lirmm-00399025, version 1 - 25 Jun 2009

Yet Another Matcher*

ABSTRACT

Discovering correspondences between schema elements is a crucial task for data integration. Most matching tools are semi-automatic, e.g. an expert must tune some parameters (thresholds, weights, etc.). They mainly use several methods to combine and aggregate similarity measures. However, their quality results often decrease when one requires to integrate a new similarity measure or when matching particular domain schemas. This paper describes YAM (Yet Another Matcher), which is a matcher factory. Indeed, it enables the generation of a dedicated matcher for a given schema matching scenario, according to user inputs. Our approach is based on machine learning since schema matchers can be seen as classifiers. Several bunches of experiments run against matchers generated by YAM and traditional matching tools show how our approach (i) is able to generate the best matcher for a given scenario and (ii) easily integrates user preferences, namely recall and precision tradeoff.

Keywords: schema matching, data integration, matcher factory, machine learning, XML schemas.

1. INTRODUCTION

Discovering correspondences between schema elements is a crucial task for many data integration and data sharing tasks. There are a plethora of schema matching tools designed to help automate what can be a painstaking task if done manually. The diversity of tools hints at the inherent complexity of this problem. Different tools are designed to overcome different types of schema heterogeneity including differences in design methodologies, differences in naming conventions, and differences in the level of specificity of schemas, among many other types of heterogeneity. Furthermore, different matchers may be designed to help with very different integration tasks. Some are designed to help in automatically matching web service interfaces for the purpose of wrapping and composing web services. Others are designed for matching large, complex legacy schema to facilitate federated querying. The proliferation of schema matchers and the proliferation of new (often domain-specific) similarity measures used within these matchers has left data integration practitioners with the very perplexing task of trying to decide which matcher to use for the schemas and tasks they need to solve.

*Supported by ANR DataRing ANR-08-VERSO-007-04

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

Most matching tools are semi-automatic meaning that to perform well, an expert must tune some (matcher-specific) parameters (thresholds, weights, etc.) Often this tuning can be a difficult task as the meaning of these parameters and their effect on matching quality can only be seen through trial-and-error. Lee et al. [14] have shown how important (and difficult) tuning is, and that without tuning most matchers perform poorly. To overcome this, they proposed *eTuner*, a supervised learning approach for tuning these matching knobs. However, a user must still commit to one single matcher and then tune that matcher to their specific domain (i.e., set of training schemas with their correct correspondences). If the user makes a poor choice of matcher to begin with, for example, by choosing a matcher that does not consider structural schema similarity when this is important in the domain, *eTuner* cannot help. Furthermore, knowing beforehand whether semantic similarity or structural similarity or syntactic similarity (or some combination of these) will be most important in a domain is not an easy task.

In this work, we propose YAM, which is actually not Yet Another Matcher. Rather YAM is the first schema matcher generator designed to produce a tailor-made matcher based on user requirements. In YAM, we also use a supervised learning approach. The novelty of YAM is that unlike eTuner or any other schema matcher, YAM performs learning over a large set of matchers and a large set of similarity measures. Over this large search space, using a small amount of training data, YAM is able to produce a "dedicated" or tailor-made matcher. The matchers YAM considers are classifiers (the basis of most matchers). YAM uses a large library of classifiers and similarity measures, and is extensible in both dimensions. In particular, new similarity measures custom-made for new domains or new integration tasks, can easily be added to YAM.

Schema matchers (often implicitly) are designed with one or a few matching tasks in mind. A matcher designed for automated web service composition may use very stringent criteria in determining a match, that is, it may only produce a correspondence if it is close to 100% confident of the correspondence's accuracy. In other words, such a matcher is using precision as its performance measure. In contrast, a matcher designed for federating large legacy schema may produce all correspondences that look likely, even if they are not certain. Such a matcher may favor recall, over precision, because the human effort in "rejecting" a bad correspondence is much less than the effort needed to search through large schemas and find a missing correspondence. This difference can make a tremendous difference in the usefulness of a matcher for a given

lirmm-00399025, version 1 - 25 Jun 2009

task. In YAM, we let a user specify a preference for precision or recall, and we produce a dedicated matcher that best meets the users needs. YAM is the first tool that permits the tuning of this very important performance trade-off.

Like previous supervised-learning approaches to schema matching [5, 15, 9], YAM requires a knowledge-base containing training data in the form of pairs of schemas with their (correct) correspondences. Unlike most matchers, YAM can also take as input a subset of correct correspondences over the new schemas a user wishes to match. Such "expert correspondences" are often available. For example, in matching legacy schemas, the correct correspondence for some keys of central tables may be known to a user. YAM does not require such knowledge, but can use it, if available, to produce a better dedicated matcher.

Contributions. The main interesting features of our approach are:

- YAM is the first matcher factory capable of generating a dedicated matcher for a given scenario and according to user inputs. In the experiments, we show YAM has generated, for 200 schema matching scenarios, different dedicated matchers (i.e., they are based on different algorithms such as decision trees, rules, aggregation functions, etc.).
- Our approach integrates a user preference between precision or recall during the generation process. We demonstrate the impact of such preference on the matching quality.
- YAM is also able to use correct correspondences when provided by an user. Indeed, most matchers do not need to focus on these provided correspondences during generation of the dedicated matcher. Thus, we observe a strong increase of the matching quality.
- We finally evaluate our work with traditional matching tools. YAM achieves comparable results in terms of matching quality.

Outline. The rest of the paper is organised as follows. Section 2 describes the main notions used in the paper. Section 3 gives an overview of our approach while Section 4 contains the details. The results of experiments for showing the effectiveness of our approach are presented in section 5. Related work is described in section 6. Finally, we conclude in section 7.

2. PRELIMINARIES

In this section, we define the main notions used in this paper. Then, we describe our schema corpus and quality measures used for evaluation.

2.1 Definitions

YAM can be applied to match pairs of edge-labeled trees (a simple abstraction that can be used for XML schemas, web interfaces, Jason data types, or other semi-structured or structured data models). Schema matching task can be divided into three steps. The first one is named pre-match and is optional. Either the tool or the user can intervene, for instance to provide resources (dictionaries, expert correspondences, etc.), to set up parameters (tuning of weights, thresholds, etc.), etc. Secondly, the matching process occurs, during which correspondences are discovered. The final step, the post-match process, mainly consists in a validation of the discovered correspondences by the user.

Definition 1 (Schema): A schema is a labeled unordered tree $S = (E_S, D_S, r_S, label)$ where E_S is a set of elements; r_S is the root element; $D_S \subseteq E_S \times E_S$ is a set of edges; and *label* $E_S \rightarrow \Lambda$ where Λ is a countable set of labels.
Definition 2 (Pair): A pair of schema elements is defined as a tuple $< e_1, e_2 >$ where $e_1 \in E_1$ and $e_2 \in E_2$ are schema elements.
Definition 3 (Similarity Measure): Let E_1 be a set of elements of schema 1, and E_2 be a set of elements of schema 2. A similarity measure between a pair, denoted as $Sim(e_1, e_2)$, is a metric value, the similarity value, which indicates the likeness between both elements. It is defined by:

$$Sim : e_1 \text{x} e_2 \rightarrow [0, 1]$$

where a zero value means total dissimilarity and 1 value stands for total similarity.
Definition 4 (Correspondence): A correspondence is a pair of schema elements which have a semantic similarity. Correspondences can be either discovered by similarity measures or given by an expert. They are defined as follows, $< e_1, e_2, Sim(e_1, e_2) >$. An expert correspondence has a similarity value equal to 1.
Definition 5 (Schema matching scenario): A schema matching scenario is a set of schemas (typically from the same domain) that need to be matched.
Definition 6 (Schema matcher): A schema matcher is an algorithm which combines similarity measures. Given a schema matching scenario, it produces a set of correspondences. In YAM, we consider schema matchers that are classifiers.

2.2 Running Example

Let us imagine that we would like to match two hotel booking webforms. These webforms are depicted by figures 1(a) and 1(b). The expert correspondences between their schemas are shown in figure 2.

Figure 2: Expert correspondences between the two *hotel booking* webforms

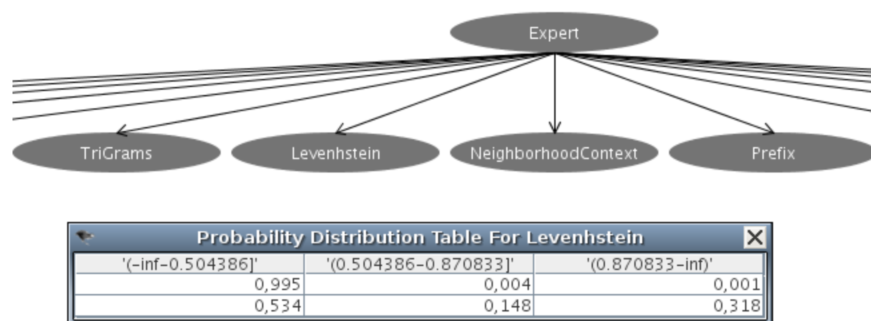
Figure 3 depicts an extract of the dedicated matcher for this *hotel booking* scenario. This schema matcher is based on the *Bayes Net* classifier. Each similarity measure is associated with a probabilistic distribution table. For instance, if the *Levenshtein measure* returns a similarity value between $-\infty$ and 0.504386, there is a 99.5% chance that the current pair of schema elements is not relevant. Using this dedicated matcher against the *hotel booking* scenario, we were able to discover the set of correspondences shown in figure 4. We notice that 8 out of the 9 relevant correspondences have been discovered. Two irrelevant correspondences have also been found, namely (*Hotel Location*, *Hotel Name*) and (*Children:*, *Chain*).

lirmm-00399025, version 1 - 25 Jun 2009

(a) Webform hotels valued

(b) Webform where to stay

Figure 1: Two *hotel booking* webforms



'(-inf-0.504386]'	'(0.504386-0.870833]'	'(0.870833-inf)'
0,995	0,004	0,001
0,534	0,148	0,318

Figure 3: Extract of the dedicated matcher

2.3 Schema Corpus

To demonstrate the effectiveness of our approach, we used several schema matching scenarios:

- **university** describes university departments and it has been widely used in the literature [10, 7].
- **thalia** [13] is a benchmark describing the courses offered by some American universities.
- **travel** are schemas extracted from airfare web forms [1].
- **currency** and **sms** are popular web services which can be found at *http://www.seekda.com*.
- **web forms** is a set of 176 schema matching scenarios, extracted from various websites by the authors of [15]. They are related to different domains, from *hotel booking* and *car renting* to *dating* and *betting*.

For all these scenarios, correct (relevant) correspondences are available, either designed manually or semi-automatically. We use these schemas, and their correct correspondences as the training data for YAM.

2.4 Quality measures

To evaluate the matching quality, we use common measures in the literature, namely precision, recall and f-measure. Precision calculates the proportion of relevant correspondences extracted among the discovered ones. Another typical measure is recall which computes the proportion of relevant discovered correspondences among all relevant ones. F-measure is a tradeoff between precision and recall. We also measured the number of user interactions required to obtain a 100% f-measure given a set of discovered correspondences. This measure is described in section 5.5.

3. OVERVIEW OF YAM

YAM (Yet Another Matcher) is a matcher factory tool, which generates a dedicated schema matcher according to user preferences and some expert correspondences.

3.1 Motivations

The motivation for our work is the following:

- There is no schema matching tool which performs best for all matching scenarios. Although matching tools enable the user to tune some parameters (strategies, weights, thresholds,

Figure 4: Correspondences discovered using the dedicated matcher

etc.), the same algorithm, for instance, COMA++'s aggregation function [2], is always used independently of the schema matching scenario. eTuner [14] automatically tunes schema matching tools by tuning the input parameters used by the matcher. Thus, it mainly depends on their capabilities since it finds the best parameters configuration. Specifically, eTuner fined the best parameter settings for a given matching algorithm. On the contrary, YAM is able to produce the dedicated schema matcher for a given scenario. Each generated schema matcher has its own features, among which a different algorithm (aggregation functions, Bayes network, decision trees, etc.). This is the **adaptable** feature of our approach.

- User intervention is always required, at least to check the discovered correspondences. In other systems, users are also asked to edit synonyms list, reduce schema size [8], or tune various parameters. In the same spirit, YAM uses some user inputs, but of a different form. Specifically, YAM can optionally use a preference between precision and recall, and some expert correspondences (that is, a small number of correct matches). This small amount of input enables the use of supervised learning to create a dedicated schema matcher. YAM is able to convert user time spent to give preferences into better quality results. Indeed, most schema matching tools focus on a better precision, but this does not seem to be the best choice in terms of post-matching effort, i.e., the quantity of work required by an expert to correct discovered correspondences. Technically speaking, it is easier for the expert to validate (or not) a discovered mapping than to manually browse two large schemas for new correspondences that the tool may have missed.

- Some matching tools are said to be extensible, for example to add new similarity measures. However, this extensibility is constrained by some parameters, which need to be manually updated (for instance, adjusting thresholds, re-weighting values, etc.). Thanks to machine learning techniques and according to user inputs, YAM automatically learns how to combine similarity measures into a classifier to produce the dedicated schema matcher. Thus, our approach is **extensible** in terms of similarity measures, but also in terms of classifiers.

Now let us discover what is inside our factory of schema matchers.

3.2 YAM Architecture

To the best of our knowledge, our approach is the first to propose a factory of schema matchers. The intuition which led to our work is as follows: the algorithms which combine similarity measures provide different results according to a given schema matching scenario. Thus, YAM aims at generating for a schema matching scenario a dedicated schema matcher. For this purpose, YAM uses *machine learning* techniques during pre-match phase.

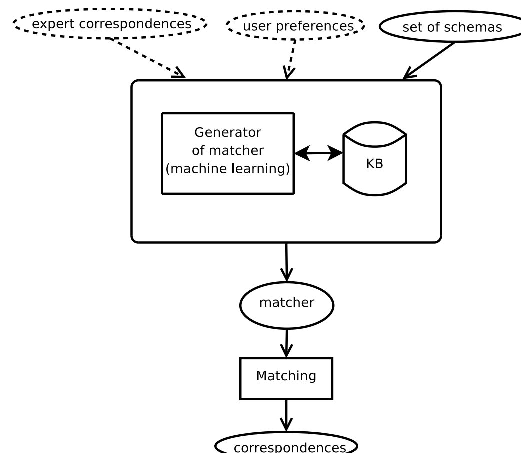


Figure 5: YAM architecture

Figure 5 depicts the YAM architecture. Circles represent inputs or outputs and the rectangles stand for processes. Note that a dashed circle means that this input is optional. YAM only requires one input, the set of schemas to be matched. However, the user can also provide additional inputs, i.e., some preferences and/or expert correspondences. The preference consists of a precision and recall tradeoff. Expert correspondences (from a domain of interest, or for the schemas to be matched) are only used by the matcher generator to produce a better dedicated matcher. YAM is composed of two main components: the matcher generator is in charge of learning a dedicated matcher for the input schemas (see section 4 for more details). This component interacts with the Knowledge Base (KB). This KB stores previously generated matchers, a set of classifiers, a set of similarity measures, and expert correspondences which have already been given or validated. Finally, YAM outputs a schema matcher, either generated or selected and which is then stored in the KB. It can obviously be used for matching the input schemas, thus producing a list of discovered correspondences between the schemas. Note that the matching process is specific to the classifier that will be used and is not detailed in this paper.

The current version of YAM includes 20 classifiers from the Weka library [12] and 30 similarity measures, including all the terminological measures from the Second String project[1], and some structural and semantic measures. YAM's knowledge base contains a large set of 200 schemas from various domains (betting, hotel booking, dating, etc.) gathered from the web.

4. LEARNING A DEDICATED MATCHER

In this section, we describe YAM's approach for learning a matcher (which we call a dedicated matcher) for a given matching scenario. Any schema matcher can be viewed as a classifier. Given the set of possible correspondences (the set of pairs of elements in the schemas), a matcher labels each pair as either *relevant* or *irrelevant*. Of course, a matcher may use any algorithm to compute

[1] http://secondstring.sourceforge.net

lirmm-00399025, version 1 - 25 Jun 2009

lirmm-00399025, version 1 - 25 Jun 2009

its result – classification, clustering, an aggregation over similarity measures, or any number of ad hoc methods including techniques like blocking to improve its efficiency.

In YAM, we use an extensible library of classifiers (in our experiments including the 20 classifiers from the Weka library [12]) and train them using an extensible library of similarity measures. (in our experiments including all the measures from the popular Second String project). The classifiers include decision trees (*J48*, *NBTree*, etc.), aggregator functions (*SimpleLogistic*), lazy classifiers (*IBk*, *K**, etc.), rules-based (*NNge*, *JRip*, etc.) and Bayes Networks.

The generation of a dedicated matcher can be split into two steps: (i) training of matchers, and (ii) final matcher selection.

4.1 Matcher Training

YAM trains each matcher using its knowledge base of training data and expert correspondences (if available). We begin our explanation with an example.

Example: Let us consider the pair *(searchform, search)* from our running example. We computed the similarity values of this pair with each similarity measure in our library. For our example, let us assume we have three similarity measures: *AffineGap* = 14.0, *NeedlemanWunsch* = -4.0, *JaroWinkler* = 0.92. From these values, a matcher must predict if the pair is relevant or not.

To classify an element pair as relevant or not, a classifier must be trained. YAM offers two training possibilities: either the user has provided some *expert* correspondences, with their relevance, or YAM uses correspondences stored in a knowledge-base (KB). Note that if the user has not provided a sufficient number of correspondences, YAM will extract some more from the KB. When the user has not provided any expert correspondences, the matcher is learned from the KB, i.e., YAM will use a matcher that provides the best average results on the KB.

During training, all the thresholds, weights, and other parameters of the matcher are automatically set. Although each matcher performs differently, we briefly sum up how they work. First, they select the similarity measures which provides a maximum of correctly classified correspondences. Then, the similarity measures that might solve harder cases are taken into account.

Example: If the user has provided the following expert correspondences *(* City:, City)* and *(State:, State)*, string matching measures like *JaroWinkler* or *Levenshtein* will be first considered by the machine learning algorithms. Indeed, they enable a correct classification of both pairs.

In general, these algorithms aim at reducing the misclassification rate. Two errors can occur while training: classifying an irrelevant correspondence as relevant (a.k.a. a false positive or extra incorrect correspondence) and classifying a relevant correspondence as irrelevant (a.k.a. a false negative or a missed correct correspondence). The first error decreases precision while the second one decreases recall. Many algorithms assign the same penalty to a false positive (i.e., an irrelevant pair that has been discovered) and to a false negative (i.e. a relevant pair that has been missed). To increase recall on a given dataset, we assign a greater penalty to false positives. Thus, we should obtain a better recall for a given dataset. Note that promoting recall (respectively precision) mainly decreases precision (respectively recall). Our approach is able to generate matchers which respect a user specified preference for recall or precision.

At the end of this step, YAM has generated a trained matcher for each classifier in the KB.

4.2 Selecting a Dedicated Matcher

A dedicated matcher is selected according to its accuracy on the given training data (KB and possible expert correspondences). If a user has not provided any expert correspondences, all matchers have been trained on the KB. Thus, YAM cross-validates each matcher with the correspondences stored in the KB. On the contrary, if user has provided correspondences, cross validation is performed against the given correspondences. Finally, the matcher which discovered most of the correspondences and the fewest irrelevant ones is selected as the dedicated schema matcher. Note that these computations correspond to the f-measure (perhaps weighted by a user preference for recall or precision).

Example: Let us imagine that we have generated 3 matchers, named *NNge*, *J48* and *SMO*. They respectively achieve the following f-measure during cross-validation: 0.56, 0.72 and 0.30. Thus, *J48* would be chosen as the dedicated matcher.

5. EXPERIMENTS

In these experiments, we first demonstrate that YAM is able to produce an effective dedicated matcher. Thus, we evaluate the results of generated matchers against 200 scenarios. Then, we measure the quality impact according to the number of training scenarios. Our goal is to show that the amount of training data needed to produce a high performing matcher is not onerous. Next, we study the impact of a user preference between recall and precision. Then we consider the performance of YAM with respect to the number of expert correspondences. Finally, we compare our results with two matching tools that have excellent matching quality, COMA++ [2] and Similarity Flooding [16]. These tools are described in more detail in section 6.

The schema corpus we use was presented in section 2.3. Experiments were run on a 3.6 Ghz computer with 4 Go RAM under Ubuntu 7.10. Our approach is implemented in Java 1.6. In training, we used 200 runs to minimize the impact of randomness.

5.1 Comparing generated matchers

We begin with a study of which matchers were selected as for different matching scenarios. This study highlights how different the performance of each match can be on different scenarios, and therefore the importance of matching factory tool such as YAM for selecting among these matchers to produce an effective matcher.

We have run YAM against 200 scenarios, and we measured the number of times a given matcher is selected as the dedicated matcher. For this evaluation, we included no expert correspondences, so all matchers were trained simply with the KB. The KB contained 20 scenarios, and this process took roughly 1400 seconds to produce a dedicated matcher for each given scenario. Figure 6 depicts the number of scenarios (out of 200) for which each matcher was selected as the dedicated matcher. Notice that 2 matchers, *VFI* and *BayesNet*, are selected in half of the scenarios. These two matchers can be considered as robust as they provide acceptable results in most scenarios in our KB. However, matchers like *CR* or *ADT*, which have a very low average f-measure on these 200 scenarios (5% for *CR* and 28% for *ADT*), were respectively selected 3 and 10 times. This shows that dedicated matchers based on these classifiers are effective, in terms of quality, for specific scenarios. Thus, they can provide benefits to some users. These results support our hypothesis that schema matching tools have to be flexible. YAM, by producing different matchers and selecting the best one for a given scenario, fulfills this requirement.

We also note that aggregation functions, like *SLog* or *MLP*, which are commonly used by traditional matching tools, are only selected as dedicated matchers in a few scenarios. Thus, they do not provide optimal quality results in most schema matching scenarios. In the next section, showing the impact of the parameters, we only keep the 5 most robust classifiers, namely *VFI*, *BayesNet*, *NBTree*, *NNge* and *IB1*.

lirmm-00399025, version 1 - 25 Jun 2009

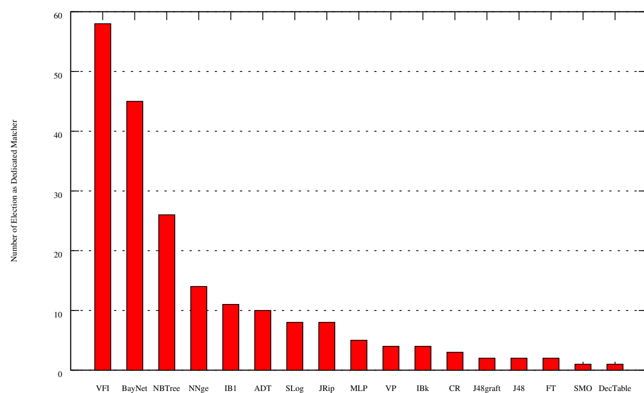


Figure 6: Number of selections as dedicated matcher

5.2 Impact of the Training Scenarios

Figure 7 depicts the average f-measure of several matchers as we vary the number of training scenarios. Note that the average f-measure has been computed over 40 scenarios (randomly selected, 20 runs each). The training scenarios vary from 10 to 50. We note that two matchers (*VFI*, *IB1*) increase their f-measure of 20% when they are generated with more training scenarios. This can be explained by the fact that *IB1* is an instance-based classifier[2], thus the more examples it has, the more accurate it becomes. Similarly, *VFI* uses a voting system on intervals that it builds. Voting is also appropriate when lots of training examples are supplied. *NBTree* and *NNge* also increases their average f-measure from around 10% as training data is increased. On the contrary, *BayesNet* achieves the same f-measure (60% to 65%) regardless of the number of training scenarios. Thus, as expected, most matchers increase their f-measure when the number of training scenarios increases. With 30 training scenarios, they already achieve an acceptable matching quality.

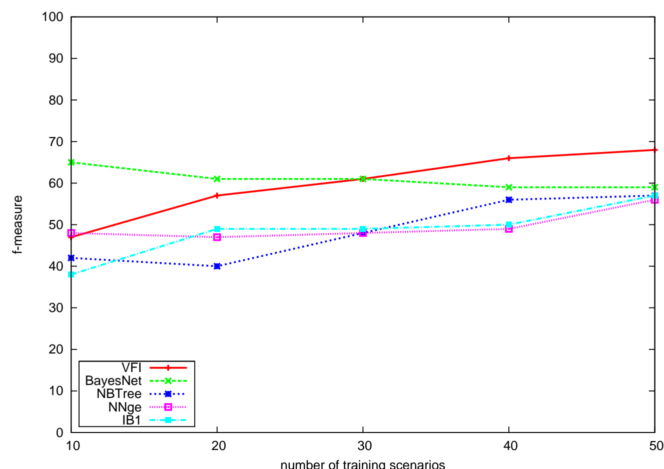


Figure 7: Average f-measure when varying number of training scenarios

Note that the number of training scenarios is not a parameter that the user must manage. Indeed, YAM automatically chooses the number of training scenario according to the matchers that have to be learned. We have run more than 11, 500 experiment results, from which we deduce the number of training scenarios for a given classifier. Table 1 shows the conclusion of our empirical analysis. For instance, when learning a schema matcher based on *J48* classifier, YAM ideally chooses a number of training scenarios between 20 to 30.

Number of training scenarios	Classifiers
20 and less	SLog, ADT, CR
20 to 30	J48, J48graft
30 to 50	NNge, JRip, DecTable BayesNet, VP, FT
50 and more	VFI, IB1, IBk SMO, NBTree, MLP

Table 1: Number of training scenarios chosen by YAM for each classifier

5.3 Precision vs. Recall Preference

We now present another interesting feature of our tool, the possibility to choose between promoting recall or precision, by tuning the weight for false positives. Figures 8(a) and 8(b) respectively depicts the average recall and f-measure of five matchers for 40 scenarios, when tuning the preference between precision and recall. Without any tuning (i.e., weight for false negatives and false positives is equal to 1), this means that we give as much importance to recall and precision.

For 2 matchers (*NBTree* and *NNge*), the recall increases up to 20% when we tune in favor of recall. As their f-measures does not vary, it means that this tuning has a negative impact on the precision. However, in terms of post-match effort, promoting recall may be a better choice depending on the integration task for which matching is being done. For example, let us imagine we have two schemas of 100 elements: a precision which decreases by 20% means a user has to eliminate 20% of the irrelevant discovered correspondences. But a 20% increase in recall means (s)he has 20% less correspondences to search through among 10, 000 possible pairs ! Hence, this tuning could have a tremendous effect on the usability of the matcher for certain tasks.

For the three other matchers (*BayesNet*, *VFI* and *IB1*), tuning in favor of recall has no significant effect. Note that without any tuning, only one matcher (*BayesNet*) has an average recall superior to its precision. Indeed, many of the matchers in our library promote by default precision. But when setting a weight for false negatives to 2, then four matchers have a higher recall than precision. And with a weight for false negatives equal to 3, five other matchers have reduced the gap between precision and recall to less than 5%. Thus, this shows how YAM is able to take into account this very important user preference, which directly impacts post-match (manual) effort.

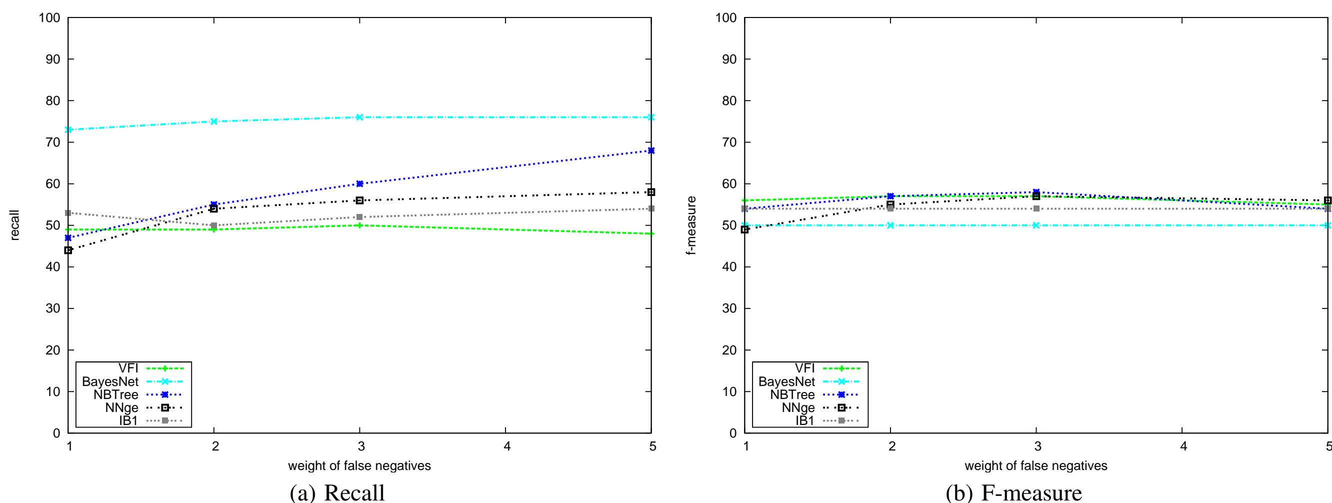
5.4 Impact of Expert Correspondences

As in Glue [6], the number of expert correspondences is an input (compulsory for Glue, but optional for YAM) to the system. YAM can use these expert correspondences to learn better matchers. In this study, we measured the gain in terms of matching quality when a user provides these correspondences.

In these experiments, the training phase used 20 scenarios and expert correspondences were randomly selected. We report the size of the sets of expert correspondences, providing 5% of expert correspondences means that we only give 1 or 2 correspondences as

[2]This classifier is named instance-based since the correspondences (included in the training scenarios) are considered as instances during learning. Our approach does not currently use schema instances.

lirmm-00399025, version 1 - 25 Jun 2009



(a) Recall (b) F-measure

Figure 8: Quality of various matchers when tuning weight of false negatives

input. Figure 9 depicts the average f-measure (on 10 random scenarios) for different matchers.

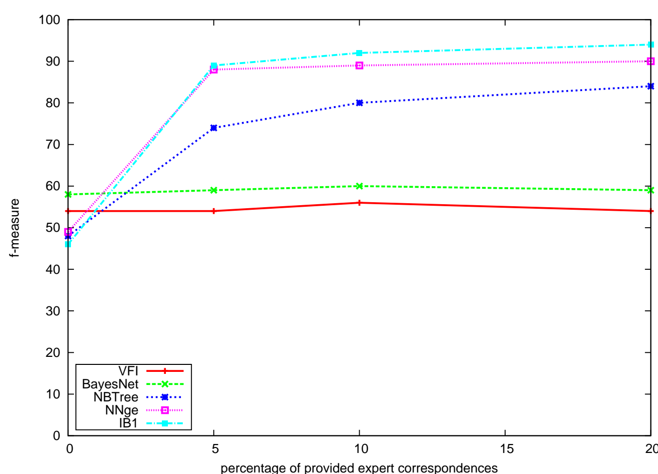


Figure 9: F-measure of various matchers when tuning the input expert correspondences

With only 5% of the correspondences given as expert correspondences, *NNge* and *IB1* are able to increase their f-measure by 40%. The classifier NBTree also achieves an increase of 20%. Similarly, the f-measure of these mathers still increases by as 10% of the correspondences are provided as expert correspondences. On the contrary, the *VFI* and *BayesNet* matchers do not benefit at all from this input. Note that providing some expert correspondences does not require a tiresome effort from the user.[3] Yet, this input can improve the matching quality of most matchers.

5.5 Comparing with Other Matching Tools

We now compare YAM with two matching tools known to provide an good matching quality: COMA++ and Similarity Flooding (SF). To the best of our knowledge, these tools are the only ones publicly available. COMA++ [2] uses 17 similarity measures to build a matrix between pairs of elements and aggregated their similarity values. Similarity Flooding[16] builds a graph between input schemas. Then, it discovers some initial correspondences using a string matching measure. These correspondences are refined using a structural propagation mechanism. Both matching tools are described in more detail in section 6.

As explained in the previous section, a user does not need to choose the number of training scenarios. YAM automatically adjusts this number according to the classifier which is going to be trained. We have trained YAM against 50 random schemas from the KB to generate a robust matcher for each schema matching scenario. Neither COMA++ nor Similarity Flooding can take any expert correspondence as input. Hence, for this comparison, we did not include expert correspondences. Similarly, no weight for false negatives has been set because COMA++ and Similarity Flooding do not have this capability.

5.5.1 *Accuracy Comparison*

Figures 10(a) and 10(b) depict the F-measure obtained by YAM, COMA++ and Similarity Flooding on the 10 scenarios. YAM obtains the highest f-measure in 7 scenarios, and reaches 80% f-measure in 4 scenarios. COMA++ achieves the best f-measure for *currency* and *university* scenarios. SF obtains the best f-measure in one scenario (*travel*). In addition, COMA++ is the only tool which does not discover any correspondence for one scenario (*travel*). However, we notice that YAM obtains better results on the webforms scenarios since it was trained with webforms. With non-webforms scenarios, YAM is able to achieve acceptable results.

These results show how our matcher factory relies on the diversity of classifiers. Indeed, the dedicated matchers that it has generated for these scenarios are based on various classifiers (*VFI*, *BayesNet*, *J48*, etc.) while COMA++ and SF only rely on respectively an aggregation function and a single graph propagation algorithm.

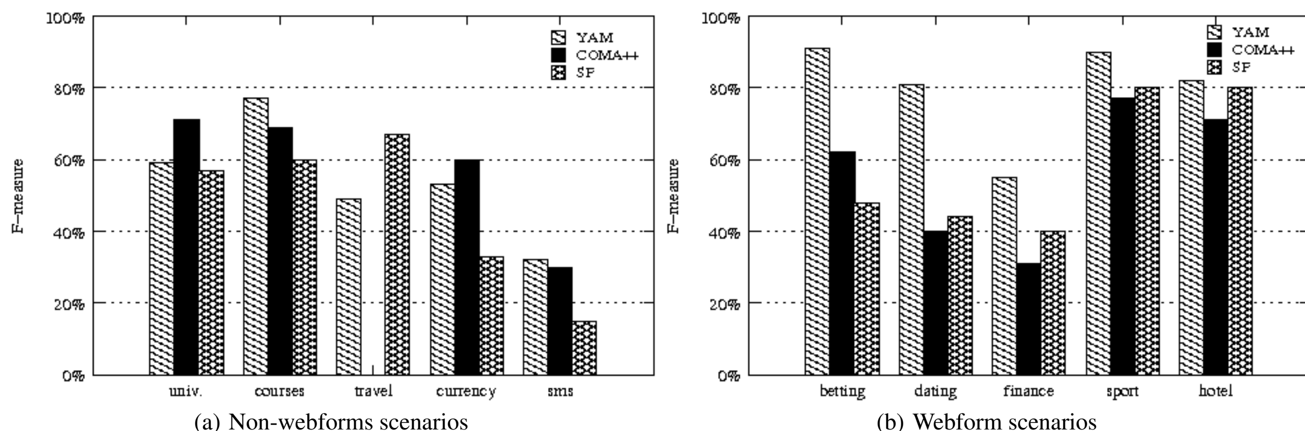
YAM obtains the highest average f-measure (67%) while COMA++ and SF average f-measures are just over 50%. Thus, YAM is a more robust matching tool, specifically because it is able to generate matchers based on various classifiers.

5.5.2 *Post-match Effort*

Most schema matching tools, including most matchers generated by YAM (without tuning), mainly promote precision to the detri-

[3]Some GUIs already exist to facilitate this task by suggesting the most probable correspondences.

lirmm-00399025, version 1 - 25 Jun 2009



(a) Non-webforms scenarios (b) Webform scenarios

Figure 10: Precision, recall and f-measure achieved by the three matching tools on 10 scenarios

ment of recall. However, this is not always the best choice for a given task. The list of discovered correspondences, provided by matching tools, have two issues, namely (i) irrelevant discovered correspondences and (ii) missing correspondences. Users first have to check each correspondence from the list, either to validate or remove it. Then, they have to browse the schemas and discover the missing correspondences. Thus, we propose to evaluate this user post-match effort by counting the number of user interactions. A user interaction is an (in)validation of one pair of schema elements (either from the list of correspondences or between the schemas).

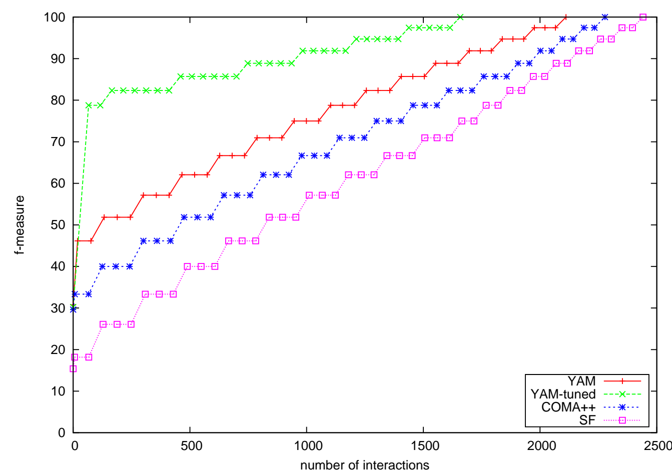


Figure 11: Number of user interactions needed to obtain a 100% **f-measure**

Figure 11 shows the importance of recall by comparing four matching tools (COMA++, SF, YAM and YAM-tuned, which promotes recall with a weight for false negatives set to 2). It depicts user effort, in number of interactions, to reach a 100% f-measure after that the four matching tools have discovered a list of correspondences for the *sms* scenario. Note that this is the worst-case situation, in which all possible pairs must be checked. For the user, the first step of the post-match effort consists in (in)validating discovered correspondences. For instance, YAM-tuned outputs 66 correspondences and thus a user would require 66 interactions to (in)validate them. At the end of first step, all evaluated matchers have a 100% precision. During a second step of the post-match effort, f-measure has the same value distribution for all matchers (since only recall can be improved). This facilitates comparisons between matchers. The second step deals with the manual discovery of all missing correspondences. We assume that all pairs which have not been invalidated yet must be analyzed by the user. Thus, to discover the 20% correspondences missed by YAM-tuned, a user requires about 1600 interactions.

Now let us study figure 11 in more detail. We notice that after the first step, during which a user (in)validates discovered correspondences, YAM, COMA++ and SF only increases their f-measures by a few percent. In contrast, YAM-tuned's f-measure increases from 32% to 80% with only 66 interactions. As a comparison, with the three other matchers, there are at least 1100 user interactions needed to reach this 80% f-measure. Finally, to achieve a 100% f-measure, YAM-tuned interacts 1600 times with the user while other tools requires more than 2100 interactions. Note that these example schemas from *sms* scenario still have reasonable size, but with larger schemas, the post-match effort would require thousands of user interactions. Thus, promoting recall strongly reduces post-match effort. In addition, it enables a user to quickly obtain an acceptable f-measure. Contrary to other matching tools, YAM is able to take this preference for recall into account.

5.6 Discussion

These experiments support the idea that machine learning techniques are suitable for the matching task. We have shown the impact on the matching quality when tuning several parameters, like the number of training scenarios, the tradeoff between recall and precision and the number of input expert correspondences. The first experiment would tailor YAM to automatically adjust the number of training scenarios according to the classifier to be generated. The second study demonstrates how YAM can promote recall (or precision). And the third study describes how users can improve the quality of their results by providing some expert correspondences. We finally compared our approach with two other matching tools to show that YAM outperforms them in most scenarios.

Time performance. Although matching two or more schemas is not time consuming, generating all classifiers and selecting the best one is a time consuming process (up to several hours if the KB of training data is large). This process could be sped up by only generating a subset of available matchers, which are the most appropriate according to the features of the schemas to be matched and user preferences. For instance, if user a wants to promote recall, YAM can only generate matchers based on *BayesNet*, *tuned SMO* and *tuned JRip* which empirically demonstrated good results

lirmm-00399025, version 1 - 25 Jun 2009

for this criterion. Of course, in practice, it is the performance of the dedicated matcher which is crucial and the matchers produced by YAM have comparable or better time performance to other matchers (including COMA++ and Similarity Flooding).

6. RELATED WORK

Much work has been done both in schema matching and ontology alignment. However, we only describe in this section approaches which are based on machine learning techniques, and the tools against which we compared our approach. You can refer to these surveys [11, 17] for more details about other approaches.

In [15], the authors use the *Boosting* algorithm to classify the similarity measures, divided into first line and second line matchers. The Boosting algorithm consists in iterating weak classifiers over the training set while re-adjusting the importance of elements in this training set. An advantage of this algorithm is the important weight given to misclassified pairs. However, the main drawback deals with the Boosting machine learning technique. Although it gives acceptable results, we have noticed in section 5 that several classifiers might give poor results with some scenarios. Thus, only relying on one classifier is risky. Contrary to our work, the authors chose to promote precision and do not accept expert correspondences as input.

MatchPlanner approach [9] makes use of decision trees to select the most appropriate similarity measures. This approach provides acceptable results w.r.t other matching tools. However, the decision trees are manually built, thus requiring an expert intervention. Besides, decision trees are not always the best classifier, as shown in section 5.

eTuner [14] aims at automatically tuning schema matching tools. It proceeds as follows: from a given schema, it derives many schemas which are semantically equivalent. The correspondences between the initial schema and its derivations are stored. Then, a given matching tool (e.g., COMA++ or Similarity Flooding) is applied against the set of correspondences until an optimal parameters configuration of the matching tool is found. eTuner strongly relies on the capabilities of the matching tools. Conversely, YAM learns a dedicated matcher for a given scenario. It is also able to integrate important feature like user preference between recall and precision. Contrary to eTuner, YAM is extensible in terms of similarity measures and classifiers, thus enhancing the possibilities of our tool.

COMA/COMA++ [2] is a generic, composite matcher with very effective match results. It can process the relational, XML, RDF schemas as well as ontologies. Internally it converts the input schemas as trees for structural matching. For linguistic matching it utilizes a user defined synonym and abbreviation tables, along with n-gram name matchers. Similarity of pairs of elements is calculated into a similarity matrix. At present it uses 17 element level similarity measures. For each source element, elements with similarity higher then than threshold are displayed to the user for final selection. The COMA++ supports a number of other features like merging, saving and aggregating match results of two schemas. On the contrary, our approach is able to learn the best combination of similarity measures instead of using the whole set. It outperforms COMA++ both in matching quality and time performance.

S-MATCH/S-MATCH++ [3] takes two directed acyclic graphs like structures e.g., XML schemas or ontologies and returns semantic correspondences between pairs of elements. It uses external dictionary Wordnet, to evaluate the linguistic matching along with its structural matcher to return a subsumption type correspondence. At present, it uses 13 element level matchers and 3 structural level matchers. It is also heavily dependent on SAT solvers, which decrease its time efficiency.

Similarity Flooding [16] has been used with Relational, RDF and XML schemas. These schemas are initially converted into labeled graphs and SF approach uses fix-point computation to determine correspondences of 1:1 local and m:n global cardinality between corresponding nodes of the graphs. The algorithm has been implemented as a hybrid matcher, in combination with a name matcher based on string comparisons. First, the prototype does an initial element-level name mapping, and then feeds these mappings to the structural SF matcher for the propagation process. The weight of similarity between two elements is increased, if the algorithm finds some similarity between the related elements of the pair of elements. In a modular architecture, the components of SF, such as schema converters, the name and structural matchers, and filters, are available as high-level operators and can be flexibly combined within a script for a tailored match operation. One of the main drawback of Similarity Flooding is the matching quality. But this weak point is compensated by the performance. Our approach goes further by using different matching techniques selected by a planner.

Glue [6], and its predecessor LSD, are also based on machine learning techniques. They have four different learners, which exploit different information of the instances. The name learner (Whirl, a nearest-neighbor classifier) makes predictions using word frequency (TF/IDF distance) on the label of the schema elements. The content learner (also based on Whirl and TF/IDF) applies a similar strategy on the instances associated to each schema element. A Naive Bayes classifier considers labels and attributes as a set of tokens to perform text classification. The XML learner (based on Naive Bayes too) exploits the structure of the schema (hierarchy, constraints, etc.). Finally, a meta-learner, based on stacking, is applied to return a linear weighted combination of the four learners. First, the user must give some correspondences between the schemas that require to be matched. These correspondences are then used for training the learners, and their combination resulting from the meta-learner is performed against the input schemas to discover the rest of the correspondences. YAM is also based on machine learning techniques. However, the user correspondences are optional, since a knowledge base is provided with our tool. Authors of Glue do not detail how many correspondences should be given by the user. We have shown in section 5.2 that some classifiers require many training scenarios (and thus correspondences) to be efficient. YAM generates a dedicated matcher, based on a classifier which best combines all similarity measures. On the contrary, Glue uses classifiers on the same similarity measures. The meta-learner is a linear regression function, with its drawbacks in terms of quality and extensibility, as explained in [9].

AUTOMATCH [4] is the predecessor of AUTOPLEX, which uses schema instance data and machine learning techniques to find possible correspondences between two schemas. A knowledge base, called attribute dictionary, contains attributes with a set of possible instances and their probability. This dictionary is populated using Naive Bayesian algorithm to extract relevant instances from Relational schemas fields. A first step consists of matching each schema element to dictionary attributes, thus computing a similarity value between them according to the number of common instances. Then, the similarity values of two schema elements that match the same dictionary attribute are summed and *minimum cost*

lirmm-00399025, version 1 - 25 Jun 2009

maximum flow algorithm is applied to select the best correspondences. The major drawback of this work is the importance of the data instances. Although this approach is interesting on the machine learning aspect, that matching is not as robust since it only uses one similarity function based on a dictionary.

7. CONCLUSION

In this paper, we have presented YAM, a factory of schema matchers. During pre-match phase, it generates, thanks to machine learning algorithms, a dedicated matcher for a given matching scenario. Experiments have shown that the dedicated matcher obtains acceptable results with regards to other matching tools. Besides, the possibility to learn matchers whose algorithm is completely different enables to efficiently match specific scenarios. In the experiments, our empirical analysis enables to adjust YAM so that it automatically chooses the number of training scenarios according to the classifier that it has to generate.

Similarly to other approaches, we enable the user to provide some initial expert correspondences. As most classifiers are able to efficiently use this input, generated matchers are better appropriate for a given matching scenario. As a result, matching quality strongly increases, even when providing only 1 or 2 expert correspondences.

Our approach is also the first work to let users choose the promoting of either precision or recall. This choice strongly impacts the post-match effort, when user (in)validates discovered correspondences and manually browse schemas to find missing ones. We have demonstrated that promoting recall is more appropriate to reduce user post-match interactions.

In the future, we first plan to test more matchers. Indeed, there exists plenty of machine learning classifiers among which we only experiment a subset. Another ongoing work consists in reducing the learning time. To tackle this issue, we explore the possibility to reuse previously generated matchers, which are stored in the KB. And we intend to use case-based reasoning techniques to select the dedicated one among them.

8. REFERENCES

[1] The UIUC web integration repository. Computer Science Department, University of Illinois at Urbana-Champaign. http://metaquerier.cs.uiuc.edu/repository, 2003.

[2] D. Aumueller, H. H. Do, S. Massmann, and E. Rahm. Schema and ontology matching with coma++. In *SIGMOD Conference, Demo paper*, pages 906–908, 2005.

[3] P. Avesani, F. Giunchiglia, and M. Yatskevich. A large scale taxonomy mapping evaluation. In *International Semantic Web Conference*, pages 67–81, 2005.

[4] J. Berlin and A. Motro. Database schema matching using machine learning with feature selection. In *CAiSE*, 2002.

[5] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. In *SIGMOD*, pages 509–520, 2001.

[6] A. Doan, J. Madhavan, R. Dhamankar, P. Domingos, and A. Y. Halevy. Learning to match ontologies on the semantic web. *VLDB J.*, 12(4):303–319, 2003.

[7] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Ontology matching: A machine learning approach. In *Handbook on Ontologies in Information Systems*, 2004.

[8] C. Drumm, M. Schmitt, H. H. Do, and E. Rahm. Quickmig: automatic schema matching for data migration projects. In *CIKM*, pages 107–116. ACM, 2007.

[9] F. Duchateau, Z. Bellahsene, and R. Coletta. A flexible approach for planning schema matching algorithms. In *OTM Conferences (1)*, pages 249–264, 2008.

[10] F. Duchateau, Z. Bellahsene, and M. Roche. A context-based measure for discovering approximate semantic matching between schema elements. In *RCIS*, 2007.

[11] J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007.

[12] S. R. Garner. Weka: The waikato environment for knowledge analysis. In *In Proc. of the New Zealand Computer Science Research Students Conference*, pages 57–64, 1995.

[13] J. Hammer, M. Stonebraker, , and O. Topsakal. Thalia: Test harness for the assessment of legacy information integration approaches. In *Proceedings of ICDE*, pages 485–486, 2005.

[14] Y. Lee, M. Sayyadian, A. Doan, and A. Rosenthal. etuner: tuning schema matching software using synthetic scenarios. *VLDB J.*, 16(1):97–122, 2007.

[15] A. Marie and A. Gal. Boosting schema matchers. In *OTM Conferences (1)*, pages 283–300, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] S. Melnik, H. G. Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Data Engineering*, pages 117–128, 2002.

[17] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.