 2 uses genetic programming to compute complex n:1 matches, based on the information returned by Phase 1.

3.1 Phase 1: Computing Simple Datatype Property Matches with Estimated Mutual Information

Let $\boldsymbol{p}=(p_1,\ldots,p_u)$ and $\boldsymbol{q}=(q_1,\ldots,q_v)$ be two lists of sets. The *co-occurrence matrix* of $\boldsymbol{p}$ and $\boldsymbol{q}$ is defined as the matrix $[m_{ij}]$ such that $m_{ij} = |p_i \cap q_j|$, for $i \in [1,u]$ and $j \in [1,v]$. The *Estimated Mutual Information matrix (EMI)* of $\boldsymbol{p}$ and $\boldsymbol{q}$ is defined as the matrix $[EMI_{pq}]$ such that:

$$EMI_{pq} = \frac{m_{pq}}{M} \cdot log\left(M \cdot \frac{m_{pq}}{\sum_{j=1}^{v} m_{pj} \cdot \sum_{i=1}^{u} m_{iq}} \right) \tag{1}$$

where $M = \sum_{i=1}^{u} \sum_{j=1}^{v} m_{ij}$.

We now adapt these concepts to define Phase 1 of the datatype property matching process. Let S and T be two classes with sets of datatype properties $\boldsymbol{A}=\{A_1,\ldots,A_u\}$ and $\boldsymbol{B}=\{B_1,\ldots,B_v\}$, respectively. Let $\boldsymbol{s}$ and $\boldsymbol{t}$ be sets of instances of the properties in $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively ($\boldsymbol{s}$ and $\boldsymbol{t}$ therefore are sets of RDF triples).

Rather than simply using the cardinality of set intersections to define the co-occurrence matrix $[m_{ij}]$, Phase 1 computes $[m_{ij}]$ using *set comparison functions* that take two sets and return a non-negative integer. Such functions play the role of *flexibilization points* of Phase 1, as illustrated in Section 4.1.

The set comparison functions depend on the types of the values of the datatype properties as well as on whether the functions take advantage of instance matches. For example, given a pair of datatype properties, A_i and B_j, m_{ij} may be defined as the number of pairs of triples (a, A_i, b) in $\boldsymbol{s}$ and (c, B_j, d) in $\boldsymbol{t}$ such that instances a and c match (or

are identical) and the literals b and d are equal (or are considered equal, under a *literal comparison function* defined for the specific datatype of b and d).

For instance, Leme et al. [19] adopt the cosine similarity function to compare strings. Thus, m_{ij} is computed as the number of (string) values of triples for property A_i in ***s*** whose cosine distance to values of instances for property B_j in ***t*** is above a given threshold ($\alpha = 0.8$ in [19]).

To compute simple matches (1:1), the cosine similarity function proved to be appropriate, especially if the strings to be compared have approximately the same number of tokens. However, the cosine similarity function turned out not to be appropriate when using the co-occurrence matrix to suggest complex matches to Phase 2 of the technique. We therefore adopted the Jaccard similarity coefficient to compute the co-occurrence matrix, defined as

$$Jaccard(b, d) = \frac{|b \cap d|}{|b \cup d|} \tag{2}$$

which counts the number of tokens that strings b and d have in common.

Thus, given two properties A_i and B_j, m_{ij} is computed as the sum of $Jaccard(A_i, B_j)$, for all pairs of strings d and b such that there are triples of the form (a, A_i, b) in ***s*** and (c, B_j, d) in ***t***.

Phase 1 proceeds by computing the EMI matrix based on the co-occurrence matrix, as in Eq. 1. Next, it computes a 1:1 matching, μ_{EMI}, between the properties in ***A***=$\{A_1, \ldots, A_u\}$ and those in ***B***=$\{B_1, \ldots, B_v\}$ such that, for any pair of properties A_p and B_q, $(A_p, B_q) \in \mu_{EMI}$ *iff* $EMI_{pq} > 0$ and $EMI_{pj} \leq 0$, for all $j \in [1, v]$, with $j \neq q$, and $EMI_{iq} \leq 0$, for all $i \in [1, u]$, with $i \neq p$. Furthermore, Phase 1 assumes that the property mappings, $\mu_{EMI}[A_r, B_s]$, are always the identity function.

Finally, Phase 1 also outputs a list of datatype properties to be considered for complex matching in Phase 2. For the kth column of the EMI matrix, it outputs the pair *(**A**^{k},B_k)* as a candidate n:1 complex match, where B_k is the property of T that corresponds to the k_{th} column and ***A***^{k} is the set of properties A_i of S such that $EMI_{ik} > 0$. Indeed, if $EMI_{ik} \leq 0$, then A_i and B_k have no information in common. However, note that this heuristics does not indicate what is a candidate property mapping μ[***A***^{k},B_k]. This problem is faced in Phase 2.

3.2 Phase 2: Computing Complex Property Matches with Genetic Programming

The second phase of the technique uses genetic programming to create mappings between the properties that have some degree of correlation, as identified in the first phase. Briefly, the process goes as follows.

Recall that *genetic programming* refers to an automated method to create and evolve programs to solve a problem [16]. A *program*, also called an *individual* or a *solution*, is represented by a tree, whose nodes are labeled with functions (concatenate, split, sum, etc) or with values (strings, numbers, etc). New individuals are generated by applying genetic operations to the current population of individuals. Note that genetic programming does not enumerate all possible individuals, but it selects individuals that should be bred by an evolutionary process. The *fitness function* assigns a *fitness value* to each individual, which represents how close an individual is to the solution and determines the chance of the individual to remain in the genetic process.

The process requires two configuration steps, carried out just once. First, certain parameters of the process must be properly calibrated to prevent overfitting problems, to avoid unnecessary runtime overhead, and to help finding good solutions (see Section 4). Once the parameters are calibrated, the second configuration step is to determine the stop criterion. We opted to stop after a predetermined maximum number of generations and return the best-so-far individual to limit the cost of searching for individuals.

We now show how to use genetic programming to compute complex datatype property matches. Let S and T be two classes with sets of datatype properties $\boldsymbol{A}=\{A_1,\ldots,A_u\}$ and $\boldsymbol{B}=\{B_1,\ldots,B_v\}$, respectively. Let $\boldsymbol{s}$ and $\boldsymbol{t}$ be lists of sets of instances of the properties in $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively.

The genetic programming phase receives as input the candidate matches that Phase 1 outputs and the sets $\boldsymbol{s}$ and $\boldsymbol{t}$. For each input candidate match, it outputs a property mapping $\mu[\boldsymbol{A^k},B_k]$, if one exists; otherwise it discards the candidate match.

Let $(\boldsymbol{A^k},B_k)$ be a candidate match output by the first phase, where $\boldsymbol{A^k}$ is a set of properties in $\boldsymbol{A}$ and B_k is a property in $\boldsymbol{B}$. The genetic programming phase first generates a random initial population of candidate property mappings. In each iteration step, it creates new candidate property mappings using genetic operations. It keeps the best-so-far individual, and returns it when the stop criterion is reached.

The process depends on the following specifications (see [24] for a concrete example), which should be regarded as flexibilization points.

A candidate property mapping $\mu[\boldsymbol{A^k},B_k]$ (the individual in this case) is represented as a tree whose leaves are labeled with the properties in $\boldsymbol{A^k}$ and whose internal nodes are labeled with primitive mapping functions.

The maximum population size, $\sigma_{population}$, is a parameter of the process. The initial population consists of $\sigma_{population}$ randomly generated trees. Each tree has a maximum height, defined by the parameter σ_{height}, each leaf is labeled with a property from $\boldsymbol{A^k}$ and each internal node is labeled with a primitive mapping function.

The reproduction operation simply preserves a percentage of the property mappings from one generation to the next, defined by the parameter $\sigma_{reproduction}$.

The crossover operation exchanges subtrees of two candidate property mappings to create new candidate mappings. For example, suppose that $\boldsymbol{A^k}=\{firstName, middleName, lastName\}$ and $B_k=fullName$ and consider the following two candidate property mappings (which use the concatenation operation, "//", and are represented using the notation adopted in Section 2.1):

$$\mu_1[\boldsymbol{A^k}, B_k] = \text{``}fullName \leftarrow (lastName//\textbf{(firstName // middleName)})$$
$$\mu_2[\boldsymbol{A^k}, B_k] = \text{``}fullName \leftarrow (\textbf{(middleName // firstName)}//lastName)$$

The crossover operation might generate the following two new candidate property mappings (by swapping the sub-expressions in boldface):

$$\mu_3[\boldsymbol{A^k}, B_k] = \text{``}fullName \leftarrow (lastName//\textbf{(middleName // firstName)})$$
$$\mu_4[\boldsymbol{A^k}, B_k] = \text{``}fullName \leftarrow (\textbf{(firstName // middleName)}//lastName)$$

The mutation operation randomly alters a node (labeled with a property or with a primitive mapping function) of a candidate property mapping. For example, the node labeled

with "middleName" of $\mu_4[\boldsymbol{A^k}, B_k]$ can be mutated to "firstName", resulting in a new candidate property mapping (which is acceptable, but not quite reasonable, since it repeats firstName):

$$\mu_5[\boldsymbol{A^k}, B_k] = \text{``}fullName \leftarrow ((firstName//firstName)//lastName)$$

Finally, recall that s and t are lists of sets of instances of the properties in $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively. The fitness value of $\mu[\boldsymbol{A^k}, B_k]$ is computed by applying $\mu[\boldsymbol{A^k}, B_k]$ to the instances of the properties in $\boldsymbol{A^k}$ occurring in $\boldsymbol{s}$, creating a new set of instances for B_k, which is then compared with the set of instances of B_k occurring in $\boldsymbol{t}$. As in Section 3.1, the exact nature of the fitness function depends on the types of the values of the datatype properties as well as on whether the function takes advantage of instance matches or not (which is possible when implementing S4). For instance, we used the Levenshtein similarity function for string values and KL-divergence measure [2] for numeric values.

The Levenshtein similarity function is normalized to fall into the interval $[0, 1]$, where 1 indicates that a string is exactly equal to the other and 0 that the two strings have nothing in common, while the KL-divergence measure is used to compute the similarity between two value distributions.

Recall that we are given two samples, $\boldsymbol{p}$ and $\boldsymbol{q}$, of instances of properties of classes P and Q, respectively. Construct the set X of strings that occur as literals of instances of B_k obtained by applying $\mu[\boldsymbol{A^k}, B_k]$ to $\boldsymbol{p}$, and the set Y of strings that occur as literals of instances of B_k in $\boldsymbol{q}$. The fitness score for a candidate property mapping is:

$$Fitness_{string}(\mu[\boldsymbol{A^k}, B_k]) = \frac{1}{n} \sum_{\substack{x \in X \\ y \in Y}} Levenshtein(x, y) \tag{3}$$

where n is the number of pairs in $X \times Y$.

In the case of numeric values, construct the set X of numeric values that occur as literals of instances of B_k, obtained by applying $\mu[\boldsymbol{A^k}, B_k]$ to $\boldsymbol{p}$, and the set Y of numeric values that occur as literals of instances of B_k in $\boldsymbol{q}$. The fitness score for a candidate property mapping is:

$$Fitness_{numeric}(F, G) = \frac{1}{n} \sum_{\substack{x \in X \\ y \in Y}} ln\left(\frac{F(x)}{G(y)}\right) F(x) \tag{4}$$

where n is the number of pairs in $X \times Y$, $F(x)$ represents the target distribution of instances in X and $G(y)$ is the the set of materialized mapping μ in Y from the source distribution of instances.

4 An Example Implementation

With the help of an example, we illustrate how to implement the two-phase technique. We assume that the implementation is in the context of S1 of the process described in Section 2.2, that is, we will not use instance matches. We start with Phase 1, described in Section 3.1.

Table 1. Example schemas

<table>
<tr><th>#</th><th>P</th><th>#</th><th>Q</th></tr>
<tr><td>A_1</td><td>FirstName</td><td rowspan="2">B_1</td><td rowspan="2">FullName
(FirstName // LastName)</td></tr>
<tr><td>A_2</td><td>LastName</td></tr>
<tr><td>A_3</td><td>E-Mail</td><td>B_2</td><td>E-Mail</td></tr>
<tr><td>A_4</td><td>Address</td><td rowspan="4">B_3</td><td rowspan="4">FullAddress
(Address // Number //
Complement // Neighborhood)</td></tr>
<tr><td>A_5</td><td>Number</td></tr>
<tr><td>A_6</td><td>Complement</td></tr>
<tr><td>A_7</td><td>Neighborhood</td></tr>
</table>

The example is based on personal information classes, modeled by class P, with 7 properties and class Q with 3 properties. Table 1 shows the properties from the two classes P and Q, and also indicates which properties or sets of properties match. For example, $\{A_1, A_2\}$ matches B_1.

4.1 Phase 1: Computing Simple Property Matches with Estimated Mutual Information

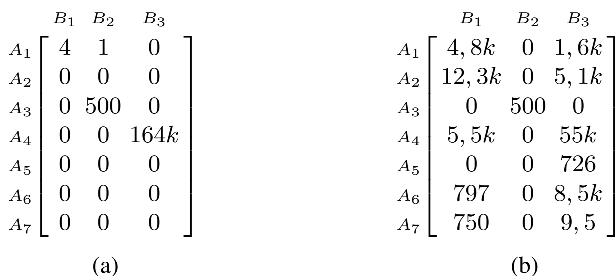
Recall from Section 3.1 that an implementation of Phase 1 requires defining set comparison functions used to compute the co-occurrence matrix $[m_{ij}]$. We discuss this point in what follows, with the help of the running example.

We assume that all property values are string literals and that we are given two samples, ***p*** and ***q***, of instances of properties of classes P and Q, respectively (each with 500 instances). As mentioned in Section 3, Leme et al. [19] use the cosine similarity function to compute the co-occurrence matrix, which is able to indicate only simple 1:1 matches. By contrast, we used the Jaccard similarity coefficient that measures the similarity between sets, which is able to find simple 1:1 matches and *suggest* complex matches.

Figure 1 (a) shows the co-occurrence matrix computed using the cosine similarity measure. Note that $m_{43} = 164k$, which is high because the values of A_4 and B_3 come from a controlled vocabulary with a small number of terms (not indicated in Table 1). By contrast, $m_{32} = 500$, which is low because A_3 and B_2 are keys (also not indicated in Table 1).

Figure 1 (b) shows the co-occurrence matrix computed using the Jaccard similarity (see Eq. 2), which measures the similarity and diversity between sets. Thus, the co-occurrence indices are more sparse between the attributes that have values in common.

To clarify, consider A_7 (Neighborhood) and B_3 (FullAddress) and suppose that "Cambridge" is an observed value of A_7 and "* Oxford Street Cambridge MA, United States" of B_3. The cosine similarity of these two strings is 0.37, which is lower than the threshold set by [19] (again, $\alpha = 0.8$). Hence, these two strings are considered not to be similar. However, also observe that "Cambridge" is fully contained in "* Oxford Street Cambridge MA, United States", which might indicate that A_7, perhaps concatenated with the values of other datatype properties, might match B_3. Continuing with this argument, lowering the threshold also proved not to be efficient to account for these situations, since this increases noise in the matching process.



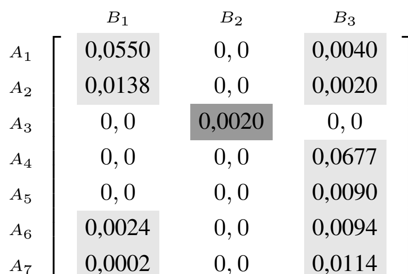
(a)

	B_1	B_2	B_3
A_1	4	1	0
A_2	0	0	0
A_3	0	500	0
A_4	0	0	$164k$
A_5	0	0	0
A_6	0	0	0
A_7	0	0	0

(b)

	B_1	B_2	B_3
A_1	$4,8k$	0	$1,6k$
A_2	$12,3k$	0	$5,1k$
A_3	0	500	0
A_4	$5,5k$	0	$55k$
A_5	0	0	726
A_6	797	0	$8,5k$
A_7	750	0	$9,5$

Fig. 1. Co-occurrence matrices using (a) cosine similarity and (b) Jaccard similarity coefficient



	B_1	B_2	B_3
A_1	0,0550	$0,0$	0,0040
A_2	0,0138	$0,0$	0,0020
A_3	$0,0$	0,0020	$0,0$
A_4	$0,0$	$0,0$	0,0677
A_5	$0,0$	$0,0$	0,0090
A_6	0,0024	$0,0$	0,0094
A_7	0,0002	$0,0$	0,0114

Fig. 2. EMI matrix: dark gray cells represent simple matches and light gray cells represent possible complex matches for the property in the column

Thus, given two properties A_i and B_j, m_{ij} is computed as the sum of $Jaccard(x, y)$, for all pairs of strings x and y such that there are triples of the form (a, A_i, x) in $\boldsymbol{p}$ and (b, B_j, y) in $\boldsymbol{q}$ (see Figure 1). Once the co-occurrence matrix $[m_{ij}]$ is obtained, we compute the EMI matrix $[EMI_{ij}]$, as described in Section 3.1 (see Figure 2).

The result of Phase 1 therefore is the matching μ_{EMI} between the sets of properties $\{A_1, \ldots, A_u\}$ and $\{B_1, \ldots, B_v\}$, computed as in Section 3.1 (which we recall is $1:1$), assuming that, for each $(A_i, B_j) \in \mu_{EMI}$, the property mappings $\mu[A_i, B_j]$ is always the identity function (see Figure 2).

4.2 Phase 2: Computing Complex Property Matches with Genetic Programming

The second phase of the technique was implemented using a genetic programming toolkit [21], (the discussion on calibration is omitted for brevity, see [24] for more details).

The first phase of the technique outputs, for instance, a candidate match between properties A_1, A_2, A_4, A_5, A_6 and A_7 (FirstName, LastName, Address, Number, Complement and Neighborhood, respectively) and property B_3 (FullAddress), see Figure 2. Note that quite frequently streets are named after famous people, which justifies why EMI outputs A_1 and A_2 as candidates properties. Following the example, having 6 properties as input, the genetic process begins the search for the solution.

As the property values are strings, the fitness function selected to find the best individual is the Levenshtein (see Eq. 3). Thus, after randomly generate an initial set of

individuals, the fitness function assigns to each individual a score. For each new generation, a new set of individuals is created from those individuals chosen according to a probability based on their fitness value. After a predetermined number of generations, the process stops with an expression that represents a property mapping that maps the concatenation of the properties A_4, A_5, A_6 and A_7, that is, the expression:

$$((Address//Number)//(Complement//Neighborhood))$$

into property B_3 (that is, FullAddress).

5 Evaluation and Results

The first result in this paper is the comparison of the two approaches, Estimated Mutual Information and Genetic Programming, when separately evaluated.

For this evaluation, we used three datasets[1] from three different domains. Table 2 lists and describes the datasets used and their schema information. The "Personal Information" dataset lists information about people, the "Real Estate" dataset lists information about houses for sale, while the "Inventory" dataset describes product inventories.

Column "EMI" of Table 3 indicates that, using only the Estimated Mutual Information approach, we obtained a precision of 1.0 for all datasets, which indicates that none of the matches were mistakenly found; the rate of recall was low, between 0.21 and 0.38, indicating a high rate of missed property matches; and the F-Measure varied from 0.34 to 0.54, hinting that this approach is insufficient to find simple and complex matches. Indeed, out of the 12 simple matches expected for the "Personal Information" dataset, this approach correctly obtained 6 matches only. Likewise, the EMI found 3 out of 4 and 4 out of 6, for the datasets "Inventory" and "Real Estate", respectively.

However, according to the discussion at the end of Section 3.1, as well as by observing the column "EMI" marked with "*" in Table 2, there are several candidate complex matches that were suggested to the GP phase in each approach. Note that amongst those are the exact remaining matches not found by the EMI technique. This is an indication that, although not sufficient in itself, the EMI approach is an effective pre-processing stage to the GP approach, by reducing the complexity of the search space while providing a high quality list of candidate complex matches.

Column GP of Table 3 indicates that, using genetic programming alone, the F-Measure obtained was higher, and that all simple mappings were found. However, precision was 0.8 for the "Personal Information" dataset and 0.96 for the "Inventory" dataset, which indicates that some matches were mistakenly suggested.

Table 3 shows that our two-phase technique resulted in a considerable improvement over the independent use of the EMI and GP approaches when used independently. This improvement is related to the fact that the first phase, using the EMI matrix, correctly found all simple matches and suggested correct complex matches to the second phase.

[1] With exception of the "Personal Information" dataset due to privacy reasons, other datasets are available at `http://pages.cs.wisc.edu/~anhai/wisc-si-archive/domains/`

Table 2. Mapping results for three datasets in different domains

Datasets	Type		EMI	GP	**EMI+GP**	#Match
Personal Information	String	1:1	6	12	**12**	12
		1:n	11*	1	**4**	5
	Numeric	1:1	0	0	**0**	0
		1:n	0	0	**0**	0
Inventory	String	1:1	3	4	**4**	4
		1:n	18*	2	**4**	4
	Numeric	1:1	6	25	**25**	25
		1:n	18*	1	**3**	4
Real Estate	String	1:1	4	4	**6**	6
		1:n	7*	2	**5**	5
	Numeric	1:1	1	1	**1**	1
		1:n	7*	0	**0**	3

(∗) Complex matches suggested by EMI.

Table 3. P/R/F1 results for three datasets in different domains

	EMI			GP			EMI+GP		
Dataset	P	R	F1	P	R	F1	P	R	F1
Personal Information	1	0.38	0.54	0.8	0.75	0.77	**1**	**0.94**	**0.96**
Inventory	1	0.24	0.39	0.96	0.87	0.91	**0.97**	**0.97**	**0.97**
Real Estate	1	0.33	0.5	1	0.47	0.64	**1**	**0.8**	**0.89**

The fact that the EMI matrix suggests correlated properties helps reduce the solution space considered by the genetic programming algorithm, thus improving its overall performance. In our tests, the run time of the combined approach showed an improvement of approximately 36% when compared with the run time of the genetic programming approach alone.

Furthermore, we also compared our method against state of the art methods. As a baseline we used the iMap system [5], which similar to our approach addresses the problem of 1:1 and n:1 (complex) matchings. From previously reported results in terms of accuracy, iMap obtains 0.84 and 0.55 for 1:1 and 1:n mappings respectively, while we obtain 1 and 0.955 for the “Inventory” dataset. For the “Real Estate” dataset, iMap achieves 0.58 and 0.32, whereas we achieve 1 and 0.72, respectively. We also compared our method against LSD [7], which is able to find only simple 1:1 matchings and achieves an accuracy of 0.67.

6 Related Work

Ontology alignment frameworks implement a set of similarity measures to find the correct mappings. For instance, Duan et al. [9] utilize user feedback to determine the importance of each similarity measure in the final mapping result. Similarly, Ritze et al. [27] introduce ECOMatch that uses alignment examples to define parameters to set the correct mapping strategy. Dhamankar et al. [6] describe iMap that predefines modules of functions to semi-automatically find simple and complex matches by

leveraging external knowledge. Likewise, Albagli et al. [1] search for mappings using Markov Networks, which combines different sources of evidence (e.g. human experts, existing mappings, etc). Finally, Spohr et al. [30] use a translation mechanism to discover mappings in cross-lingual ontologies.

A drawback in most approaches is scalability. Duan et al. [8] address the scalability problem using a local sensitivity hashing to match instances inside a cluster. Jiménez-Ruiz and Grau [15] propose an “on the fly” iterative method called LogMap that, based on a set of anchors (exact mappings), creates, extends and verifies mappings using a logical reasoner. Complementary, Wang et al. [31] suggest a method for reducing the number of anchors needed to match ontologies. Recent advances, such as RiMOM [20], offer an automated environment to select an appropriate matching strategy through risk minimization of Bayesian decision, while ASMOV ([14]) uses semantic validation to verify mappings. Falcon [13] applies a divide-and-conquer approach to ontology matching. Several other systems, such as DSSim [22], S-Match [11], Anchor-Flood [12], Agreement-Maker [3], ATOM [26] and SAMBO [17] tackle the alignment for ontologies and schemas relying on lexical, structural and semantical similarity measures. In a recent survey, [29] analyze in more details well-established frameworks and outline future directions and challenges in this field. Additional surveys are provided by [28,25].

Contrasting with the approaches just outlined, we provide an automatic technique that finds simple and complex mappings between RDF datatype properties without prior knowledge that can evolve to adapt to schema and ontology changes, previously described in [23]. Similar to our approach, Carvalho et al.[4] propose a genetic programming approach for deduplication problem. However, as the results show, our two-phase approach achieves better results than those using only the genetic programming approach. Moreover, we extend his work to match simple and complex numeric datatype properties.

7 Conclusion

In this paper, we described an instance-based, property matching technique that follows a two-phase strategy. The first phase constructs the Estimated Mutual Information matrix of the property values to identify simple property matches and to suggest complex matches, while the second phase uses a genetic programming approach to detect complex property matches and to generate their property mappings. This combined strategy proved promising to beat combinatorial explosion. In fact, our experiments prove that the technique is a promising approach to construct complex property matches, a problem rarely addressed in the literature.

Acknowledgement. This work has been partially supported by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement No 317620 (LinkedUp).

References

1. Albagli, S., Ben-Eliyahu-Zohary, R., Shimony, S.E.: Markov network based ontology matching. Journal of Computer and System Sciences 78(1), 105–118 (2012)

2. Cover, T.M., Thomas, J.A.: Elements of information theory. Wiley, New York (1991)
3. Cruz, I.F., Antonelli, F.P., Stroe, C.: Agreementmaker: Efficient matching for large real-world schemas and ontologies. PVLDB 2(2), 1586–1589 (2009)
4. de Carvalho, M.G., Laender, A.H.F., Gonçalves, M.A., da Silva, A.S.: A genetic programming approach to record deduplication. IEEE Trans. Knowl. Data Eng. 24(3), 399–412 (2012)
5. Dhamankar, R., Lee, Y., Doan, A., Halevy, A., Domingos, P.: imap: discovering complex semantic matches between database schemas. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD 2004, pp. 383–394. ACM, New York (2004)
6. Dhamankar, R., Lee, Y., Doan, A., Halevy, A.Y., Domingos, P.: imap: Discovering complex mappings between database schemas. In: SIGMOD Conference, pp. 383–394 (2004)
7. Doan, A., Domingos, P., Levy, A.: Learning Source Descriptions for Data Integration. In: Proceedings of the Third International Workshop on the Web and Databases, Dallas, TX, pp. 81–86. ACM SIGMOD (2000)
8. Duan, S., Fokoue, A., Hassanzadeh, O., Kementsietsidis, A., Srinivas, K., Ward, M.J.: Instance-based matching of large ontologies using locality-sensitive hashing. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 49–64. Springer, Heidelberg (2012)
9. Duan, S., Fokoue, A., Srinivas, K.: One size does not fit all: Customizing ontology alignment using user feedback. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 177–192. Springer, Heidelberg (2010)
10. Euzenat, J., Shvaiko, P.: Ontology matching. Springer (2007)
11. Giunchiglia, F., Autayeu, A., Pane, J.: S-match: An open source framework for matching lightweight ontologies. Semantic Web Journal 3(3), 307–317 (2012)
12. Hanif, M.S., Aono, M.: An efficient and scalable algorithm for segmented alignment of ontologies of arbitrary size. Journal of Web Semantics 7(4), 344–356 (2009)
13. Hu, W., Qu, Y., Cheng, G.: Matching large ontologies: A divide-and-conquer approach. IEEE Trans. Knowl. Data Eng. 67(1), 140–160 (2008)
14. Jean-Mary, Y.R., Shironoshita, E.P., Kabuka, M.R.: Ontology matching with semantic verification. Journal of Web Semantics 7(3), 235–251 (2009)
15. Jiménez-Ruiz, E., Cuenca Grau, B.: LogMap: Logic-based and scalable ontology matching. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 273–288. Springer, Heidelberg (2011)
16. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge (1992)
17. Lambrix, P., Tan, H.: Sambo - a system for aligning and merging biomedical ontologies. Journal of Web Semantics 4(3), 196–206 (2006)
18. Leme, L.A.P.P., Brauner, D.F., Breitman, K.K., Casanova, M.A., Gazola, A.: Matching object catalogues. ISSE 4(4), 315–328 (2008)
19. Leme, L.A.P.P., Casanova, M.A., Breitman, K.K., Furtado, A.L.: Instance-based OWL schema matching. In: Filipe, J., Cordeiro, J. (eds.) ICEIS 2009. LNBIP, vol. 24, pp. 14–26. Springer, Heidelberg (2009)
20. Li, J., Tang, J., Li, Y., Luo, Q.: Rimom: A dynamic multistrategy ontology alignment framework. IEEE Transactions on Knowledge and Data Engineering 21(8), 1218–1232 (2009)
21. Meffert, K.: Jgap - java genetic algorithms and genetic programming package (2013), `http://jgap.sf.net/` (Online; accessed January 31, 2013)
22. Nagy, M., Vargas-Vera, M., Stolarski, P.: Dssim results for oaei 2009. In: Ontology Matching (2009)

23. Nunes, B.P., Caraballo, A.A.M., Casanova, M.A., Breitman, K., Leme, L.A.P.P.: Complex matching of rdf datatype properties. In: Ontology Matching (2011)
24. Nunes, B.P., Mera, A., Casanova, M.A., Breitman, K., Leme, L.A.P.P.: Complex matching of rdf datatype properties. Technical Report MCC-11/12 (September 2011)
25. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB Journal 10(4), 334–350 (2001)
26. Raunich, S., Rahm, E.: Atom: Automatic target-driven ontology merging. In: ICDE Conference, pp. 1276–1279 (2011)
27. Ritze, D., Paulheim, H.: Towards an automatic parameterization of ontology matching tools based on example mappings. In: Ontology Matching (2011)
28. Shvaiko, P., Euzenat, J.: A survey of schema-based matching approaches, pp. 146–171 (2005)
29. Shvaiko, P., Euzenat, J.: Ontology matching: State of the art and future challenges. IEEE Trans. Knowl. Data Eng. 25(1), 158–176 (2013)
30. Spohr, D., Hollink, L., Cimiano, P.: A machine learning approach to multilingual and cross-lingual ontology matching. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 665–680. Springer, Heidelberg (2011)
31. Wang, P., Zhou, Y., Xu, B.: Matching large ontologies based on reduction anchors. In: IJCAI, pp. 2343–2348 (2011)