YAM: a Step Forward for Generating a Dedicated Schema Matcher

Fabien Duchateau[1], Zohra Bellahsene[2]

[1]Université Lyon 1, LIRIS UMR5205, France
fabien.duchateau@univ-lyon1.fr
[2]Université Montpellier, LIRMM, France
bella@lirmm.fr

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 certain parameters (thresholds, weights, etc.). They mainly use aggregation methods to combine similarity measures. The tuning of a matcher, especially for its aggregation function, has a strong impact on the matching quality of the resulting correspondences, and makes it difficult to integrate a new similarity measure or to match specific domain schemas. In this paper, we present YAM (Yet Another Matcher), a matcher factory which enables the generation of a dedicated schema matcher for a given schema matching scenario. For this purpose we have formulated the schema matching task as a classification problem. Based on this machine learning framework, YAM automatically selects and tunes the best method to combine similarity measures (e.g., a decision tree, an aggregation function). In addition, we describe how user inputs, such as a preference between recall or precision, can be closely integrated during the generation of the dedicated matcher. Many experiments run against matchers generated by YAM and traditional matching tools confirm the benefits of a matcher factory and the significant impact of user preferences.

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

1 Introduction

There are a plethora of schema matching tools designed to help automate what can be a painstaking task if done manually [3]. The diversity of tools hints at the inherent complexity of this problem. The proliferation of schema matchers and the proliferation of new (often domain-specific) similarity measures used within these matchers have 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. Traditionally, the matcher, which combines various similarity measures, is based on an aggregation function. 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 [39]. Lee et al. have shown how important (and difficult) tuning is, and that without tuning most matchers perform poorly, thus leading to a low quality of the data integration process [27]. To overcome this, they proposed *eTuner*, a supervised learning approach for tuning these matching knobs. However, eTuner has to be plugged into a matching tool, which requires programming skills. A user must also still commit to one single matcher (the matcher provided in the matching tool). Several research papers [10, 12, 18, 31] led to the conclusion that matchers based on machine learning provide acceptable results w.r.t. existing tools. The main idea consists of training various similarity measures with a sample of schemas and correspondences, and applying them to match another set of schemas. Our intuition is that machine learning can be used at the matcher level.

Another motivation deals with the pre-match interaction with the users [38]. They usually have some preferences or minor knowledge of the schemas to be matched, which are rarely used by the schema matchers [43]. For instance, a quick examination of the schemas may have revealed a few correct correspondences, or the user may have an external resource (dictionary, ontology) or a dedicated similarity measure which could be exploited for a specific domain. 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, i.e., it may only produce a correspondence if it is close to 100% confident of the correspondence's accuracy. In other words, such a matcher uses 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 task. Integrating these user preferences or knowledge prior to the matching is a challenge for improving the quality results of a schema matcher.

In this context, we present **YAM**, which is actually not ***Y**et **A**nother **M**atcher*[1]. Rather YAM is the first schema matcher generator designed to produce a tailor-made matcher, based on the automatic tuning of the matcher and the optional integration of user requirements. While schema matching tools produce correspondences between schemas, YAM is a matcher factory because it produces a dedicated schema matcher (that can be used later for discovering correspondences). This means that theoretically, YAM could generate schema matchers which are very similar to the tools COMA++ [2] or MatchPlanner [18]. Schema matching tools only have one predefined method for combining similarity measures (e.g., a weighted average), while a schema matcher generated by YAM in-

[1] The name of the tool refers to a discussion during a panel session at XSym 2007

cludes a method selected among many available (e.g., a decision tree, a Bayesian network, a weighted average). To fulfill this goal, YAM considers the schema matcher as a machine learning classifier: given certain features (i.e., the similarity values computed with various similarity measures), the schema matcher has to predict the relevance of a pair of schema elements (i.e., whether this pair is a correspondence or not). In this framework, any type of classifier can be trained for schema matching, hence the numerous methods available for combining similarity measures. YAM does not only select the best method for combining similarity measures but it also automatically tunes the parameters inherent to this method (e.g., weights, thresholds). The automatic tuning capability has been confirmed as one of the challenges proposed in [43]. In addition, YAM integrates user preferences or knowledge (about already matched scenarios and training data), if available, during the generation of the schema matchers. For instance, our approach benefits from expert correspondences provided as input, because they are used for generating the schema matcher. In YAM, we also allow a user to specify her/his preference for precision or recall, and we produce a dedicated matcher that best meets the users needs. YAM is the first tool that allows the tuning of this very important performance trade-off.

The main contributions in this paper are:

– Our approach is the first to refer to **schema matching as a classification task**. Although other approaches may use classifiers as a similarity measure, we propose to consider a schema matcher which combines various similarity measures as a classifier.
– In addition, our work is the first **matcher factory** for schema matching. Contrary to traditional matching approaches, our factory of matchers generates different matchers and selects the dedicated matcher for a given scenario, i.e., the matcher which includes the most relevant similarity measures, combined with the most appropriate method, and best tuned.
– Another contribution deals with the **close integration of user preferences** (e.g., preference between precision or recall, expert correspondences).
– A **tool named YAM** has been implemented. Its main features include self-tuning (the method for combining similarity measures in a matcher is tailored to the schemas to be matched) and extensibility (new similarity measures or classifiers can be added with no need for manual tuning).
– Experiments over well-known datasets were performed to **demonstrate the significant impact of YAM** at different levels: the need for a matcher factory, the benefit of user preferences on the matching quality, and the evaluation with other schema matching tools in terms of matching quality and performance time.

Outline. The rest of the paper is organized as follows. Section 2 contains the necessary background and definitions of the notions and concepts that are used in this paper. Section 3 gives an overview of our approach while Section 4 provides the details of the learning process. The results of experiments 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

Schema matching is traditionally applied to matching pairs of edge-labeled trees (a simple abstraction that can be used for XML[2] schemas, web interfaces, JSON[3] data types, or other semi-structured or structured data models). The 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.) or to set up parameters (tuning of weights, thresholds, etc.). Secondly, the **matching process** occurs, during which correspondences are discovered. The final step, the **post-match process**, mainly consists of 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.

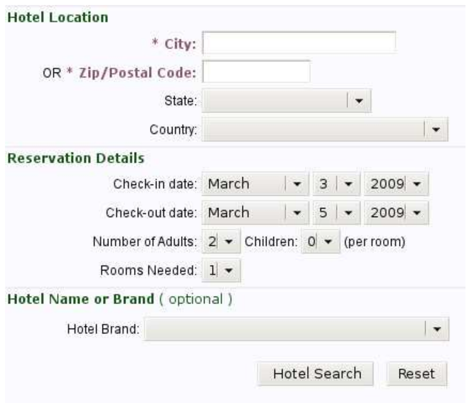
Definition 2 (Schema matching scenario): A schema matching scenario is a set of schemas (typically from the same domain, e.g., *genetics* or *business*) that need to be matched. A scenario may reflect one or more properties (e.g., domain specific, large scale schemas). An example of schema matching scenario is composed of two *hotel booking* web forms, such as those depicted by Figures 1(a) and 1(b). Optionally, a schema matching scenario can include user preferences (preference for precision or recall, expert correspondences, scenarios from the same domain, number of training data and choice of the matching strategy). These options are detailed in Section 4.3. In the next definitions, we focus on a scenario with two schemas S_1 and S_2 for clarity, but the definitions are valid for a larger set of schemas.

Definition 3 (Dataset): A dataset is a set of schema matching scenarios. For instance, the dataset used in the experiments of this paper is composed of 200 scenarios from various domains.

Definition 4 (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. For instance, a pair from the two *hotel booking* schemas is $<$*city, hotel name*$>$.

Definition 5 (Similarity Measure): A similarity measure is a function which computes a similarity value between a pair of schema elements $<e_1, e_2>$.

[2] Extensible Markup Language (XML) (November 2015)
[3] JavaScript Object Notation (JSON) (November 2015)


Hotel Location

* City:

OR * Zip/Postal Code:

State:

Country:

Reservation Details

Check-in date: March 3 2009

Check-out date: March 5 2009

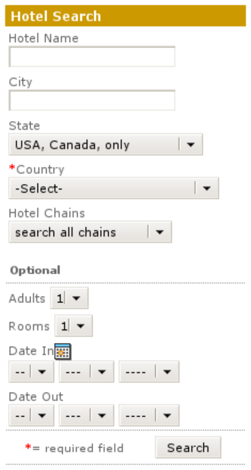
Number of Adults: 2 Children: 0 (per room)

Rooms Needed: 1

Hotel Name or Brand (optional)

Hotel Brand:

(a) Web form *hotels-valued*


Hotel Search

Hotel Name

City

State: USA, Canada, only

*Country: -Select-

Hotel Chains: search all chains

Optional

Adults 1

Rooms 1

Date In -- --- ----

Date Out -- --- ----

*= required field

(b) Web form *where-to-stay*

Fig. 1. Two web forms about hotel booking

The similarity value is noted $sim(e_1, e_2)$ and it indicates the likeness between both elements. It is defined by:

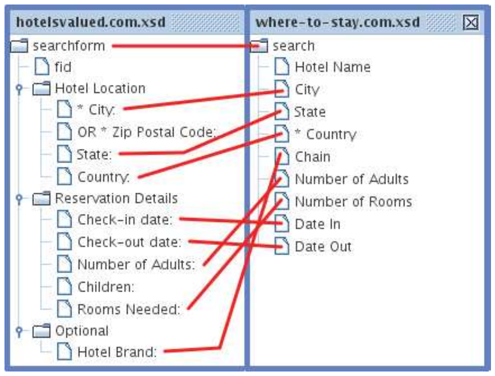
$$sim(e_1, e_2) \rightarrow [0, 1]$$

where a *zero* value means total dissimilarity and a value equal to *one* stands for total similarity. Note that measures computed in $\Re$ can usually be converted in the range $[0, 1]$. In the last decades, many similarity measures have been defined [20, 25, 29] and are available in libraries such as Second String[4].

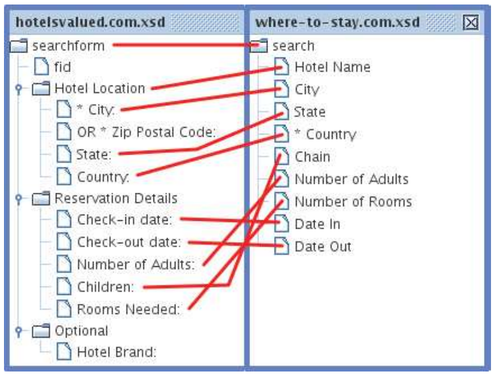
Definition 6 (Correspondence): A correspondence is a pair of schema elements which are semantically similar. It is defined as a tuple $<e_1, e_2, k>$, where k is a confidence value (usually the average of all similarity values computed for the pair $<e_1, e_2>$). A set of correspondences can be provided by an expert (ground truth) or it may be produced by schema matchers. Figure 2 depicts two sets of correspondences. The set on the left side (Figure 2(a)) is the expert set, which includes expected correspondences. The set on the right side (Figure 2(b)) has been discovered by a schema matcher (YAM). Note that an expert correspondence traditionally has a similarity value equal to 1. As an example, $<$*searchform, search,* 1$>$ is an expert correspondence.

Definition 7 (Schema matcher): A schema matcher is an algorithm or a function which combines similarity measures (e.g., the average of similarity values for a pair of schema elements). In addition, a matcher includes a decision

[4] Second String (November 2015)



(a) Expert set



(b) YAM's set (using a dedicated matcher)

Fig. 2. Two sets of correspondances for the hotel booking example

step to select which pair(s) become correspondences (e.g., the decision may be a threshold or a top-K). Given a pair $<e_1, e_2>$ and its similarities values computed for k similarity measures, we represent the combination *comb* and the decision *dec* of a matcher as follows:

$$dec(comb(sim_1(e_1, e_2), \ldots, sim_k(e_1, e_2))) \rightarrow <e_1, e_2, k> \text{ or } \emptyset$$

A few matchers also use information about other pairs (e.g., $<e_1, e_3>$) to decide whether the pair $<e_1, e_2>$ is a correspondence or not. Thus, a more generic definition of a schema matcher M focuses on the fact that it produces a set of correspondences between two schemas S_1 and S_2:

$$M(S_1, S_2) \rightarrow \{<e_1, e_2, k>\}$$

Definition 8 (Schema matching quality): The schema matching quality evaluates the effectiveness of a schema matcher for a given schema matching scenario, by comparing the correspondences discovered by this matcher for the scenario to a ground truth. Figure 2(a) provides the ground truth between the two web forms. Different metrics have been designed to measure the effectiveness of a matcher. For instance, precision measures the rate between the number of correct correspondences discovered by a matcher and the number of expected correspondences (provided in the ground truth).

One of the main assumptions in YAM is that the schema matching process can be seen as a binary classification algorithm. Indeed, the goal of the schema matching process is to determine whether a pair of schema elements is a correspondence or not. For instance, a matcher will have to classify the pair *<city, hotel name>* either as a correct or an incorrect correspondence. Thus, YAM views schema matchers as machine learning classifiers [34].

Definition 9 (Classifier): A classifier is an algorithm that aims at sorting input data into different classes. In our context, two classes indicate the validity of a pair to be considered as a correspondence: *relevant* and *irrelevant*. Different types of classifiers are available such as decision trees or support vector machines [23,34]. Two processes are usually associated to a classifier: *training* (or learning) consists of building a classifier from a given type by exploiting training data while *using* stands for the application of the trained classifier against another dataset. To train a classifier[5], training data is described with features and a class. In our context, the set of training data τ is represented with similarity measures and associated values and its validity v:

$$\tau{=}\{((sim_1, sim_1(e_1, e_2)), \dots, (sim_k, sim_k(e_1, e_2)), v)\}$$

Given a set of training data τ, the training for a type of classifier ω produces a classifier c as follows:

$$\omega(\tau) \rightarrow c$$

When training a classifier, the main objective is to promote a given evaluation metric, and the chosen metric for YAM is the misclassification rate. At the end of the learning, the generated classifier efficiently combines (a subset of) similarity measures. In our context, using a classifier is equivalent to a schema matcher, i.e., it produces a set of schema correspondences between two input schemas:

$$c(S_1, S_2) \rightarrow \{<e_1, e_2, k>\}$$

Definition 10 (Matcher factory): A factory of matchers such as YAM produces different schema matchers based on the same inputs. Each of those matchers has its own specificity, mainly for combining similarity measures, tuning internal parameters, taking a decision. In our context, these specificities mainly relate to the types of classifiers. Given two schemas S_1 and S_2, a set of training data τ, a set of type of classifiers Ω, and optional user preferences Φ, a matcher factory generates a set of generated matchers $\mathcal{C}$, i.e., one matcher for each type of classifier.

[5] We focus on supervised classification, i.e., all training data are labelled with a class.

$$M(S_1, S_2, \tau, \Omega, \Phi) \rightarrow \mathcal{C} \qquad \text{where } \mathcal{C} = \{c_1, \ldots, c_n\} \text{ and } |\Omega| = n$$

Definition 11 (Dedicated matcher): The motivation for generating many classifiers within a factory comes from the fact that a given schema matcher, even craftily tuned, may not reach the quality level of another matcher for a given scenario [27]. Yet, no schema matcher performs well in all possible scenarios. In addition to generating many matchers, a factory of schema matchers is also in charge of selecting the dedicated schema matcher, i.e., the "best matcher among all those generated". The notion of "best matcher" depends on a strategy which encompasses three criteria: an evaluation metric, a validation dataset and a pool of matchers. Strategies are described in more detail in Section 4.4. Broadly speaking, given an evaluation metric μ, a set of matchers $\mathcal{C}$ and a validation dataset $\mathcal{X}$, the dedicated matcher $\Gamma \in \mathcal{C}$ is the classifier which obtains the highest score for the evaluation metric against the validation dataset:

$$\forall c_i \in \mathcal{C}, \mu(\Gamma, \mathcal{X}) \geq \mu(c_i, \mathcal{X})$$

This dedicated matcher Γ is then used for matching S_1 and S_2.

3 Overview of our approach

YAM is a self-tuning and extensible matcher factory tool, which generates a dedicated schema matcher according to a scenario and optional user requirements. Broadly speaking, YAM includes a repository of training data (scenarios with expert correspondences) and a set of types of classifier. It generates tuned schema matchers for various types of classifier, and then select the "best" one - according to a strategy - as the dedicated matcher. This dedicated matcher can be used for matching the input scenario. The **self-tuning** feature stands for the ability to produce a matcher with appropriate characteristics for a given scenario, mainly the method for combining similarity measures (aggregation functions, Bayes network, decision trees, etc.). The **extensible** feature enables users of a matching tool to add new similarity measures. Traditional matching tools which offer this extensibility are often restricted by the manual update of the configuration for both the similarity measures and the method which combines them (e.g., adjusting thresholds, re-weighting values). However, YAM automatically tunes these parameters and is therefore easily extensible. Finally, the **integration of user requirements** allows YAM to convert user time spent to specify these requirements into better quality results, mainly by generating matchers specifically tuned for the scenario. YAM provides these three capabilities because it is based on machine learning techniques, as described in the next part on the architecture. The last part of the section illustrates a running example with YAM.

3.1 Architecture of the YAM system

To the best of our knowledge, YAM is the first factory of schema matchers and it aims at generating a dedicated schema matcher (i.e., a craftily tuned schema matcher) for a given schema matching scenario. For this purpose, YAM uses machine learning techniques during the pre-match phase.

Figure 3 depicts the architecture of YAM. The circles represent inputs or outputs and the rectangles stand for processes. Note that a dotted circle means that such an input is optional. YAM requires only one input, the set of schemas to be matched. However, the user can also provide additional inputs, i.e., preferences and/or expert correspondences (from a domain of interest, or for the input schemas to be matched). The preferences consist of a precision and recall trade-off and a strategy to select the dedicated matcher. In YAM, a repository stores a set of classifiers (currently 20 from the Weka library [23]), a set of similarity measures (mainly from the Second String project [41]), a set of training data (200 schema matching scenarios from various domains with their expert correspondences). The **schema matcher generator** is in charge of generating one tuned matcher for each classifier in the repository according to the user inputs (see Sections 4.2 and 4.3). Then, the **schema matcher selector** applies a strategy to choose the dedicated matcher among all the tuned matchers (see Section 4.4). Finally, this dedicated schema matcher can be used for matching, and specifically the input schemas for which it was tailored. This matching process produces a list of correspondences discovered between the input schemas. Note that the matching process is specific to the type of classifier that will be used and it is not detailed in this paper. For instance, MatchPlanner performs the matching with a decision tree [18] while SMB uses the Boosting meta-classifier [31]. Next, we explain how YAM works with a simple example based on the *hotel booking* web forms.

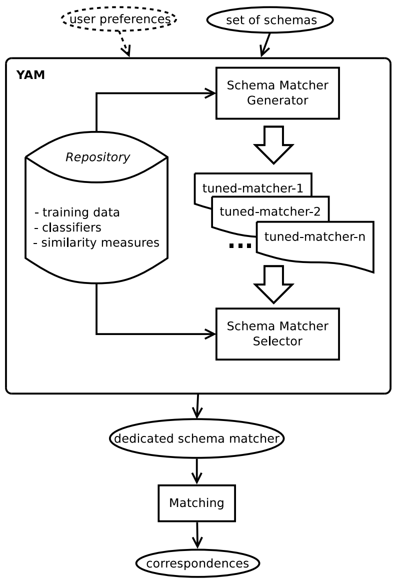


Fig. 3. Architecture of YAM

3.2 Running an example

A user needs to match two schemas for *hotel booking*. The user is not an expert in data integration, and does not have any idea about the appropriate similarity measures or the configuration of the parameters for matching these schemas. By using YAM, the user simply provides the two schemas as input and runs the matching. Since no preferences have been provided, YAM has a default behaviour and it uses random training data from its repository for learning the dedicated matcher. First, YAM generates and tunes one schema matcher for each type of classifier. Among these generated schema matchers, the one with the best results on the training data is elected as the dedicated matcher. Let us imagine that the dedicated matcher is based on a *Bayes Net* type of classifier. YAM can finally use this dedicated matcher to match the schemas of the *hotel booking* scenario (Figure 1). The matching phase depends on the type of classifier of the dedicated matcher (*Bayes Net*). In this example, the dedicated matcher computes the probability of a pair of schema elements being a correspondence for each similarity measure. A global probability is finally derived to indicate whether the pair is a correspondence or not. The set of correspondences resulting from this matching is shown in Figure 2(b). In comparison with the ground truth (Figure 2(a)), we notice that 8 out of the 9 correct correspondences have been discovered. However, two irrelevant correspondences have also been found, namely (*Hotel Location*, *Hotel Name*) and (*Children:*, *Chain*).

In this simple motivating example, we have described the main workflow of YAM. The next section describes the learning process in detail by including the integration of user requirements, which reduces the impact of the random training data.

4 Learning a dedicated matcher

In this section, we describe YAM's approach to learning a dedicated matcher for a given matching scenario. This section is organized as follows: the first part explains the relation between classification and schema matching. Then, we describe the matcher factory, or how YAM generates a matcher for each classifier. Next, we detail how user preferences are integrated during the learning process. Finally, we focus on the strategies which aim at selecting the dedicated matcher among all generated matchers.

4.1 Matching as a machine learning task

The machine learning classification problem consists in predicting the class of an object from a set of its attributes [34]. Thus, any schema matcher can be viewed as a classifier. Each pair of schema elements is considered as a machine learning object where its attributes are the similarity values computed by a set of selected similarity measures of these elements. Given the similarity values of a pair, a matcher labels this pair as either *relevant* or *irrelevant* (i.e., as a correspondence or not). Of course, a matcher may use any algorithm to compute its result – classification, clustering, an aggregation of similarity measures, or any number of ad hoc methods including techniques such as blocking to improve its efficiency. In YAM, we use an extensible library of types of classifiers, among which are decision trees (e.g., *J48*, *NBTree*), aggregation functions (e.g., *SimpleLogistic*), lazy classifiers (e.g., *IBk*, *K**), rule-based classifiers (e.g., *NNge*, *JRip*), voting systems or Bayes Networks. Three assumptions are required to include a type of classifier in YAM: first, the type of classifier should support supervised learning, i.e., all training data have to be labelled. Secondly, it has to use both numerical and categorical features, i.e., the similarity measures may return either numerical values or semantic values (e.g., *synonym*). The last assumption deals with the discretization ability [15, 21], i.e., the type of classifier should be able to split values for continuous features (e.g., similarity measures which return a distance or a value in $[0, 1]$). The generation of a dedicated matcher can be divided into two steps: (i) training of tuned matchers, which can be impacted by parameters and user preferences, and (ii) selection of the dedicated matcher.

Example: Let us consider the pair *(searchform, search)* from our running example. We computed the similarity values of this pair for each similarity measure in our library. For instance, let us assume we have three similarity measures: *AffineGap* [1], *NeedlemanWunsch* [36] and *JaroWinkler* [45]. Processing them over the pair *(searchform, search)* provides the following similarity values: $AffineGap = 14.0$, $NeedlemanWunsch = -4.0$, $JaroWinkler = 0.92$. From these values, a matcher must predict whether the pair is a relevant correspondence or not.

4.2 Training tuned matchers

In this part, we explain how to generate tuned matchers which aim at classifying pairs in a class, either relevant or irrelevant. To reach this goal, classifiers have to be trained. YAM trains each matcher using its repository of training data and potential expert correspondences provided by the user (see Section 4.3). The training data in the repository consists of expert correspondences, i.e., pairs of schema elements with their relevance[6]. The training algorithm is specific to each classifier [34]. However, we shall briefly sum up the main intuition: first, an algorithm selects the similarity measures which provide a maximum of correctly classified correspondences (i.e., a minimal misclassification rate). Then, the similarity measures that might solve harder cases are taken into account.

To illustrate the training process, we have chosen a well-known classifier, the decision tree. In our context, a decision tree is a tree whose internal nodes represent the similarity measures, and the edges stand for conditions applied to the result of the similarity measure (e.g., the similarity value computed by a measure must be greater than a value). All leaf nodes in the tree are the classes, either a *relevant correspondence* or an *irrelevant correspondence*. Algorithm 1

[6] The two schemas of a pair may be necessary to compute similarity values, for instance with structural or contextual measures.

describes the learning of a decision tree in our context. The learning takes as input a set of training data $\mathcal{T}$ and a set $\mathcal{S}$ containing k similarity measures. This training data is defined as a set $\mathcal{T} = \{t\}$, and a single training data $t_i \in \mathcal{T}$ is represented as $t_i = \{(sim_{i1}, v_{i1}), \ldots, (sim_{ik}, v_{ik}), (label_i, class_i)\}$. The output is a decision tree. In the initialization function *buildDecisionTree*, an empty decision tree is created and the recursive function *partition* is called (lines 2 and 3). The goal of this second function is to split the training data into one or more classes, thus creating a new level in the decision tree. To fulfill this goal, the best feature has to be selected for partitioning the training data. Note that similarity measures are continuous features (values in the range $[0, 1]$) and they need to be discretized. This discretization is a well-known problem [15, 21] and it generates cut points, e.g., conditions representing a range of values associated to a class. For each similarity measure, the algorithm produces a set of cut points (lines 6 to 11). Each training data is then associated to one class, i.e., the training data satisfies the condition of a given cut point (lines 12 to 19). As a result, $\mathcal{P}_{sim}^{class}$ contains the training data of the class *class* according to a given feature *sim*. When the partitioning has been performed for all similarity measures, the algorithm is able to select the best partition according to an evaluation metric (line 24). Various evaluation metrics are available, such as information gain, Gini criterion or gain ratio [46], and we rely on misclassification rate in our context. Finally, if the partition produced by the best similarity measure only contains one class[7], then there is no more partitioning of the data and that single class is added as a child node in the tree (line 26). If several classes are present in the partition of the best similarity measure, then the algorithm adds each of these classes as child nodes in the tree, and the function *partition* is recursively called for each class and its training data (lines 28 to 31). Note that Algorithm 1 aims at facilitating the understanding of the learning process, but the building of a classifier is usually improved with heuristics such as pruning [34].

Example: Let us study an example for generating a decision tree with this algorithm. The training data is composed of nine pairs of elements, among which three are relevant, namely *<searchform, search>*, *<city, city>* and *<brand, chain>*. Two similarity measures, *Trigrams* and *Context*, serve as attributes. Figure 4 depicts the generation of the decision tree at the first iteration. A matrix represents the classification performed by a measure, and a pair is either classified as relevant (R) or irrelevant (I). The **white** background colour (respectively **grey**) indicates that a pair is correctly (respectively incorrectly) classified. For instance, the pair *<searchform, city>* has been correctly classified as an irrelevant correspondence by the *Trigrams* measure (Figure 4(a)) while it has been incorrectly classified as a relevant correspondence by the *Context* measure (Figure 4(b)). Note that each classifier is in charge of adjusting the thresholds of each similarity measure to obtain the best classification. Given these matrices, the misclassification rate ϵ is computed for each measure (by counting the number of grey boxes). In this case, *Trigrams* has achieved the lowest misclassi-

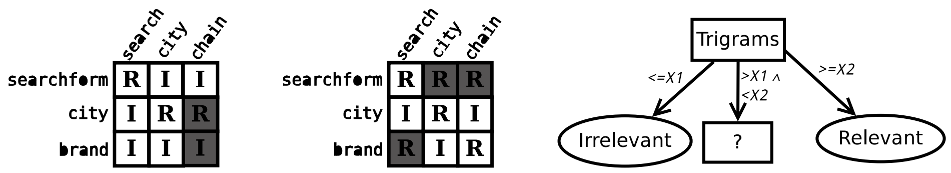
[7] Other stop conditions may be used, for instance "all training data have been correctly classified".

Algorithm 1 Algorithm for building a decision tree

```
Input: set of training data $\mathcal{T}$, set of similarity measures $\mathcal{S}$
Output: a decision tree $\mathcal{D}$
function buildDecisionTree($\mathcal{T}$, $\mathcal{S}$)
$\mathcal{D} \leftarrow \emptyset$
partition($\mathcal{T}$, $\mathcal{S}$, $\mathcal{D}$)
end function

function partition($\mathcal{T}$, $\mathcal{S}$, $parent$)
$\mathcal{P} \leftarrow \emptyset$
for all sim $\in \mathcal{S}$ do
    $\mathcal{P}_{sim} \leftarrow \emptyset$
    $\mathcal{T}_{sim} \leftarrow T$
    $\mathcal{CP} \leftarrow$ discretize($\mathcal{T}_{sim}$, $\mathcal{S}$)
    for all ($condition$, $class$) $\in \mathcal{CP}$ do
        $\mathcal{P}_{sim}^{class} \leftarrow \emptyset$
        for all $t \in \mathcal{T}_{sim}$ do
            if $t \vdash condition$ then
                $\mathcal{P}_{sim}^{class} \leftarrow \mathcal{P}_{sim}^{class} \cup \{t\}$
                $\mathcal{T}_{sim} \leftarrow \mathcal{T}_{sim}$ - $\{t\}$
            end if
        end for
        $\mathcal{P}_{sim} \leftarrow \mathcal{P}_{sim} \cup \{\mathcal{P}_{sim}^{class}\}$
    end for
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{P}_{sim}\}$
end for
$best\_sim$ = findBestClassification($\mathcal{P}$)
if $|\mathcal{P}_{best\_sim}| = 1$ then
    addChild($parent$, $class$)
else
    for all $class \in \mathcal{P}_{best\_sim}$ do
        addChild($parent$, $class$)
        partition($\mathcal{P}_{best\_sim}$, $\mathcal{S}$, $class$)
    end for
end if
end function
```

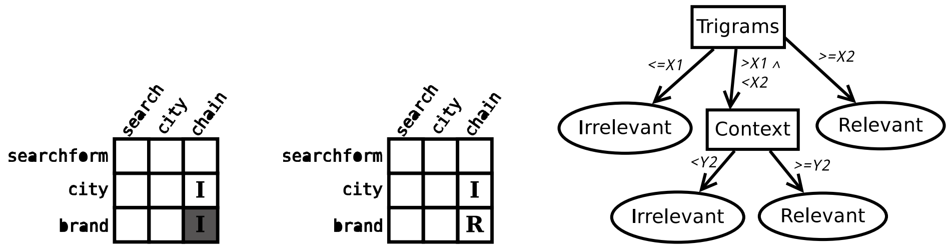
fication rate ($\frac{2}{9}$) and it is therefore selected to be added to the decision tree, as shown in Figure 4(c). The variables X_1 and X_2 stand for the threshold values which enable the achievement of this best classification. At the end of the first iteration, two pairs were not correctly classified by the *Trigrams* measure. Thus, a new iteration begins in order to classify these two pairs with all similarity measures. The matrices for the second iteration are shown in Figure 5(a) and 5(b). Since the training data are now composed of two pairs at the second iteration, the classifier proposes different threshold and parameter values for each similarity measure. This time, the *Context* measure has correctly minimized the



(a) Trigrams: $\epsilon = \frac{2}{9}$ (b) Context: $\epsilon = \frac{3}{9}$ (c) Decision Tree

Fig. 4. Training of a decision tree at first iteration

misclassification rate and it is promoted in the decision tree, as shown in Figure 5(c). Since all the training data have been classified, the algorithm stops.



(a) Trigrams: $\epsilon = \frac{1}{2}$ (b) Context: $\epsilon = \frac{0}{2}$ (c) Decision Tree

Fig. 5. Training of a decision tree at second iteration

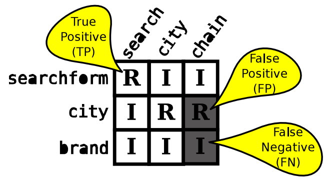
During the training phase, all the thresholds, weights, and other parameters of the matcher (i.e., classifier) are automatically configured, thus providing tuned matchers. Next, we study how user preferences are integrated during this training phase.

4.3 Integrating user preferences

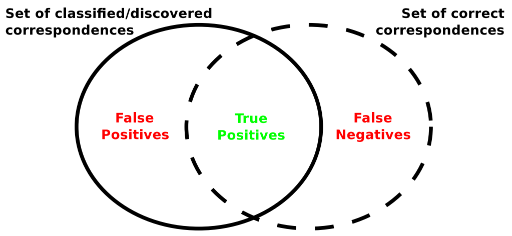
We have identified five options that the user may configure: (i) **preference for precision or recall**, (ii) **expert correspondences**, (iii) **scenarios from the same domain** (iv) **number of training data** and (v) **strategy** to select a dedicated matcher. These options can be combined to improve the matching quality. We should keep in mind that the user has no requirement to provide options, and specifically the training data and the strategy which are automatically selected by YAM when necessary.

Preference for precision or recall The ability to promote either precision or recall is the first attempt to leverage the results of the matching quality. Many applications need such tuning. For instance, matching tools may require training data (usually expert correspondences as in Glue [13]), and YAM could automatically discover a few correct correspondences by generating a high-precision matcher. On the other hand, a typical scenario in which a high recall is necessary is a matching process followed by a manual verification. Since the validation of a discovered correspondence is cheaper in terms of time and resources than the search of a missing correspondence, the discovery of a maximum number of correct correspondences is crucial and implies a high-recall matcher [17].

We have seen that classification algorithms aim at reducing the misclassification rate. As shown in Figure 6, two errors can occur while classifying: (i) an irrelevant correspondence is labeled as correct, i.e., a false positive (FP) or (ii) a relevant correspondence is classified as incorrect, i.e., a false negative (FN). Since precision corresponds to the ratio $\frac{TP}{TP+FP}$, the first error decreases the precision value. Conversely, recall is computed with formula $\frac{TP}{TP+FN}$, thus the second error has a negative impact on recall.



(a) Matrice with TP, FP, FN



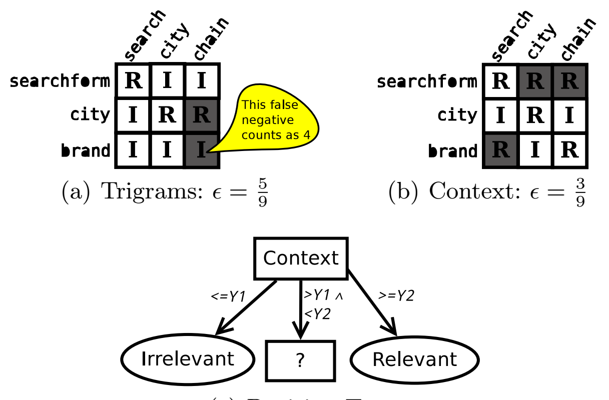
Set of classified/discovered correspondences
Set of correct correspondences
False Positives
True Positives
False Negatives

(b) Relations between TP, FP and FN

Fig. 6. Understanding the impact of FP and FN on precision and recall

To produce tuned matchers which promote either precision or recall, we propose to set a penalty for false positives or false negatives during the learning. This means that false positives (or false negatives) have a stronger impact when computing the misclassification rate. To increase precision (respectively recall) on a given training dataset, we assign a greater penalty to false positives (respectively false negatives). Note that promoting recall (respectively precision) mainly decreases precision (respectively recall).

Example: Back to the first iteration of our example, but let us imagine that we want to promote recall to avoid the misclassification of relevant correspondences (see Figure 7). Therefore, a penalty - equal to 4 in this example - is set for false negatives. Due to the false negative $<$*brand, chain*$>$, the misclassification rate of the measure *Trigrams* drops to $\frac{5}{9}$ and the *Context* measure is selected to be added in the decision tree with its two threshold values Y_1 and Y_2. The three false positives of the *Context* measure should be reclassified in the next iteration. In that way, YAM is able to produce matchers which favour the preference of the user in terms of matching quality.



(a) Trigrams: $\epsilon = \frac{5}{9}$ (b) Context: $\epsilon = \frac{3}{9}$

(c) Decision Tree

Fig. 7. Training of a decision tree at first iteration while promoting recall

Expert correspondences The training data mainly consist of correspondences from the repository. However, the user may decide to provide expert correspondences between the schemas to be matched[8]. The benefit for providing these correspondences is threefold. First, expert correspondences enable a better tuning for some similarity measures. For instance, structural or contextual measures analyse the neighbouring elements in the schema for each element of the correspondence in order to evaluate their similarity. Such measures are given more weight in this case. Another benefit is specific to the constraint-based measures which may discard candidate pairs by relying on the relevance of expert correspondences. Finally, the logic for designing a schema is usually kept throughout the whole process. An expert correspondence between the two schemas reflects the fact that the similarity measure(s) which confirm this correspondence may have captured (a part of) this logic. These measures can be useful for assessing the relevance of other pairs between the same schemas. Section 5.3 includes experiments showing the impact of these expert correspondences on the matching quality.

Example: Let us illustrate this impact with our running example. The consequence of providing the expert correspondence <*searchform, search*> can be explained as follows. The *Context* measure (Figures 4(b) or 7(b)) analyses the similarity of other pairs (e.g., <*searchform, city*>). Since an expert correspondence is established between <*searchform, search*>, the context of the *searchform* element is modified, and it can imply that the pair <*searchform, city*> is now classified as irrelevant. This means that the *Context* measure can have a lower misclassification rate and thus be promoted as the measure to be added to the decision tree at the first iteration.

[8] If the user has not provided a sufficient number of correspondences, YAM will extract others from the repository.

Scenarios from the same domain Similarly to expert correspondences, providing scenarios from the same domain as the schemas to be matched (e.g., the *hotel booking* domain) produces better tuned matchers. The main reason is the vocabulary which is shared across the domain, thus promoting the relevant similarity measures to detect correspondences . The exploitation of external resources (e.g., domain ontology, thesaurus) for schema matching has already been studied [16, 47]. However, these external resources differ from our same-domain scenarios: when using external resources, a matching has to be performed between the schemas and the external resource. On the other hand, our same-domain scenarios are used in the same fashion as other training data in order to generate a tuned schema matcher. Besides, our approach is able to exploit external resources when they are available with a similarity measure. For instance, YAM already includes the Resnik similarity measure, which exploits Wordnet [40].

Example: Let us imagine that three pairs of schemas respectively include the following correct correspondences <*booking-form, search-form*>, <*form, form*> and <*searchform, form*> and that we use them as training data. Then, the classifier selects similarity measures which facilitate the correct classification of the pair <*searchform, search*>.

Number of training data The number of training data clearly has an impact on generating the tuned matchers, both in terms of matching quality and time performance. The common rule is that a classifier performs better when trained with more data. Yet, there are threshold values above which a classifier can provide stable results. To determine these threshold values for generating robust tuned matchers, we have conducted extensive experiments described in Section 5.3. Thus, the number of training data is automatically set by YAM, but the user can tune this value.

Strategy This last preference can be provided by the user to select the dedicated matcher. The available strategies are detailed in the next part.

4.4 Selecting a dedicated matcher

The final step of the learning process deals with the selection of the dedicated matcher among all tuned matchers which have been generated by YAM for a given schema matching scenario. This selection depends on the adopted strategy, which can be user-defined or automatically configured by YAM. The strategy is defined as a combination of one value for each of the three following criteria:

– **Choice of the quality measure**: *precision*, *recall* or *F-measure*
– **Choice of the validation dataset**: the *repository*, the *set of expert correspondences* or the *similar scenarios* (when provided)
– **Pool of matchers**: *generate* (select the dedicated matcher among the tuned matchers generated in Section 4.2), *reuse* (select the dedicated matcher among those stored in the repository), *both* (select the dedicated matcher among the repository and the tuned matchers).

If the user has not manually set up a strategy, YAM applies the following algorithm to configure the strategy:

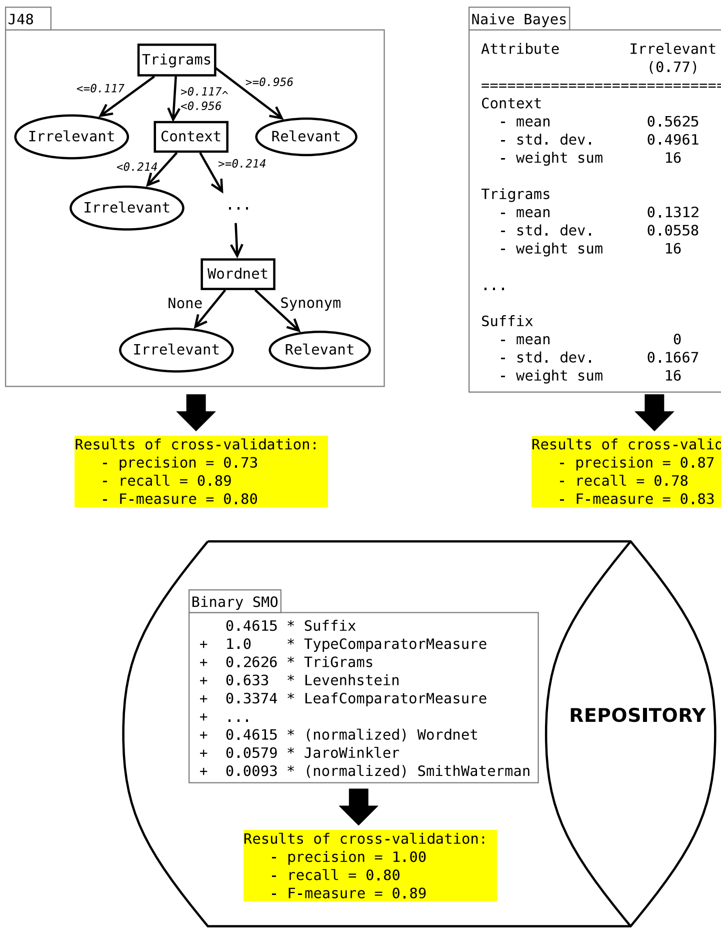
– If a preference for *precision* or *recall* is set, this preference is the quality measure. Otherwise, the default measure is the *F-measure*.
– If a *set of expert correspondences* or a *set of similar schemas* is provided, this set becomes the validation dataset. Otherwise, YAM uses (a subset of) the scenarios from the *repository*.
– The *generated* matchers form the pool of matchers. The *reuse* and *both* options need to be provided by the user and they mainly aim at speeding up the execution process by avoiding the learning of new tuned matchers.

Once the criteria of the strategy have been fixed, each matcher (from the pool of matchers) computes their accuracy (depending on the selected measure) by performing a cross-validation process against the validation dataset. The matcher which obtains the best value is elected as the dedicated matcher. Note that the dedicated matcher is finally stored in the repository, thus allowing it to be reused for further experiments.

Example: To illustrate the impact of the strategy, let us comment Figure 8. YAM has generated two tuned matchers (*J48* and *Naive Bayes*) and one matcher from a previous generation is stored in the repository (*Binary SMO*). Each matcher has its own technique for combining the different similarity measures (reflected by thresholds, weights or intern metrics such as standard deviation). To simplify the example, the training data consist only of examples stored in the repository. The boxes below each matcher indicate the results of cross-validation achieved by the matcher over the training data. If the user has not provided any strategy or preference, YAM automatically selects the matcher among those *generated* with the *best F-measure* value (default strategy equal to <*F-measure, repository, generated*>). In this case, the dedicated matcher will be *Naive Bayes* (83% F-measure) to the detriment of *J48*. On the contrary, if the user has set a preference for recall, the strategy is defined as <*recall, repository, generated*>. Since *J48* obtains a 89% recall, it is selected as the dedicated matcher. Finally, if the strategy is <*precision, repository, both*>, this means that the user needs a matcher with the best precision value among all matchers both generated and stored in the repository. In this context, *Binary SMO* achieves the best precision value (100%) and it will be elected as the dedicated matcher.

5 Experiments

This section begins with a description of the protocol. Next, we firstly demonstrate the need for a matcher factory (self-tuning feature). Then we study the integration of user preferences (described in Section 4.3.), and their impact on the matching quality. Finally, we compare our results with two matching tools that have excellent matching quality, COMA++ [2] and Similarity Flooding (SF) [32]. These tools are described in more detail in Section 6.



```
Results of cross-validation:
   - precision = 0.73
   - recall = 0.89
   - F-measure = 0.80
```

```
Naive Bayes
 Attribute        Irrelevant    Relevant
                    (0.77)       (0.23)
 ======================================
 Context
   - mean           0.5625           1
   - std. dev.      0.4961      0.1667
   - weight sum        16            4

 Trigrams
   - mean           0.1312      0.1969
   - std. dev.      0.0558      0.1652
   - weight sum        16            4

 ...

 Suffix
   - mean              0         0.25
   - std. dev.      0.1667       0.433
   - weight sum        16            4
```

```
Results of cross-validation:
   - precision = 0.87
   - recall = 0.78
   - F-measure = 0.83
```

```
Binary SMO
    0.4615 * Suffix
 +  1.0    * TypeComparatorMeasure
 +  0.2626 * TriGrams
 +  0.633  * Levenhstein
 +  0.3374 * LeafComparatorMeasure
 +  ...
 +  0.4615 * (normalized) Wordnet
 +  0.0579 * JaroWinkler
 +  0.0093 * (normalized) SmithWaterman
```

```
Results of cross-validation:
   - precision = 1.00
   - recall = 0.80
   - F-measure = 0.89
```

Fig. 8. Results of cross-validation for different matchers

5.1 Experimental protocol

Experiments were run on a 3.6 GHz computer with 4 Go RAM under Ubuntu 11.10.

Configurations of the tools The default configuration for SF was used in the experiments. We tested the three pre-configured strategies of COMA++ (*AllContext*, *FilteredContext* and *Fragment-based* in the version *2005b*) and we kept the best score among the three.

The current version of YAM is implemented in Java 1.6. Our tool includes 20 classifiers from the Weka library [23] and 30 similarity measures, including all terminological measures [8] from the Second String project[9], a contextual

[9] Second String (November 2015)

measure named Approxivect [19], the Resnik semantic similarity measure [40] and a simple structural measure that compares the constraints and data types, as described in Similarity Flooding [32]. YAM's repository contains a large set of 200 schema matching scenarios from various domains.

Dataset The dataset used in these experiments is composed of more than 200 schema matching scenarios, covering the following domains:

- **University department** describes the organization of university departments [20]. These two small schemas have very heterogeneous labels.
- **Thalia courses**. These 40 schemas have been taken from Thalia collection [24] and they are widely used in literature [14, 19]. Each schema has about 20 elements and they describe the courses offered by some worldwide universities. As explained in [44], this dataset could refer to a scenario where users need to generate an exchange schema between various data sources.
- **Travel** includes 5 schemas that have been extracted from airfare web forms [37]. In data sharing systems, partners have to choose a schema or a subset of schema that will be used as a basis for exchanging information. This *travel* dataset clearly reflects this need, since schema matching enables data sharing partners to identify similar concepts that they are willing to share.
- **Currency** and **sms** datasets are popular web services[10]. Matching the schemas extracted from web services is a recent challenge to build new applications such as mashups or to automatically compose web services.
- **Web forms** are a set of 176 schemas, extracted from various websites by the authors of [31]. They are related to different domains, from hotel booking and car renting to dating and betting. For instance, the *finance* domain contains more than ten schemas of small size. Authors of [44] state that schema matching is often a process which evaluates the costs (in terms of resources and money) of a project, thus indicating its feasibility. These scenarios can be a basis for project planning, i.e., to help users decide if integrating their data sources is worth or not.

Table 1 summarizes the features of the schema matching scenarios. The *size* column indicates the average number of schema elements in the scenario. The *structure* column checks how deep the schema elements are nested. We consider a schema to be flat when it includes at most three levels, and a schema is said to be nested with at least four levels. The last column provides information about the number of schemas in the scenario.

For all these scenarios, the expert correspondences are available, either manually or semi-automatically designed. We use these 200 scenarios, and their correct correspondences, both to train YAM and to demonstrate the effectiveness of the three matching tools.

[10] Free Web Services (November 2015)

	Average size		Structure		Number of
	Small (<10)	Average (10-100)	Flat (≤3)	Nested (>3)	schemas
Univ. dept	×		×		2
Thalia courses	×	×	×		40
Travel	×		×		5
Currency		×		×	2
Sms		×		×	2
Web forms	×	×	×		176

Table 1. Schema matching scenarios according to their properties

Quality metrics To evaluate the matching quality, we use common metrics in the literature, namely precision, recall and F-measure [3, 17, 20]. Precision calculates the proportion of relevant correspondences extracted among those discovered. Another typical metric is recall which computes the proportion of relevant discovered correspondences between all relevant ones. F-measure is a trade-off between precision and recall.

5.2 Self-tuning feature

We begin with a study of the self-tuning feature, i.e., the ability to select the most effective schema matcher. More specifically, we justify the need for a schema matcher factory, since our approach can adapt the method for combining similarity measures to the scenario. In other words, if a traditional schema matching tool (e.g., COMA++) performs matching for these 200 scenarios, the same method for combining similarity measures would be used (i.e., an aggregation function for COMA++). With YAM, we demonstrate that from one scenario to another, the optimal method is different (i.e., the dedicated schema matcher generated with YAM is based on different types of classifier).

Let us describe the experiment. We ran YAM against 200 scenarios, and we measured two criteria: the number of times (out of 200) that a type of classifier was selected as the dedicated matcher (Figure 9(a)) and the average F-measure achieved by a type of classifier over the 200 scenarios (Figure 9(b)). For instance, the type of classifier *VFI* was selected as a dedicated matcher 57 times (out of 200). This type of classifier *VFI* achieves over the 200 scenarios an average F-measure equal to 59%. For this evaluation, we included no user preference, so all matchers were trained only with the repository (20 random schema matching scenarios) and the dedicated matcher was selected with the default strategy. This process took roughly 1400 seconds to produce the dedicated matcher for each given scenario. The plots are limited to the to the 14 best types of classifiers.

The first comment for Figure 9(a) is the diversity of types of classifier which have been selected. There is not one best schema matcher for matching the 200 scenarios, but more than fourteen. This means that a matcher factory, such as YAM, is necessary to cope with the differences in the schema matching scenarios. Secondly, we note that 2 types of classifier, namely *VFI* (Voting Feature Intervals) and *Bay* (Bayes networks), are selected in half of the 200 scenarios. The matchers based on these types of classifiers can be considered as robust because they provide acceptable results in most scenarios in our repository. This trend is confirmed with the second plot (Figure 9(b)) on which *VFI* and *Bayes Net* achieve the best average F-measure values over the 200 scenarios. Another comment on these plots deals with the aggregation functions, represented by *SLog* (Simple Logistic) and *MLP*. These functions, which are commonly used by traditional matching tools, are selected as dedicated matchers in only a few scenarios. Thus, they do not provide optimal matching quality results in most schema matching scenarios. Finally, a good ranking in terms of F-measure does not guarantee that the type of classifier will be selected many times. For instance, the decision trees *J48* and its alternative *J48graft* obtain an average 40% F-measure but they are selected as dedicated matchers only a few times. Conversely, the types of classifiers *CR* (Conjunction Rules) and *ADT* (Alternating Decision Tree), which achieve 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 very effective, in terms of matching quality, for solving specific scenarios. Thus, these results support our claim that a matcher factory such as YAM is a promising perspective.

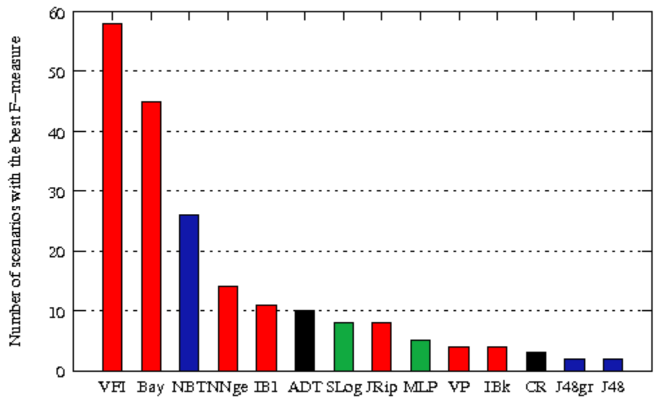
5.3 Impact of the integration of user preferences

In this part, we analyse the impact of three user preferences, which have been described in Section 4.3: the number of training data, the preference between precision or recall and the providing of expert correspondences. Note that for these experiments, we only keep the 5 most robust classifiers (see Section 5.2), namely *VFI*, *BayesNet*, *NBTree*, *NNge* and *IB1*.

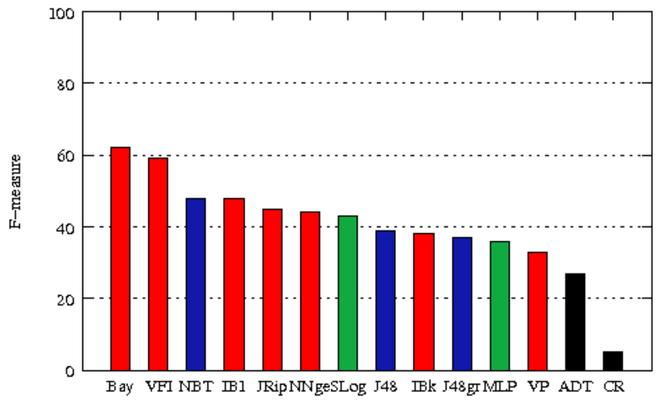
Number of training data In this experiment, our goal is to show that the amount of training data needed to produce a high performing matcher is not onerous and that the number of training data can be automatically chosen (when the user does not provide this input). Figure 10 depicts the average F-measure[11] of five 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 types of classifiers (*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[12], 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 numerous training examples are supplied. *NBTree* and *NNge* also increase their average F-measure from around 10% as training data is increased. On the contrary, *BayesNet* achieves the same F-measure (60% to 65%)

[11] Only the F-measure plot is provided since the plots for precision and recall follow the same trend as the F-measure.

[12] 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.



(a) Number of selections as dedicated matcher



(b) Average F-measure by type of classifier

Fig. 9. Effectiveness by type of classifier

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.

Remember that YAM automatically chooses the number of training scenario according to the matchers that have to be learned. To select this number of training scenarios, we conducted an extensive series of experiments. More than $11,500$ experiments resulted from the runs, and we use them to deduce the number of training scenarios for a given classifier. Table 2 shows the conclusion of our empirical analysis. For instance, when learning a schema matcher based on the *J48* classifier, YAM ideally chooses a number of training scenarios between 20 and 30.

In a machine learning approach, it is crucial to analyse the relationship between performance and the size of training data. Therefore, we evaluate the performance of YAM according to the size of the training data. We have aver-

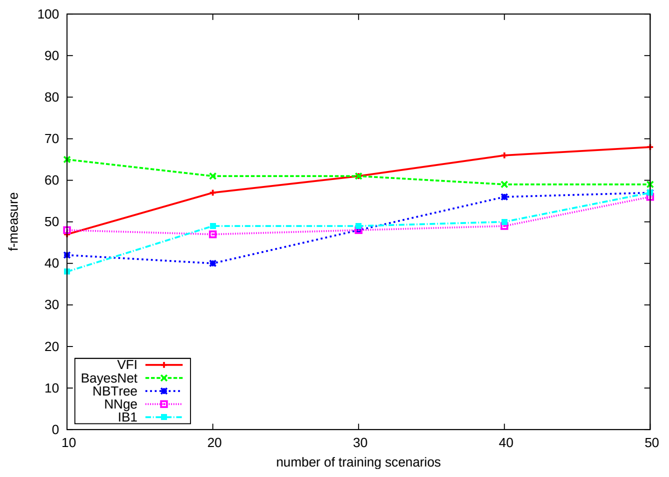


Fig. 10. Average F-measure when varying the number of training scenarios

Table 2. Number of training scenarios for each type of classifier

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

aged the training and matching times for 2000 runs (10 runs for each of the 200 scenarios) according to different number of training data (from 5 to 50). Table 3 summarizes the results for training 20 classifiers (i.e., 20 tuned matchers), selecting the dedicated matcher, and performing the matching with the dedicated matcher. This experiment is independent from the empirical results shown in Table 2, i.e., 20 classifiers were generated with 5 training data, 20 classifiers were generated with 10 training data, etc. This means that the training time (e.g., 165 seconds for 5 training data) corresponds to the training of 20 classifiers. Obviously some types of classifier are quicker than others to generate a matcher, but our main motivation is the selection of the best tuned matcher among a large panel rather than an efficient generating process. The training step is time-consuming but this is a fair time for learning 20 tuned matchers. The training time seems constant according to the number of training scenarios. The matching time is not significant (between 13 seconds and up to 128 seconds). We note that the matching time slightly decreases to 110 seconds with 50 training scenarios. We believe this is due to the type of classifier which is used: as shown in Table 2, the types of classifiers which are selected with 50 and more training scenarios are mainly instance-based classifiers such as *IB1*, *VFI* or *IBk*. In our context, the matching with such classifiers seems more efficient. It should be remembered that the training is carried out with 20 classifiers and 30 similarity measures. If required, these numbers can be reduced to improve performance, for instance based on the empirical results from Table 2. Still, an automatic matching performed in around one hour is an advantage compared to a manual matching. Besides, the current strategy for selecting the dedicated matcher is only based on (matching) quality criteria. But we could also take into account the training time for each type of classifier within the strategy.

Table 3. Average times according to the number of training scenarios for both training 20 tuned matchers and matching with the dedicated matcher

Number of training scenarios	5	10	20	30	40	50
Time for training (in seconds)	165	601	2227	3397	5182	6506
Time for matching (in seconds)	13	24	47	124	128	110
Total time (in seconds)	178	625	2274	3521	5310	6616

Precision vs. recall preference We now present another interesting feature of our tool, the possibility of choosing between promoting recall or precision, by tuning the weight for false positives or false negatives. Schema matching tools usually favour a better precision, but we demonstrate that YAM tuned with a preference for recall effectively allows to obtain a better recall, with no significant impact on F-measure. In other words, the gain in terms of recall is proportionally equivalent to the loss in terms of precision, thus the F-measure is roughly constant. Figures 11(a) and 11(b) respectively depict 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 as to precision.

For 2 matchers (*NBTree* and *NNge*), the recall increases up to 20% when we tune in favour 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 the matching process is being performed. 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 irrelevant discovered correspondences. But a 20% increase of recall means that (s)he has 20% fewer correspondences to search through among $10,000$ possible pairs ! Hence, this tuning could have a highly significant effect on the usability of the matcher for certain tasks. Indeed, we highlight the fact that matching tools may be disregarded because the amount of work during pre-match effort (tuning the tool) and the amount of work during post-match effort (manual verification of the discovered correspondences) is sometimes not worthwhile compared to the benefit of the tool, especially if the user cannot leverage the results towards more precision or recall.

For the three other matchers (*BayesNet*, *VFI* and *IB1*), tuning in favour of recall has no significant effect. This does not mean that only a few types of classifiers can promote recall. Without any tuning, only one matcher (*BayesNet*) has an average recall superior to its precision. Indeed, most of the matchers in our library promote by default precision. However, when setting a weight for false negatives to 2, then four matchers from the library have a higher recall than precision. And with a weight for false negatives equal to 3, five other matchers 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 [17].

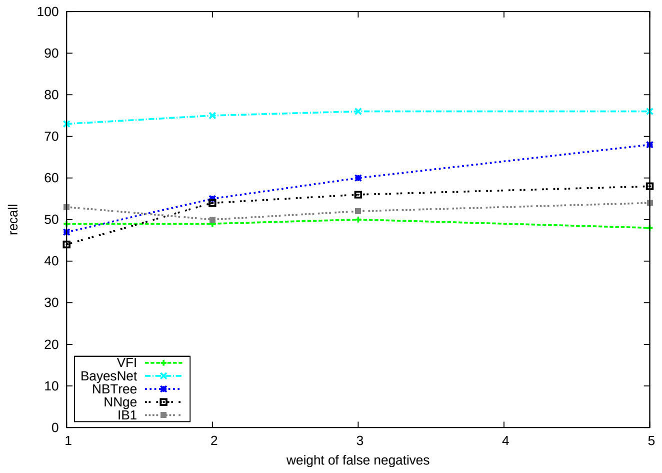
Impact of expert correspondences As in Glue [13], the number of expert correspondences is an input - compulsory for Glue, but optional for YAM - to the system. YAM can use these expert correspondences to learn more appropriate matchers. In this study, we measured the gain in terms of matching quality when a user provides these correspondences. The training phase used 20 scenarios and expert correspondences were randomly selected. We report the size of the sets of expert correspondences in percentages, given that 5% of expert correspondences usually means that we only provide 1 or 2 correspondences as input.

Figure 12 depicts the average F-measure for 40 random scenarios for the five robust matchers. 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 matchers still increases 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 tedious effort by the user[13]. Yet, this input can improve the matching quality of most matchers, even with a small amount of expert correspondences. Besides, YAM closely integrates these expert correspondences in generating a better matcher, while other tools such as Glue mainly use these correspondences as a bootstrap.

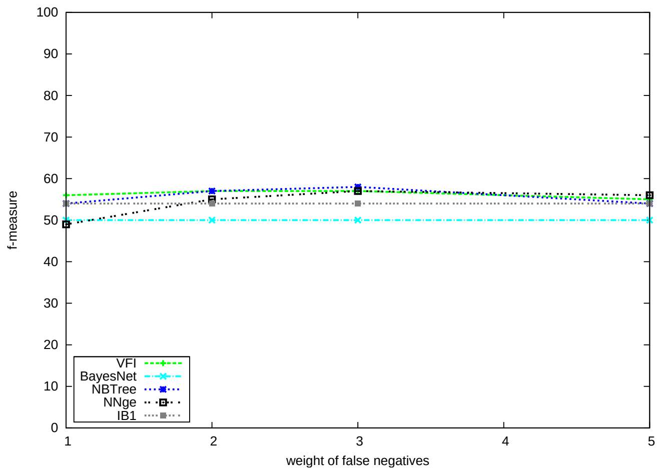
5.4 Comparing with other matching tools

In this last experiment, we compare YAM with two matching tools known to provide a good matching quality: COMA++ and Similarity Flooding (SF). COMA++ [2] uses 17 similarity measures to build a matrix between pairs of elements and aggregate their similarity values. Similarity Flooding [32] 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

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



(a) Recall



(b) F-measure

Fig. 11. Matching quality of robust matchers when promoting recall

detail in Section 6. YAM, our factory of schema matchers, uses the default strategy (<*F-measure, repository, generated*>) to produce the dedicated matcher. The number of training data is automatically adjusted according to the classifier which is going to be trained (using Table 2).

Figure 13(a) and 13(b) depict the F-measure obtained by YAM, COMA++ and Similarity Flooding on 10 schema matching 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*). Besides, COMA++ is the

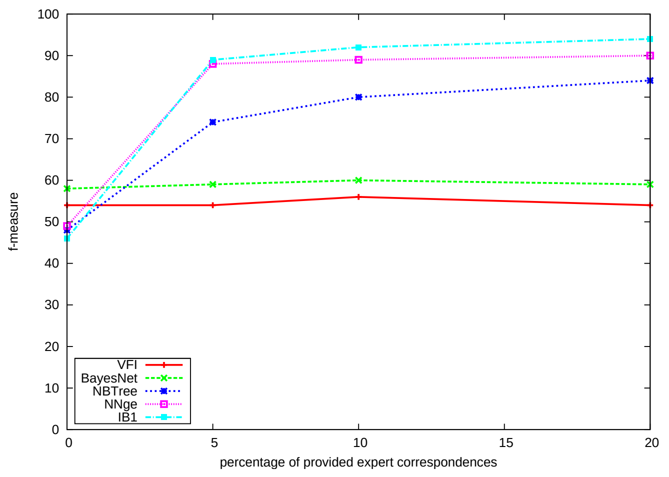


Fig. 12. F-measure of robust matchers when increasing the number of input expert correspondences

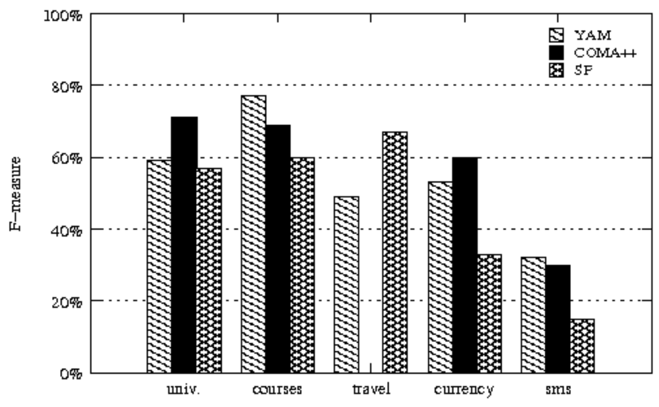
only tool which does not discover any correspondence for one scenario (*travel*). However, we notice that YAM obtains better results in the web forms scenarios since it was mainly trained with web forms (stored in the repository). With non-web forms scenarios, YAM is still competitive with the other tools.

We have summarized the results of this comparison in Table 4. The numbers in this table represent an average for the 10 scenarios in terms of precision, recall and F-measure. YAM obtains the highest average F-measure (71%) while COMA++ and SF achieve an average F-measure around 50%. In addition, in the bottom part of the table we present the matching quality for YAM with user preferences. We note that the promotion of recall is effective (78% instead of 65%) but to the detriment of precision. When YAM is trained with scenarios from the same domain, the quality of results slightly improves (F-measure from 71% to 76%). The most significant increase in quality is due to the integration of expert correspondences during training, which enables F-measure to reach 89%.

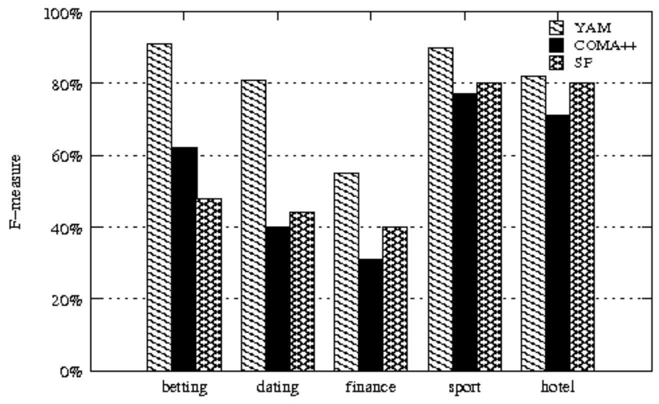
These experiments show how our matcher factory relies on the diversity of classifiers. Indeed, the dedicated matchers that it has generated for these scenarios are based on various classifiers (*VFI*, *BayesNet*, *J48*, etc.) while COMA++ and SF only rely on respectively an aggregation function and a single graph propagation algorithm. Besides, YAM is able to integrate user preferences to produce more efficient dedicated matchers and to improve the matching quality.

6 Related work

Much work has been done both in schema matching and ontology alignment. One can refer to the following books and surveys [3, 6, 20, 22, 42] for more details



(a) Non-web forms scenarios



(b) Web form scenarios

Fig. 13. Precision, recall and F-measure achieved by the three matching tools on 10 scenarios

about schema and ontology matchers. All related approaches aims at performing matching (or alignment). On the contrary, YAM is a matcher factory, which produces a schema matcher. There is no equivalent approach to our generator of schema matchers. In this section, we have chosen to present an overview of the last decade of research in schema and ontology matching, which has served as a basis to our work. Still, a deeper comparison between traditional matching tools and our factory of matchers is difficult due to the nature of the tools.

Harmony schema matcher [35, 44] combines multiple matching algorithms by using a vote merger. The vote merging principle is a weighted average of the match scores provided by each match voter. A match voter provides a confidence score for each pair of schema elements to be matched. Then, the Similarity Flooding strategy [32] is applied to adjust the confidence scores based on structural information. Thus, positive confidence scores propagate throughout the graph.

Table 4. Average matching quality of the tools: COMA++, SF and YAM (single and tuned with 3 parameters)

	Precision	**Recall**	**F-measure**
COMA++	66%	38%	48%
SF	61%	43%	50%
YAM	81%	65%	71%
YAM-recall	68%	**78%**	73%
YAM-domain-specific-scenarios	80%	72%	**76%**
YAM-expert-correspondences (5%)	88%	90%	**89%**

An interesting feature of Harmony lies in its graphical user interface for viewing and modifying the discovered schema correspondences through filters.

RiMOM [28] is a multiple strategy dynamic ontology matching system. Different matching strategies are applied to a specific type of ontology information. Based on the features of the ontologies to be matched, RiMOM selects the best strategy (or strategy combination) to apply. When loading the ontologies, the tool also computes three feature factors. The underlying idea is that if two ontologies share similar feature factors, then the strategies that use these factors should be given a high weight when computing similarity values. For instance, if the *label meaningful* factor is low, then the *Wordnet-based strategy* will not be used. Each strategy produces a set of correspondences, and all sets are finally aggregated using a linear interpolation method. A last strategy dealing with ontology structure is finally performed to confirm discovered correspondences and to deduce new ones. Contrary to other approaches, RiMOM does not rely on machine learning techniques to select the best strategy. It is quite similar to the *AHP* work (described hereafter) in selecting an appropriate matcher based on the input's features.

AgreementMaker [9] provides a combination strategy based on the linear interpolation of the similarity values. The weights can be either user assigned or evaluated through automatically-determined quality measures. The system allows for serial and parallel composition where, respectively, the output of one or more methods can be used as input to another one, or several methods can be used on the same input and then combined. The originality of AgreementMaker is the capability of manually tuning the quality of matches. Indeed, this tool includes a comprehensive user interface supporting both advanced visualization techniques and a control panel that drives the matching methods.

In [31], the authors propose a machine learning approach, SMB. It uses 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. Thus, SMB automatically selects a pair of similarity measures as a matcher by focusing on harder training data. A specific feature of this algorithm is the important weight given to misclassified pairs during training. Although this approach makes use of several similarity measures, it mainly combines a similarity measure (first line matcher) with a decision maker (second line matcher). Empirical results show that the selection of a pair does not depend on their individual performance. Thus, only relying on one classifier is risky.

In a broader way, the STEM framework [26] identifies the most interesting training data set which is then used to combine matching strategies and tune several parameters such as thresholds. First, training data are generated, either manually (i.e., an expert labels the entity pairs) or automatically (at random, using static-active selection or active learning). Then, similarity values are computed using pairs in the training data set to build a similarity matrix between each pair and each similarity measure. Finally, the matching strategy is deduced from this matrix thanks to supervised learned algorithm. The output is a tuned matching strategy (how to combine similarity measures and tune their parameters). The framework enables a comparative study of various similarity measures (e.g., Trigrams, Jaccard) combined with different strategies (e.g., decision tree, linear regression) whose parameters are either manually or automatically tuned.

The MatchPlanner approach [18] makes use of decision trees to select the most appropriate similarity measures. This approach provides acceptable results with regard to 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 [27] 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 to the set of correspondences until an optimal parameters configuration of the matching tool is found. eTuner strongly depends on the capabilities of the matching tool, and it has to be integrated in an existing matching tool by a programmer. Conversely, YAM learns a dedicated matcher according to a given matching scenario. It is also able to integrate important features like user preference between recall and precision. Contrary to eTuner, YAM is extensible in terms of similarity measures and classifiers, thus enhancing the capabilities of our tool.

Authors of [30] have proposed to select a relevant and suitable matcher for ontology matching. They have used Analytic Hierarchical Process (AHP) to fulfill this goal. They first define characteristics of the matching process divided into six categories (inputs, approach, usage, output, documentation and costs). Users then fill in a requirements questionnaire to set priorities for each defined characteristic. Finally, AHP is applied with these priorities and it outputs the most suitable matcher according to user requirements.

COMA/COMA++ [2,11] is a hybrid matching tool that incorporates many independent similarity measures. It can process Relational, XML, RDF schemas as well as ontologies. Internally it converts the input schemas as trees for structural matching. It provides a library of 17 element-level similarity measures. For linguistic matching it utilizes a user defined synonym and abbreviation tables, along with n-gram name matchers. Similarity values between each possible pair of elements and for each similarity measure are stored in a similarity matrix. Next, the combination of the values is performed using aggregation operators such as *max*, *min*, *average*. Different strategies, e.g., reuse-oriented matching or fragment-based matching, can be included, offering different results. For each source element, pairs with a combined similarity value higher than a threshold are displayed to the user for validation. COMA++ supports a number of other features like merging, saving and aggregating match results of two schemas.

Similarity Flooding (SF) and its successor Rondo [32, 33] can be used with Relational, RDF and XML schemas. These input data sources are initially converted into labelled graphs and SF approach uses fix-point computation to determine correspondences between graph nodes. The algorithm has been implemented as a hybrid matcher, in combination with a terminological similarity measure. First, the prototype does an initial element-level terminological matching, and then feeds the computed candidate correspondences to the structural similarity measure for the propagation process. This structural measure includes a few rules, for instance one of them states that two nodes from different schemas are considered similar if their adjacent neighbours are similar. When similar elements are discovered, their similarity increases and it impacts adjacent elements by propagation. This process runs until there is no longer similarity increasing. Like most schema matchers, SF generates correspondences for pairs of elements having a similarity value above a certain threshold. The generation of an integrated schema is performed using Rondo's *merge* operator. Given two schemas and their correspondences, SF converts the schemas into graphs and it renames elements involved in a correspondence according to the priorities provided by the users.

Glue [13], and its predecessor LSD [12], are also based on machine learning techniques. They have four different learners, which exploit different information from the instances. The name learner (Whirl, a nearest-neighbour 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 to the instances associated to each schema element. A Naive Bayes classifier considers labels and attributes as a set of tokens for performing 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.

AUTOMATCH [5] is the predecessor of AUTOPLEX [4], which uses schema instance data and machine learning techniques to find possible correspondences between two schemas. An 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 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.

The main difference between YAM and all these matchers lies in the level of abstraction. Theoretically, YAM could generate most of these matching tools. This is actually the case with MatchPlanner [18]. The most relevant existing approach to YAM is the configuration tool eTuner, since both approaches discover the best configuration of a matcher. Yet, eTuner's capabilities are limited compared to YAM: it has to be plugged into an existing matching tool (which requires programming skills) and it totally depends on that matching tool, especially for the method which combines similarity measures. Thus, it does not offer the extensibility and self-tuning features encompassed in YAM.

7 Conclusion

In this paper, we have presented YAM, the first extensible and self-tuning factory of schema matchers. Instead of producing correspondences between schemas, YAM generates a dedicated schema matcher for a given matching scenario. This is made possible by formalizing the matching problem as a classification problem. In addition, we described how to integrate user requirements into the generation process so that the dedicated matcher fulfills the needs and preferences of the user. Our approach is also the first work to let users choose the promotion of either precision or recall. Experiments have shown that the dedicated matchers generated with YAM obtain acceptable quality results with regard to reputed matching tools. Finally, we outline here the lessons learned:

– We have demonstrated a strong need for a schema matcher factory;
– Our experiments support the idea that machine learning classifiers are suitable for the matching task and that the traditional aggregation functions are not always the most efficient method for combining similarity measures;
– We have studied the impact and the benefits on the matching quality when the user provides preferences such as the promotion of recall/precision or input expert correspondences.

In the future, we first plan to test further classifiers. Indeed, there exist a large number of machine learning classifiers of which we have experimented only a subset. Among them, the meta-classifiers base their predictions using the results of several classifiers and therefore offer the possibilities for improving matching quality. In a similar fashion, we foresee the possibility to deduce some correspondences between the matching results of all matchers. These highly probable correspondences could serve as input expert correspondences to produce a smart dedicated matcher. Finally semi-supervised learning [7] could be used to improve the accuracy of the dedicated matcher: the intuition is to include in the training data some unlabelled pairs from the schemas to be matched.

References

1. S. F. Altschul and B. W. Erickson. Optimal sequence alignment using affine gap costs. *Bull Math Biol*, 48(5-6):603–616, 1986.
2. David Aumueller, Hong Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with coma++. In *SIGMOD*, pages 906–908, 2005.
3. Zohra Bellahsene, Angela Bonifati, and Erhard Rahm, editors. *Schema Matching and Mapping*. Springer, 2011.
4. Jacob Berlin and A Motro. Automated discovery of contents for virtual databases. In *CoopIS*, pages 108–122, 2001.
5. Jacob Berlin and A Motro. Database schema matching using machine learning with feature selection. In *CAiSE*, 2002.
6. Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011.
7. O. Chapelle, B. Schölkopf, and A. Zien, editors. *Semi-Supervised Learning*. MIT Press, Cambridge, MA, 2006.
8. W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string distance metrics for name-matching tasks. In *In Proceedings of the IJCAI-2003*, 2003.
9. Isabel F. Cruz, Flavio Palandri Antonelli, and Cosmin Stroe. AgreementMaker: efficient matching for large real-world schemas and ontologies. *PVLDB*, 2(2):1586–1589, 2009.
10. Warith Eddine Djeddi and Mohamed Tarek Khadir. Ontology alignment using artificial neural network for large-scale ontologies. *Int. J. Metadata Semant. Ontologies*, 8(1):75–92, May 2013.
11. Hong Hai Do and Erhard Rahm. Coma - a system for flexible combination of schema matching approaches. In *VLDB*, pages 610–621, 2002.
12. AnHai Doan, Pedro Domingos, and Alon Y. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. In *SIGMOD*, pages 509–520, 2001.
13. AnHai Doan, Jayant Madhavan, Robin Dhamankar, Pedro Domingos, and Alon Y. Halevy. Learning to match ontologies on the semantic web. *VLDB J.*, 12(4):303–319, 2003.
14. Anhai Doan, Jayant Madhavan, Pedro Domingos, and Alon Halevy. Ontology matching: A machine learning approach. In *Handbook on Ontologies in Information Systems*, pages 397–416. Springer, 2004.
15. James Dougherty, Ron Kohavi, Mehran Sahami, et al. Supervised and unsupervised discretization of continuous features. In *Machine learning: proceedings of the twelfth international conference*, volume 12, pages 194–202, 1995.
16. Eduard Dragut and Ramon Lawrence. Composing mappings between schemas using a reference ontology. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, pages 783–800. Springer, 2004.
17. Fabien Duchateau and Zohra Bellahsene. Designing a benchmark for the assessment of schema matching tools. In *Open Journal of Databases (OJDB)*, volume 1, pages 3–25. RonPub, Germany, 2014.
18. Fabien Duchateau, Zohra Bellahsene, and Remi Coletta. A flexible approach for planning schema matching algorithms. In *OTM Conferences (1), CooPerative Information Systems (CooPIS)*, pages 249–264, 2008.
19. Fabien Duchateau, Zohra Bellahsene, and Mathieu Roche. A context-based measure for discovering approximate semantic matching between schema elements. In *Research Challenges in Information Science (RCIS)*, 2007.

20. Jérôme Euzenat and Pavel Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007.
21. Usama M Fayyad and Keki B Irani. On the handling of continuous-valued attributes in decision tree generation. *Machine learning*, 8(1):87–102, 1992.
22. Avigdor Gal. *Uncertain Schema Matching*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.
23. Stephen R. Garner. Weka: The waikato environment for knowledge analysis. In *Proc. of the New Zealand Computer Science Research Students Conference*, pages 57–64, 1995.
24. Joachim Hammer, Mike Stonebraker, , and Oguzhan Topsakal. Thalia: Test harness for the assessment of legacy information integration approaches. In *ICDE*, pages 485–486, 2005.
25. Angelos Hliaoutakis, Giannis Varelas, Epimenidis Voutsakis, Euripides GM Petrakis, and Evangelos Milios. Information retrieval by semantic similarity. *International Journal on Semantic Web and Information Systems*, 2(3):55–73, 2006.
26. Hanna Köpcke and Erhard Rahm. Training selection for tuning entity matching. In *QDB/MUD*, pages 3–12, 2008.
27. Yoonkyong Lee, Mayssam Sayyadian, AnHai Doan, and Arnon Rosenthal. eTuner: tuning schema matching software using synthetic scenarios. *VLDB J.*, 16(1):97–122, 2007.
28. Juanzi Li, Jie Tang, Yi Li, and Qiong Luo. Rimom: A dynamic multistrategy ontology alignment framework. *IEEE Trans. on Knowl. and Data Eng.*, 21(8):1218–1232, 2009.
29. Dekang Lin. An information-theoretic definition of similarity. In *ICML*, volume 98, pages 296–304, 1998.
30. Mochol Malgorzata, Jentzsch Anja, and Euzenat Jérôme. Applying an analytic method for matching approach selection. In *Ontology Matching*, volume 225 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.
31. Anan Marie and Avigdor Gal. Boosting schema matchers. In *OTM Conferences (1)*, pages 283–300, Berlin, Heidelberg, 2008. Springer-Verlag.
32. Sergey Melnik, Hector Garcia Molina, and Erhard Rahm. Similarity Flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of ICDE*, pages 117–128, 2002.
33. Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Developing metadata-intensive applications with Rondo. *J. of Web Semantics*, I:47–74, 2003.
34. Tom Mitchell. *Machine Learning*. McGraw-Hill Education (ISE Editions), October 1997.
35. Peter Mork, Len Seligman, Arnon Rosenthal, Joel Korb, and Chris Wolf. The harmony integration workbench. *J. Data Semantics*, 11:65–93, 2008.
36. S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.
37. University of Illinois. The UIUC web integration repository. http://metaquerier.cs.uiuc.edu/repository, 2003.
38. Heiko Paulheim, Sven Hertling, and Dominique Ritze. Towards evaluating interactive ontology matching tools. In *The Semantic Web: Semantics and Big Data*, pages 31–45. Springer, 2013.
39. Eric Peukert, Julian Eberius, and Erhard Rahm. Rule-based construction of matching processes. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2421–2424, New York, NY, USA, 2011. ACM.

40. Philip Resnik. Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language. *Journal of Artificial Intelligence Research*, 11:95–130, 1999.
41. Secondstring. http://secondstring.sourceforge.net/, 2014.
42. Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. *J. Data Semantics*, pages 146–171, 2005.
43. Pavel Shvaiko and Jérôme Euzenat. Ten challenges for ontology matching. In *OTM Conferences (2)*, pages 1164–1182, 2008.
44. Ken Smith, Michael Morse, Peter Mork, Maya Li, Arnon Rosenthal, David Allen, and Len Seligman. The role of schema matching in large enterprises. In *CIDR*, 2009.
45. William E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research*, pages 354–359, 1990.
46. Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, S Yu Philip, et al. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.
47. Li Xu and David W Embley. Using domain ontologies to discover direct and indirect matches for schema elements. pages 97–103, 2003.