Information Extraction from Web Pages Based on k-testable Tree Automaton Induction (extended abstract)

Raymond Kosala[1], **Maurice Bruynooghe**[1], **Jan Van den Bussche**[2], **Hendrik Blockeel**[1]
[1] Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Leuven
{raymond,maurice,hendrik}@cs.kuleuven.ac.be
[2] WNI, University of Limburg, Universitaire Campus, B-3590 Diepenbeek
jan.vandenbussche@luc.ac.be

Abstract

Information extraction refers to the process of extracting specific pieces of information from a document; for instance, extracting from a text the names of the authors. Much of the work on information extraction from HTML or XML documents uses methods for processing strings, such as finite automata. However, as these documents have a tree structure, it seems natural to exploit this structure by using techniques that parse trees, not strings. Tree automata are a suitable device for this. In this paper we explore methods for automatically learning tree automata that can be used for extracting specific information in tree-structured documents. We present an overview of several algorithms we have experimented with, as well as experimental results.

1 Introduction

Information extraction is the process of extracting specific information from documents. For example, one task might be to extract, from a text on some event, the date and location of the event. A wrapper is a procedure that takes as input a document and, using information extraction techniques, yields a more structured description of the document, for instance, a form with pre-defined fields that have an unambiguous meaning and contain the essential information in the document.

Information extraction interacts with information retrieval in a number of ways. First, it can be a preprocessing step for document retrieval. As argued by Lin and Ho (2002), many websites contain redundant information that can mislead search engines. An information extraction system can be used to filter many irrelevant or redundant features from a document and thus improve the indexing process. One could also consider the use of wrappers to create structured summaries of documents that can be used by document retrieval systems.

As a post-processor, an information extraction system can be used to re-score the relevance ranking of the documents stored by information retrieval systems. For instance, following the ideas in Bear et al. (1997), one could use an information extraction system to extract the most relevant nodes of a HTML document, count how many nodes are extracted, and re-score the relevance of the document based on that. Obviously, information extraction is also useful for retrieving information on a more fine-grained level than complete documents.

Information extraction procedures or wrappers are typically constructed for a specific kind of documents, because for information extraction to work well, documents must be sufficiently homogeneous with respect to semantics and/or format. Obviously, it is cumbersome to manually write wrappers for many different types of documents. Hence, there is a large interest in automatic construction of wrappers using machine learning techniques; this is referred to as wrapper induction. To automatically learn wrappers, one needs a formalism that is sufficiently powerful to represent a broad class of algorithms, but still limited enough to be able to learn them automatically.

A lot of work in the field of information extraction focuses on extracting information from unstructured text. In that case it is natural to consider the use of finite automata as wrappers: these handle strings, they have a reasonably large expressiveness and there exist algorithms for learning them automatically (Murphy, 1996). On the Web, however, many documents are tree-structured; this is specifically the case for HTML and XML documents. Although they can be handled by processing the document as a string, it seems natural to try to exploit the tree structure of these documents. Tree automata are then an obvious choice: they are the counterpart of finite automata for tree structured data.

Gottlob and Koch (2002) have recently related induction of tree automata to wrapper induction approaches, and found that all existing wrapper induction methods are special cases of induction of tree automata. This provides additional motivation for the use of tree automata for information extraction from web documents.

Kosala et al. (2002) investigate a number of approaches to induction of tree automata from HTML documents. This extended abstract is based on that

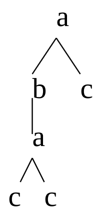

Figure 1: The tree represented in textual notation as $a(b(a(c,c)),c)$.

paper; we here focus less on technical details of the algorithms but describe them at a more intuitive level.

The remainder of this text is structured as follows. In Section 2 we briefly introduce tree automata and explain how they can be used for information extraction. In Section 3 we discuss the preprocessing of HTML documents that we found necessary in order to make our approach practically feasible. In Section 4 we turn our attention to the induction of tree automata, again in the context of information extraction. We introduce several induction algorithms with which we have experimented. These experiments are presented in detail in Section 5. In Section 6 we conclude.

2 Tree Automata

A ranked label f/n is a symbol f with a natural number n (called its rank or arity) associated with it. Let V denote a set of ranked labels, and V_k its subset of labels of rank k. $V = \cup_k V_k$. A tree over the set V is defined as follows: a label of rank 0, say $f/0$, is a tree (denoted f); if f/n is a label of rank f/n and $t_1, \ldots, t_n$ are trees, then $f(t_1, \ldots, t_n)$ is a tree; nothing else is a tree.

For example, with $V = \{a/2, b/1, c/0\}$, $a(b(a(c,c)),c)$ is a tree over V. It is depicted graphically in Figure 1.

A deterministic tree automaton (DTA) M is a quadruple (V, Q, Δ, F) where V is a set of ranked labels, Q is a finite set of states, $F \subseteq Q$ is a set of final or accepting states, and $\Delta : \cup_k (V_k \times Q^k) \to Q$ is the so-called transition function of M. Thus, if q_1, q_2 are states, then $\Delta(a, q_1, q_2) = q_2$ defines a single transition in Δ. We will also use the notation $a(q_1, q_2) \to q_2$.

A DTA processes trees bottom up. It starts with assigning states to leaves, according to the transition function. Whenever all children of a node have had a state assigned to them, the node itself is also assigned a state (again in accordance with the transition function). This process goes on until as many nodes as possible are assigned a state (nodes for which no transition applies may exist). We say that a tree is accepted if the root is assigned an accepting state (a state $q \in F$). The set of all trees accepted by a tree automaton M is called the language defined by M.

For example, for $V = \{a/2, b/1, c/0\}$, $Q = \{q_1, q_2\}$, $F = \{q_2\}$, and $\Delta = \{a(q_1, q_1) \to q_1, a(q_1, q_2) \to q_2, a(q_2, q_1) \to q_2, a(q_2, q_2) \to q_2, b(q_1) \to q_2, b(q_2) \to q_2, c \to q_1\}$, the tree automaton (V, Q, F, Δ) accepts those trees that have a b in them, and only those.

Given that a tree automaton accepts or rejects a whole tree, how can we use it for information extraction, where the aim is really to return a specific node in the tree? We follow the procedure proposed by Freitag (1997): we use a special symbol x to denote the field of interest. The automaton should accept a tree if and only if x occurs in the tree in the position of a node that is to be extracted. By consecutively substituting x for each node in a tree and running the tree automaton to see whether the resulting tree is accepted, we can identify all fields of interest.

We have now introduced the concept of tree automata and how they can be used for extracting a single node from a tree. In the next section we discuss how this approach can be followed in the context of information extraction from web pages.

3 Preprocessing of HTML Documents

The task we face is the following. We wish to learn a tree automaton that recognizes when the marker x is in the place of the information to be extracted. The input of the learning process is a set of examples, more precisely HTML documents with the fields of interest indicated. These would typically be provided by a human user. We assume that there are only positive examples, that is, the user does not need to provide examples of fields that are not to be extracted.

From the set of positive examples, our learning algorithm should construct a tree automaton that accepts all these examples, and generalizes over them in the sense that it should also accept previously unseen documents where the marker is in the right location.

It is not feasible to learn a tree automaton from the original HTML documents, for two reasons. First, a typical HTML document contains a limited set of markup tags, but at the lowest level of the tree an unlimited set of symbols (any text) can occur. To limit the complexity of the tree automaton to be learnt, we need to limit the size of the set V by preprocessing the documents.

A step that drastically reduces the size of V, is to replace all plain text by the symbol CDATA (except the text to be extracted, which is replaced by the marker x). In this case the information extraction procedure has only the exact structure of the document and the markup tags as guidance. This seems

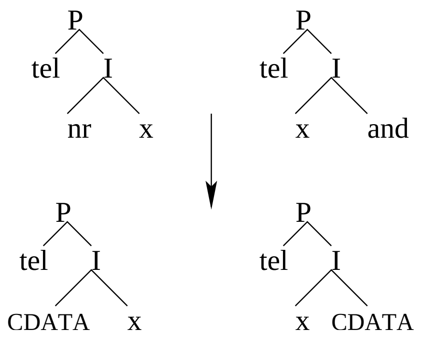

Figure 2: An example of the substitution of CDATA for text. "tel" is the closest (to x) invariant label and is kept as a distinguishing context, other text is changed into CDATA.

a very difficult task, and indeed experimental results with this approach show that overgeneralization occurs, that is, the induced tree automaton accepts many more trees than it should. This problem is alleviated in the following way.

The field to be extracted in HTML documents is often recognizable by some specific text occurring near it. We call this the distinguishing context. Our approach is to replace all text by CDATA except this distinguishing context, which is kept intact. The distinguishing context is determined automatically, using the following heuristic. The preprocessor looks for the invariant text label that is nearest to the field of interest and occurs at the same distance from the field of interest in all examples. If no such text is found, no context is used and all text is turned into CDATA. Figure 2 shows an example.

A second problem we face when trying to apply automaton induction algorithms in the context of HTML documents, is that HTML trees are unranked: a root with a given label can have an indefinite number of children. We have only ranked tree automaton inference algorithms at our disposal (induction of unranked tree automata is a different and more complex problem).

Fortunately, unranked trees can easily be transformed into ranked trees. Multiple children of a single node are then handled by putting them in a tree structure. More specifically, in the transformed tree, each node of the original tree occurs with as left child its leftmost child in the original tree (if any) and as right child the sibling that occurs immediately to its right (if any). Figure 3 illustrates the process. A formal definition of this transformation is given by Kosala et al. (2002). . Note that in the resulting binary tree it is essential to distinguish left and right subtrees of a node; if a node has only one child (that is, the other one is empty) it must still be clear whether it is the left or right child.

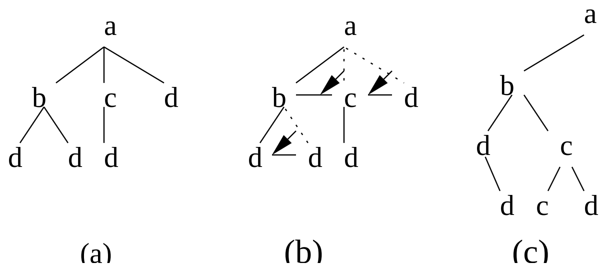

Figure 3: Transformation of an unranked tree into a ranked tree. (a) The original tree. (b) Nodes are linked to their immediately preceding sibling instead of their parent, except for the leftmost child of that parent. (c) The new tree.

4 Inducing Tree Automata

We have seen what tree automata are and how to represent HTML documents as ranked trees. The next topic we focus on is: how can one learn tree automata from example trees?

We have implemented several tree automaton induction algorithms and experimented with them. The basic algorithm that we started from, is the so-called k-testable algorithm developed by Rico-Juan et al. (2000).

A tree automaton (or the language it defines) is k-testable if it can be decided whether a tree belongs to the language or not by looking only at trees of height at most k (we define the height of a tree as the maximum number of nodes on any path from root to leaf).

For instance, if L is a 2-testable language then it is impossible that $t_1 = a(b(c), b(c))$ and $t_2 = a(b(d), b(d))$ are in L but $t_3 = a(b(c), b(d))$ is not. The reason is that $\{a(b, b), b(c), b(d)\}$ are the subtrees of height 2 that occur in t_1 and t_2 and that no other subtrees occur in t_3.

A k-testable automaton never needs more states than there are trees of height $k - 1$ or less. Given a number of trees and a fixed k, the most specific k-testable automaton[1] that accepts all these trees is the one that has a different state for each occurring subtree of height at most $k - 1$, and allows only the state transitions observed in the examples. The algorithm by Rico-Juan et al. constructs this automaton: it just collects all different subtrees of height at most $k - 1$ and creates a different state for each, assigns these states to all nodes of the tree, and subsequently adds all observed state transitions to Δ. It does this as follows.

In the terminology of Rico-Juan et al., "fork" refers to any subtree of a tree, "root" to a subtree occurring at the top of the tree, and "subtree" to a subtree occurring at the bottom of the tree (that

[1]We call an automaton M_1 more specific than M_2 if the language it defines is a subset of the language defined by M_2.

is, for which all leaves are also leaves of the original tree). From here on we use the term "subtree" in the same sense as Rico-Juan et al. For a k-testable language, the algorithm gathers all roots of height $k-1$ ("$k-1$-roots"), subtrees of height $k-1$ or less, and forks of height k. For instance, given the tree $a(b(c),c)$, with $k=3$ the root $a(b,c)$, fork $a(b(c),c)$ and subtrees $b(c)$ and c are gathered.

For a k-testable automaton, all encountered transitions are derivable from these forks, and all states needed are in the root, subsets, and $k-1$-roots of the forks. To distinguish between trees and states, we will denote the state associated with a tree t as $\overline{t}$. Thus, if we choose $k=3$, then for the example tree above the 3-testable automaton with states $\{\overline{a(b,c)}, \overline{b(c)}, \overline{c}\}$, transition function $\Delta = \{a(\overline{b(c)}, \overline{c}) \rightarrow \overline{a(b,c)}, b(\overline{c}) \rightarrow \overline{b(c)}, c \rightarrow \overline{c}\}$ and final state $\overline{a(b,c)}$ is derived. It accepts exactly this tree. A formal description of the algorithm, and also of the extensions we are about to discuss next, can be found in Kosala et al. (2002).

When applying this algorithm to information extraction, we identified some problems. One is that the resulting tree automaton, when applied to unseen documents, often was unable to parse the whole tree because certain kinds of transitions were needed that were never observed in the training set. This is in part due to the size of V, which even with the CDATA preprocessing step is still large. Note for instance that the automaton cannot reach an accepting state, hence that the extraction task fails, when the document contains a symbol that does not occur in the training set. HTML trees are often large and may contain parts that are entirely irrelevant to the extraction task. Hence any symbol in such irrelevant part that does not occur in the training set causes failure of the extraction task.

This is a clear case of undergeneralization: the resulting automaton is too specific and too strongly biased towards examples very similar to those in the training set. To improve generalization properties of the induction algorithm, we have developed two extensions to the algorithm. Both consist of generalizing the transition function of the original k-testable automaton, by learning transition rules that cover a set of states instead of a single state. This is done by introducing wildcards instead of specific labels in these transition rules.

For example, where the original algorithm would learn a transition $a(\overline{b(c)}, \overline{c}) \rightarrow \overline{a(b,c)}$ if and only if this transition occurs at least once in the example dataset, the modified algorithms might generalize this transition into $a(\overline{b(*)}, \overline{c}) \rightarrow \overline{a(b,c)}$, thus also allowing the automaton to parse $a(b(b(c)),c)$ even if no corresponding forks occurred in the examples.

Stronger generalization occurs if we put wildcards at other than the lowest level. The transition $a(*(*),c) \rightarrow a(*,c)$ indicates that also the label immediately below a can be anything, and the resulting state is in this case not determined. Thus when generalizing at any but the lowest level in this way, the resulting automaton becomes nondeterministic.[2]

Our two extensions differ in the way they introduce these wildcards.

The g-testable algorithm works as follows. Next to the parameter k, which has the same meaning as in the k-testable approach, there is a parameter l. The algorithm distinguishes so-called *context forks*, which contain the marker x, and "other forks", which do not contain the marker. Context forks are kept in the set CF and are not generalized with wildcards. Other forks are stored in the set OF and these are generalized by introducing wildcards for all labels below the level l. The idea behind this is to allow stronger generalization in those parts of the tree that do not contain the field to be extracted, than in those that do.

The gl-testable algorithm performs a more complicated kind of generalization. Instead of replacing all labels below a given level with wildcards, it inserts these wildcards more selectively. For instance, while the g-testable algorithm considers the minimal generalization of a state $a(b(c),b(c))$ to be $a(b(*),b(*))$, the gl-testable algorithm can also consider generalizations $a(b(*),b(c))$ and $a(b(c),b(*))$.

This additional flexibility causes the search for good generalizations to be more complicated. While generalizations for the g-testable algorithm are fully ordered, the generalizations considered in gl-testable are partially ordered and thus form a lattice that can be searched. The algorithm searches for generalizations that do not cover any context forks or subtrees.

More specifically, in a first step it tries to perform a relatively strong generalization over a set of forks: given a subset of forks with the same tree structure, it generalizes them towards a single fork by introducing wildcards in each node that does not have the same label in all forks. This generalization may or may not be succesful (it fails if the generalization covers a context fork). If it succeeds, the result is a single fork; if it does not, the result is the original set of forks. In a second step it cautiously further generalizes the fork(s) resulting from the first step by introducing more wildcards one by one, avoiding in each step to cover context forks.

To illustrate these procedures, assume we have sets of context forks $\{a(b,x)\}$ and other forks $\{a(b,c), a(b,b)\}$. The g-testable algoritm can only generalize the other forks to $\{a(*,*)\}$. The gl-testable algorithm, on the other hand, builds its first generalization by introducing wildcards for those

[2]In fact one could impose that the first and third wildcard in the example be the same, since we assume k-testability, but our notation of transitions does not allow this.

parts of the forks that are not common to all forks; in our case this yields the fork $\{a(b,*)\}$. This fork covers the context fork $a(b,x)$ and is therefore not allowed. The original forks are kept and cautiously generalized towards $\{a(*,b), a(*,c)\}$, none of which cover a context fork.

5 Experimental Results

We evaluated our method on some semi-structured data sets commonly used in the IE research (available from http://www.isi.edu/~muslea/RISE/): a collection of web pages containing people's contact addresses (the Internet Address Finder (IAF) database) and a collection of web pages about stock quotes (the Quote Server (QS) database). We also use a subset of the Shakespeare XML dataset (available from http://www.ibiblio.org/bosak/). For full details of the data sets we refer to Kosala et al. (2002).

We use criteria that are commonly used in the information retrieval research for evaluating our method: precision P (number of correctly extracted objects divided by total number of extractions), recall R (number of correct extractions divided by total number of objects present in the answer template), and $F1$, the harmonic mean of P and R: $2PR/(P+R)$.

Table 1 shows the results we obtained as well as those obtained by some current state-of-the-art string-based methods: an algorithm based on Hidden Markov Models (HMMs) (Freitag and McCallum, 1999), the Stalker wrapper induction algorithm (Muslea et al., 2001) and BWI (Freitag and Kushmerick, 2000). We also include the results of the k-testable algorithm in (Kosala et al., 2002b) and the g-testable algorithm in (Kosala et al., 2002a). The results of HMM, Stalker and BWI are adopted from (Freitag and Kushmerick, 2000). All tests are performed with 10-fold cross-validation following the splits used in (Freitag and Kushmerick, 2000), except for the small Shakespeare dataset where we used 2-fold cross-validation.

Table 1 shows the best results of gl-testable with a certain k that were obtained by cross-validation. As can be seen, our tree-based methods perform better in most of the test cases than the existing state-of-the-art string-based methods. The only exception is the field date in the QS dataset where BWI performs better. Compared to the results of k-testable, the gl-testable method performs better in the IAF-alt.name, IAF-organization and small Shakespeare data. Compared to the results of g-testable, gl-testable performs better in the IAF-alt.name and IAF-organization data but worse in the small Shakespeare data.

For the QS data, no difference between the k-testable, g-testable, and gl-testable algorithms was measured, which suggests that adding wildcards does not help in this case. For the small Shakespeare data, the gl-testable algorithm performs worse than the g-testable algorithm, somewhat surprisingly given its greater expressiveness. We suspect this is due to the fact that g-testible optimizes two parameters, not only k but also g.

It is also informative to look at the effect of k on the precision and recall of the induced automata. In Figure 4 these measures are shown for gl-testable. As expected, automata tend to become more specific with increasing k. In several cases an optimal value for k can clearly be identified.

The average training time of the tree-based algorithms varied from 0.5s to 7s in our experiments. Their theoretical training time is $O(km \log m)$ with m the sum of the depths of all example trees, so the algorithms scale well. The average extraction time per document varied from 4 to 14 seconds. The theoretical time complexity of the extraction procedure is $O(n^2)$ with n the number of nodes in the document, so the approach does not scale very well to very large documents (in terms of number of nodes). It is feasible for the type of documents considered here, though. We should also note that our implementation is a Prolog program and was not optimized for performance; we expect it is possible to reduce the extraction time with a considerable constant factor, for instance by indexing the states of the automaton.

6 Conclusions

We have argued for the use of tree automata, as opposed to finite automata, for information extraction from tree-structured documents such as web pages in HTML format. We have described a number of algorithms for learning such tree automata from examples (documents where the fields to be extracted are indicated) and presented experimental results indicating the performance of these algorithms on a few benchmark tasks. The results confirm our expectations that exploiting the tree structure of documents is beneficial to the performance of the information extractor.

Note that although we have discussed tree automata for information extraction, which is indirectly related to information retrieval, tree automata can in principle just as well be used for the document retrieval task itself, since a tree automaton just accepts or rejects trees (documents) as a whole. Such an approach would be useful in cases where the criterion for retrieving documents relies to some extent on their structural properties. We are unaware of any work investigating the use of tree automata for this purpose.

	IAF - alt.name			*IAF - organization*			*QS - date*			*QS - volume*		
	Prec	*Rec*	*F1*	*Prec*	*Rec*	*F1*	*Prec*	*Rec*	*F1*	*Prec*	*Rec*	*F1*
HMM	1.7	90	3.4	16.8	89.7	28.4	36.3	100	53.3	18.4	96.2	30.9
Stalker	100	-	-	48.0	-	-	0	-	-	0	-	-
BWI	90.9	43.5	58.8	77.5	45.9	57.7	100	100	100	100	61.9	76.5
k-testable	100	73.9	85	100	57.9	73.3	100	60.5	75.4	100	73.6	84.8
g-testable	100	73.9	85	100	82.6	90.5	100	60.5	75.4	100	73.6	84.8
gl-testable	100	84.8	91.8	100	84.6	91.7	100	60.5	75.4	100	73.6	84.8

	Small Shakespeare		
	Prec	*Rec*	*F1*
k-testable	56.2	90	69.2
g-testable	69.2	90	78.2
gl-testable	66.7	80	72.7

Table 1: Comparison of the results

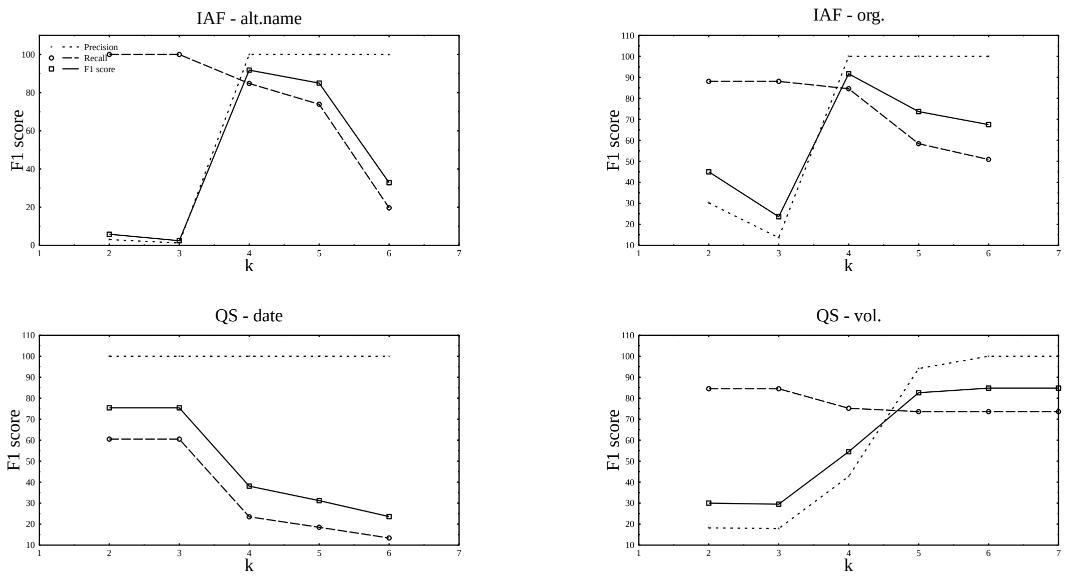

Figure 4: The graphs of F1 score vs k

Acknowledgements

We thank Nicholas Kushmerick for providing us with the datasets used for the BWI experiments. H.B is a postdoctoral fellow of the Fund for Scientific Research of Flanders (FWO-Vlaanderen), which has also supported this work through the project "Query languages for data mining".

References

J. Bear, D. Israel, J. Petit, and D. Martin. Using information extraction to improve document retrieval. In *Proceedings of the Sixth Text Retrieval Conference*, pages 367-378, 1997.

D. Freitag and N. Kushmerick. 2000. Boosted wrapper induction. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Innovative Applications of AI Conference*, pages 577–583. AAAI Press.

D. Freitag and A. McCallum. 1999. Information extraction with HMMs and shrinkage. In *AAAI-99 Workshop on Machine Learning for Information Extraction*.

D. Freitag. 1997. Using grammatical inference to improve precision in information extraction. In *ICML-97 Workshop on Automata Induction, Grammatical Inference, and Language Acquisition*.

G. Gottlob and K. Koch. 2002. Monadic datalog over trees and the expressive power of languages for web information extraction. In *21st ACM Symposium on Principles of Database Systems*, pages 17–28.

R. Kosala, M. Bruynooghe, H. Blockeel, and

J. Van den Bussche. 2002a. Information extraction by means of a generalized k-testable tree automata inference algorithm. In *Proceedings of the Fourth International Conference on Information Integration and Web-based Applications & Services (IIWAS)*. To appear.

R. Kosala, J. Van den Bussche, M. Bruynooghe, and H. Blockeel. 2002b. Information extraction in structured documents using tree automata induction. In *Proceedings of the 6th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pages 299–310.

R. Kosala, M. Bruynooghe, J. Van den Bussche and H. Blockeel. Information extraction from web pages based on k-testable tree automaton inference. Submitted, 2002.

S. Lin and J. Ho. Discovering informative content blocks from web documents. In *Proceedings of the Eight ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002.

K. Murphy. 1996. Learning finite automata. Technical Report 96-04-017, Santa Fe Institute.

I. Muslea, S. Minton, and C. Knoblock. 2001. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:93–114.

J.R. Rico-Juan, J. Calera-Rubio, and R.C. Carrasco. 2000. Probabilistic k-testable tree-languages. In A.L. Oliveira, editor, *Proceedings of 5th International Colloquium, ICGI 2000, Lisbon (Portugal)*, volume 1891 of *Lecture Notes in Computer Science*, pages 221–228. Springer.