(Not) Yet Another Matcher[*]

Fabien Duchateau, Remi Coletta, Zohra Bellahsene
LIRMM, Univ. Montpellier 2
34392 Montpellier, France
firstname.name@lirmm.fr

Renée J. Miller
Univ. of Toronto
40 St. George Street
Toronto ON M5S 2E4, Canada
miller@cs.toronto.edu

ABSTRACT

Discovering correspondences between schema elements is a crucial task for data integration. Most schema 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 schema 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 is able to generate the best matcher for a given scenario.

Classification: H.2. DATABASE MANAGEMENT
General Terms: Experimentation
Keywords: schema matching, data integration, matcher factory, machine learning, classifier, XML schemas.

1. INTRODUCTION

There is 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 schema matching 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 schemas 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.

Most schema 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. [7] 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 a specific *matching scenario*, that is, a 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. 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. Like previous works [2, 3, 8, 4], 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.

Contributions. The main features of our approach are:

- YAM is the first schema matcher factory capable of generating a dedicated matcher for a given scenario. YAM does more than tuning, it both selects and trains a dedicated matcher from a large library of matchers.
- We compare the dedicated matchers produced by YAM with two high-performing traditional matchers, COMA++ and Similarity Flooding. We show that our dedicated matchers significantly out perform traditional matchers when trained on

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

relevant matching scenarios. Even when training is done on a matching scenario that is not relevant (to the schemas to be matched), we show that YAMS's dedicated matchers may still achieve comparable quality to traditional matchers.

2. LEARNING A DEDICATED MATCHER

In this section, we describe YAM's approach for learning a schema matcher (which we call a dedicated matcher) for a given schema 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 its result – classification, clustering, an aggregation over similarity measures, or any number of ad hoc methods including techniques like blocking to improve its efficiency.

YAM 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.

2.1 Matcher Training

YAM trains each matcher using its knowledge base of training data. We begin our explanation with an example.

Example: Let us consider the pair *(searchform, search)*, for which three similarity values have been computed: *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 uses correspondences stored in a knowledge-base (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 training data is composed of the following expert correspondences *(* City:, City)* and *(State:, State)*, terminological measures like *JaroWinkler* or *Levenshtein* will be first considered by the machine learning algorithms. Indeed, they enable a correct classification of both pairs. Thus, if the trained matcher is an aggregator function, it gives a heavy weight to these terminological measures while structural or linguistic measures would have a weight equal to 0. In the case of training a decision tree, terminological measures are placed at the top of the tree, and others might not appear or be at the bottom of the tree [4].

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

2.2 Selecting a Dedicated Matcher

A dedicated matcher is selected according to its accuracy on the given training data. Thus, YAM cross-validates each matcher with the correspondences stored in the KB. 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.

3. EXPERIMENTS

In these experiments, we first 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. Then, we demonstrate that YAM is able to produce an effective dedicated matcher. Finally, we compare our results with two matching tools that have excellent matching quality, COMA++ [1] and Similarity Flooding [9].

Scenarios. To demonstrate the effectiveness of our approach, we used several schema matching scenarios: **university** from [3], **thalia** benchmark [1] , **travel** schemas[2], **currency** and **sms** web services[3] and web forms related to different domains (*hotel booking*, *dating*, *betting*, etc.) extracted by authors of [8]. 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.

Quality metrics. 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.

YAM architecture and prototype are described in [5]. The current version of YAM includes 20 classifiers from the Weka library[4] and 30 similarity measures, including all the terminological measures from the Second String project[5], and some structural and semantic measures. Experiments were run on a 3.6 Ghz computer with 4 Go RAM under Ubuntu 7.10. We used 200 runs to minimize the impact of randomness during training.

3.1 Impact of the Training Scenarios

Figure 1 depicts the average f-measure of several matchers (restricted to 5 for clarity) as we vary the number of training scenarios. Note that the average f-measure has been computed over 40 randomly selected scenarios. 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[6], 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.

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.

[1]http://www.cise.ufl.edu/research/dbintegrate/thalia
[2]http://metaquerier.cs.uiuc.edu/repository
[3]http://www.seekda.com
[4]https://svn.scms.waikato.ac.nz/svn/weka
[5]http://secondstring.sourceforge.net
[6]This classifier is named instance-based since the correspondences (included in the training scenarios) are considered as instances during learning. Our approach does not use schema instances.

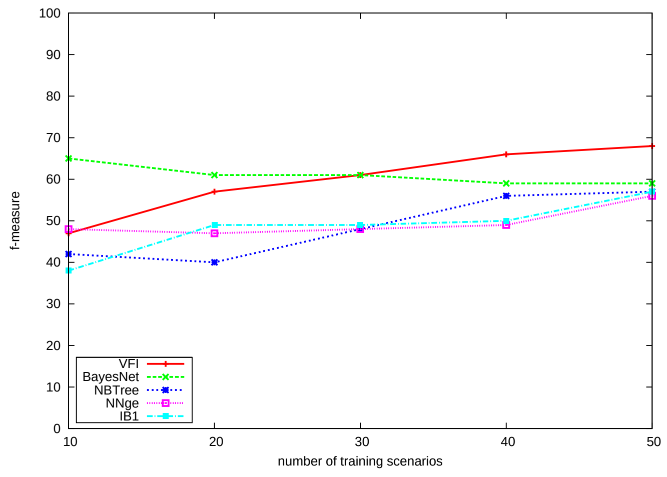


Figure 1: Average f-measure when varying number of training scenarios

# 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 for each classifier

3.2 Comparing generated matchers

We now study which matchers were selected for different schema matching scenarios. This study highlights the variability in matching quality of each matcher on different scenarios, and therefore the importance of a 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. Figure 2 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. We also note that aggregator 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. 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.

3.3 Comparing with Other Matching Tools

We now compare YAM with two matching tools known to provide good matching quality: COMA++ and Similarity Flooding (SF). Both matching tools are described in more detail in section 4. To the best of our knowledge, these tools are the only ones publicly available. As explained in the previous section, the user does not

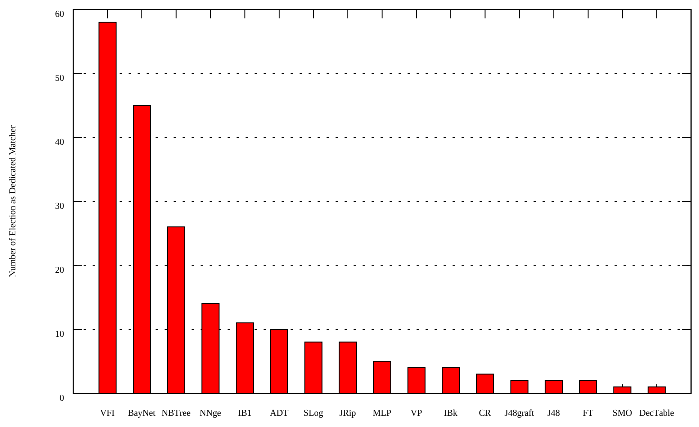


Figure 2: Number of selections as dedicated matcher

need to choose the number of training scenarios. YAM automatically adjusts this number according to the classifier which is going to be trained.

Figures 3(a) and 3(b) depict the F-measure obtained by YAM dedicated matcher, 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 generates more robust schema matchers, specifically because they are based on various classifiers.

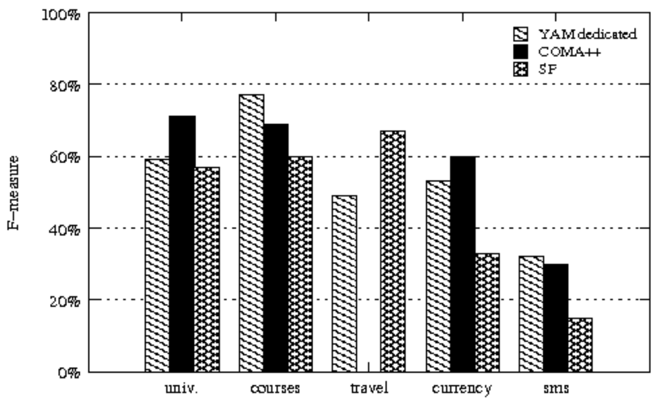
3.4 Discussion

These experiments support the idea that machine learning techniques are suitable for the schema matching task. First, we have shown that YAM has generated, for 200 schema matching scenarios, different dedicated matchers (i.e., based on different algorithms such as decision trees, rules, aggregation functions, etc.). The second experiment would tailor YAM to automatically adjust the number of training scenarios according to the classifier to be generated. We finally compared our approach with two other matching tools to show that YAM outperforms them in most scenarios.

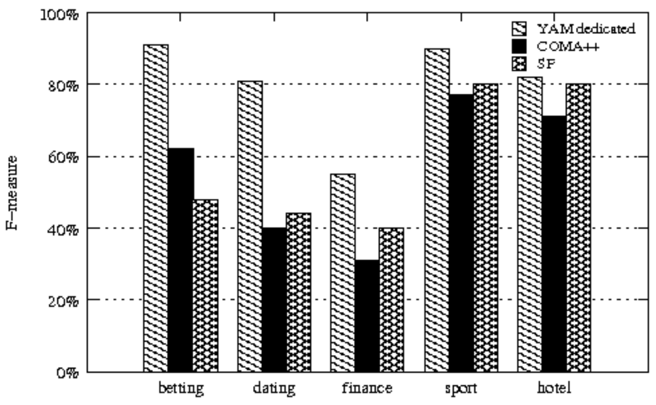
Generating all classifiers and selecting the best one is a time consuming process (up to several hours if the KB of training data is large). Of course, in practice, it is the performance of the dedicated matcher which is crucial and the matchers produced by YAM have comparable or better time performance to other matchers (including COMA++ and Similarity Flooding).

4. RELATED WORK

Much work has been done both in schema matching and ontology alignment [6, 10]. However, we only describe in this section



(a) Non-webforms scenarios



(b) Webform scenarios

Figure 3: Precision, recall and f-measure achieved by the three matching tools on 10 scenarios

schema matching approaches which are based on machine learning techniques, and the tools against which we compared our approach.

In [8], the authors use the *Boosting* algorithm to classify the similarity measures, by iterating weak classifiers over the training set while re-adjusting the importance of elements in this training set. The main drawback deals with the Boosting machine learning technique. Although it gives acceptable results, we have noticed in section 3 that several classifiers might give poor results with some scenarios. Thus, only relying on one classifier is risky.

MatchPlanner approach [4] 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 and they are not always the best classifier, as shown in section 3.

eTuner [7] aims at automatically tuning schema matching tools. A given matching tool is applied against the set of correspondences until an optimal parameter configuration of the matching tool is found. Thus, eTuner strongly relies on the capabilities of the matching tools. Conversely, YAM learns a dedicated matcher for a given scenario. Besides, it is extensible in terms of similarity measures and classifiers, thus enhancing its capabilities for efficiently handling of new schema matching scenarios.

COMA/COMA++ [1] is a generic, composite matcher with very effective matching results. The similarity of pairs of elements is calculated linguistic and terminological measures. Then, a strategy is applied to determine the pairs that are presented as mappings. 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.

Glue/LSD [3] is also based on machine learning techniques, with four different learners, which exploit different information of the instances. Then, a meta-learner, based on stacking, is applied to return a linear weighted combination of the four learners. However, 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 [4].

Similarity Flooding [9] is a neighbour affinity matching tool. First, it applies a terminological measure to discover initial correspondences, and then feeds them to the structural matcher for propagation. The weight of similarity values between two elements is increased, if the algorithm finds some similarity between related elements of the pair. With strongly heterogeneous labels or with small schemas, SF may obtain a low matching quality.

In AUTOMATCH/AUTOPLEX [2], schema elements are matched to a dictionary (i.e., a knowledge base populated with relevant data instances thanks to Naive Bayesian algorithm). Then, the similarity values of two schema elements that match the same dictionary attribute are summed and *minimum cost maximum flow* algorithm is applied to select the best correspondences. The major drawback of this work is the importance of the data instances. Besides, it only uses one similarity measure based on a dictionary.

5. CONCLUSION

In this paper, we have presented YAM, a factory of schema matchers. During a pre-match phase, it generates, thanks to machine learning algorithms, a dedicated matcher for a given schema matching scenario. Experiments have shown that the dedicated matcher obtains acceptable results with regards to other matching tools. Besides, the possibility to learn schema matchers based on different algorithms enables to efficiently match specific scenarios.

In the future, we first plan to test more machine learning classifiers. Another ongoing work consists in reducing the learning time.

6. REFERENCES

[1] 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.

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

[3] 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.

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

[5] F. Duchateau, R. Coletta, Z. Bellahsene, and R. J. Miller. Yam: a schema matcher factory (demo). In *CIKM*, 2009.

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

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

[8] A. Marie and A. Gal. Boosting schema matchers. In *OTM Conferences (1)*, pages 283–300, 2008.

[9] 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.

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