



ELSEVIER

Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

Data & Knowledge Engineering 58 (2006) 129–158

DATA &
KNOWLEDGE
ENGINEERING

www.elsevier.com/locate/datak

Information extraction from structured documents using k -testable tree automaton inference

Raymond Kosala^a, Hendrik Blockeel^{a,*}, Maurice Bruynooghe^a,
Jan Van den Bussche^b

^a Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium

^b Department WNI, Limburgs Universitair Centrum, Universitaire Campus, B-3590 Diepenbeek, Belgium

Received 3 February 2005; accepted 17 May 2005

Available online 13 June 2005

Abstract

Information extraction (IE) addresses the problem of extracting specific information from a collection of documents. Much of the previous work on IE from structured documents, such as HTML or XML, uses learning techniques that are based on strings, such as finite automata induction. These methods do not exploit the tree structure of the documents. A natural way to do this is to induce tree automata, which are like finite state automata but parse trees instead of strings. In this work, we explore induction of k -testable ranked tree automata from a small set of annotated examples. We describe three variants which differ in the way they generalize the inferred automaton. Experimental results on a set of benchmark data sets show that our approach compares favorably to string-based approaches. However, the quality of the extraction is still suboptimal.

© 2005 Elsevier B.V. All rights reserved.

Keywords: Information extraction; Wrapper induction; Tree automata; Machine learning

* Corresponding author.

E-mail addresses: raykosala@yahoo.com (R. Kosala), hendrik@cs.kuleuven.be (H. Blockeel), maurice@cs.kuleuven.be (M. Bruynooghe), jan.vandenbussche@luc.ac.be (J. Van den Bussche).

1. Introduction

When searching for particular information, one is usually confronted with two problems. The first one is the selection of relevant documents and is addressed by information retrieval; the second one is the extraction of specific information from documents and is addressed by information extraction (IE). Traditionally, IE is focussed on the extraction of information from unstructured texts. It is a major issue at fora such as the Message Understanding Conferences (MUC) and Text REtrieval Conferences (TREC). For example a IE task investigated in the Sixth Message Understanding Conference (MUC-6) [37], is “Management Succession”. Given an article, the tasks are to extract the name of the new company officers or the old officers, the company name, and the position title succeeded.

However, besides IE from unstructured texts, one distinguished also IE from semi-structured texts and IE from structured texts [51]. The latter have gained importance with the explosive growth of the World Wide Web (Web) as a medium for disseminating information and the work on Web information integration [35,55]. Indeed, there is an increasing need for IE systems that support the extraction of information from Web documents. The latter are often stored as HTML and/or XML documents, hence are (semi-)structured. These documents make use of non-linguistic elements, such as HTML/XML tags, and sometimes use ungrammatical language to convey information. This makes the methods appropriate for grammatical text mostly unusable and calls for a non-linguistic approach.

Several query languages supporting the extraction of information from Web data have been developed. Examples are in [5,57]. However, their use is time consuming and requires non-trivial skill. As argued by several authors [41,33], there is a need for systems that can learn to extract information from a few annotated examples. The reason is that building IE systems manually is not feasible and scalable for such a dynamic and diverse medium as the Web. The problem, also known as wrapper induction, has already been addressed by several authors. Several machine learning techniques for inducing wrappers have been proposed, for instance rule learning algorithms [17] or multi-strategy approaches [18]. Also grammatical inference techniques are used to induce a kind of delimiter-based patterns [41,20,19,51,16,27,10].

These methods consider the document as a string. However, structured documents such as HTML and XML documents have a tree structure. Therefore it is natural to explore the use of tree automata for IE from structured documents. Indeed, tree automata are well-established and natural tools for processing trees [14]. An advantage of using the more expressive tree formalism is that the extracted field can depend on its structural context in a document. A structural context that is close to the target field in the tree structure of the document, and at a fixed distance from it, can be arbitrarily far away and at variable distances from it in the string representing the document. This makes the learning task very difficult if a string-based approach is used, and results in wrappers with rather poor performance.

The current paper develops a novel wrapper induction method that utilizes the tree structure of the document. Accordingly, it uses tree automata as wrappers. Recent work by Gottlob and Koch [23] shows that all existing wrapper languages for structured document IE can be captured using tree automata. This result provides a strong justification for the use of tree automata instead of string automata.

We will use the k -testable tree automaton inference algorithm [46], an algorithm for grammatical inference that is able to identify in the limit [22] any k -testable tree language (in the strict sense) from positive examples only. Informally, a k -testable tree language is a language that can be determined just by looking at all the subtrees of length k . The amount of generalization occurring when learning from the positive examples of the k -testable tree algorithm is mainly determined by the value of k ; it decreases with increasing k . It is an algorithm for ranked trees (where the number of children of a node is fixed given the symbol in the node), while documents are unranked trees. The simplest way to apply the algorithm is to convert Web documents into (ranked) binary trees. As the extraction is based on some structural context, k must be large enough such that the field to be extracted and its structural context are covered in the same subtree. Because of the binarization, the value of k needed for capturing the structural (or distinguishing) context tends to be rather large. Consequently, the generalization tends to be rather low, often resulting in rather poor recall [31].

To overcome this problem, we have experimented with two generalizations of the k -testable algorithm, namely, the g -testable and gl -testable algorithms. In the g -testable algorithm [30], the generalization is parameterized by l . It considers generalizations of states (which are trees) where the state labels at the lowest l levels are replaced by wildcards. The gl -algorithm, which is introduced in this paper, considers another generalization and uses the partial order between different generalizations to limit the search. Experiments show that these generalizations improve the performance of the induced wrappers.

Not only does our method exploit the tree structure, it also requires very little user intervention. The user only has to annotate the field to be extracted in a few representative examples. Previous approaches require substantially more user intervention such as splitting the document in small fragments, and selecting some of them for use as a training example [51]; or the manual specification of the length of a window for the prefix, suffix and target fragments [20,19], and of the special tokens or landmarks such as “>” or “;” [19,41]. Not all of these parameters can easily be optimized using cross-validation, as we do with the k and l parameters.

Preliminary versions of parts of this article appeared in conference proceedings (k -testable algorithm [31], g -testable algorithm [30]).

The rest of the paper is organized as follows. Section 2 provides some background on tree automata and their use for IE. Section 3 describes our methodology, the k -testable algorithm and its generalizations: the g -testable and gl -testable algorithms. Experimental setting and results are described in Section 4 and related work in Section 5. In Section 6 we conclude by summarizing the contributions of this paper.

2. Preliminaries

2.1. Grammatical inference

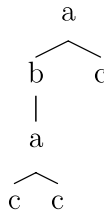
Grammatical inference refers to the process of learning rules from a set of labeled examples. It belongs to a class of inductive inference problems [2] in which the target domain is a formal

language (a set of strings over some alphabet Σ) and the hypothesis space is a family of grammars. It is also often referred to as automata induction, grammar induction, or automatic language acquisition. It is a well-established research field in AI that goes back to Gold's work [22]. The inference process aims at finding a minimum automaton (the *canonical automaton*) that is *compatible* with the examples. The compatibility with the examples depends on the applied quality criterion. Quality criteria that are generally used are exact learning in the limit of Gold [22], query learning of Angluin [1] and probably approximately correct (PAC) learning of Valiant [53]. There is a large body of work on grammatical inference, including some excellent surveys [38,49,44].

In regular grammar inference, we have a finite alphabet Σ and a regular language $L \subseteq \Sigma^*$. Given a set of examples that are in the language (S^+) and a (possibly empty) set of examples not in the language (S^-), the task is to infer a deterministic finite automaton (DFA) A that accepts the examples in S^+ and rejects the examples in S^- .

2.2. Tree automata

Assume given a finite set V of labels, each with an associated rank (or arity; a natural number). Trees labeled by V are formally defined as terms over V , as follows: a label of rank 0 ($f/0$ or just f) is a tree; and if f/n is a label of rank $n > 0$ and t_1, \dots, t_n are trees, then $f(t_1, \dots, t_n)$ is a tree. For example the term $a(b(a(c, c)), c)$ with $a/2, b/1, c/0 \in V$, represents the tree shown below.



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 (accepting) states, and $\Delta: \cup_k V_k \times Q^k \rightarrow Q$ is the transition function. Here, V_k denotes the subset of V of labels of rank k . For example, $(v, q_1, \dots, q_k) \rightarrow q$, where $v/k \in V_k$ and $q, q_i \in Q$, represents a transition.

A DTA processes trees bottom up. Given a leaf labeled $v/0$ and a transition $(v) \rightarrow q$, the state q is assigned to it. Given a node labeled v/k with children in state q_1, \dots, q_k and a transition $(v, q_1, \dots, q_k) \rightarrow q$, the state q is assigned to it. We say that a tree is accepted if the state assigned to its root is accepting, i.e., belongs to F .

Grammatical inference can be generalized from string languages to tree languages. Rather than a set of strings over an alphabet Σ given as example, we are now given a set of trees over a ranked alphabet V . Rather than inferring a standard finite automaton compatible with the string examples, we now want to infer a compatible tree automaton. Tree automata are the natural generalization of string automata. Typically algorithms for tree automata induction are developed by upgrading the existing algorithms for string automata induction (e.g., [46,48]).

2.3. Information extraction by grammatical inference

If we model structured documents as trees over some ranked alphabet V as above, an IE task can be reduced to a grammatical inference task (as noted by Freitag [16]). Specifically, suppose the IE task consists of selecting certain nodes from a tree. We are given a set of examples, each consisting of a tree and a selected node. By adding for each label $v \in V$ a new label (v, x) , where x is a new “target” symbol, we can represent such examples as trees over the new alphabet $V' = V \cup (V \times \{x\})$, where the label of precisely one node, namely the selected one, is in $V \times \{x\}$, and the other labels are in V as before.

We can now try to infer a grammar for the obtained set of example trees, producing a DTA M over V' . If successful, we can use M to perform the original IE task simply by selecting each node, one by one, relabeling it to (v, x) if its original label is v , and verifying whether M accepts the thus relabeled tree. If so, the selected node is extracted.

In what follows, we will focus on applications where only leaf nodes are to be extracted. These nodes are text nodes; unseen documents likely include unseen text nodes. As explained below, to obtain a sufficiently general wrapper, they are generalized into a fixed known label (CDATA). It implies the v in the pair (v, x) carries no useful information. Hence we can simplify the setting a bit by labeling selected nodes simply by the target symbol x instead of (v, x) and the new alphabet V' then simply becomes $V \cup \{x\}$.

3. Approach and algorithms

Structured documents in HTML, or, more generally, XML format, can be readily represented as trees, where internal nodes represent the elements, and are labeled by tags, and leaf nodes represent the text content. Before we can use grammatical inference to perform IE on such trees, as described above, we must deal with two issues:

1. How do we deal with text content? A node of the tree contains one symbol, which in the context of HTML or XML documents can be any piece of text. We cannot have a separate symbol for any text, that would yield an infinite alphabet and overly specific automata.
2. Tags are not ranked. Recall that in a ranked tree, any node with a given symbol (in this case, a given HTML or XML tag) always has the same number of children. But in HTML, for example, an `` element can have an arbitrary number of `` subelements, and more generally, in XML documents, there is no bound on the number of subelements an element can have.

In the next two subsections, we will deal with these two issues. After that, we summarize our general approach. Finally, we introduce the various concrete grammatical inference algorithms we will use for IE from structured documents.

3.1. Preprocessing

Fig. 1 shows a simplified view of a representative document. (The real documents, as used in the experiments by us and the other authors, are more complex.) In this document, the fields to be



Fig. 1. An example of a HTML document.

extracted are the fields following the 'Alt.Name' and 'Organization' fields. A document consists of a variable number of records. As we can see, in each record the number of occurrences of the fields to be extracted is also variable (from zero to several occurrences). Also the position where they occur is not fixed. There is evidence that extracting this kind of information is a difficult task [26,40].

An important issue is how to deal with the various text nodes in the document. Treating every piece of text as a distinct label is unacceptable as it results in overly specific automata. Labeling all text nodes (except the node to be extracted, which is labeled x) by some fixed label CDATA, as in XML DTD's [56], is also unacceptable, as this results in overly general automata. Indeed, consider Fig. 2 which shows a fragment of a document tree that could originate from the document shown in Fig. 1.

Suppose the target field x is always preceded by a field labeled Organization. If the labels Provider and Organization are both replaced by CDATA then any automaton that extracts the x node will likely also extract the att node when it is replaced by x . Hence we should not replace the field Organization by CDATA. Fields such as Organization and Alt.Name are called *distinguishing contexts* (or structural context). Roughly speaking, a distinguishing context is the text content of a tree node that is useful for the identification of the field of interest. However, not every field of interest has a unique distinguishing context.

In our experiments, we consistently used the following automatic procedure to determine the distinguishing context. We look for the invariant text label that is nearest to the field of interest

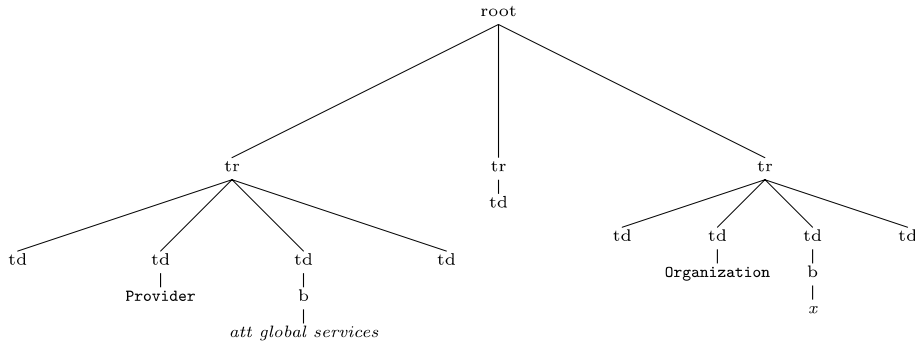


Fig. 2. A fragment of a HTML tree for the document in Fig. 1.

and occurs at the same distance from the field of interest in all examples. For example, the text ‘Organization:’ is an invariant text label that is nearest to the organization name in the HTML document of Fig. 1. If no such text is found, no context is used and all text is turned into CDATA. If there are several possibilities, one is chosen at random. As distance measure, we use the length of the shortest path in the document tree (for example, the distance of a node to its parent is one; to its sibling, two; to its uncle, three).

3.2. Conversion to ranked trees

Existing tree automata inference algorithms expect ranked trees. The simplest way to apply them on HTML or XML documents, which are unranked trees, is to transform the latter into ranked (binary) trees. This is the approach that we follow in this paper.

Using the symbol T to denote unranked trees and F to denote a sequence of unranked trees (a forest), the following grammar defines unranked trees:

$$T ::= a(F), \quad a \in V \quad F ::= \epsilon$$

$$F ::= T, F$$

One can think for several ways of representing an unranked tree by a ranked one. For example, one could replace the children of a node by a list of nodes. The unranked tree on the left of Fig. 3 would then be encoded as $a([b, c([a]), d])$, the second tree in Fig. 3. However this has the drawback of introducing extra symbols (the list constructor and the empty list) and of increasing the

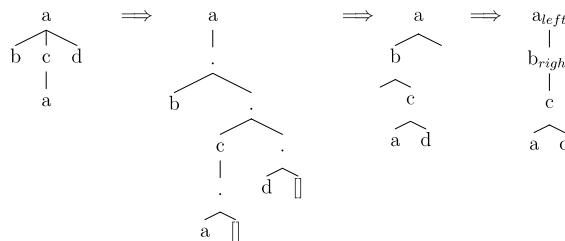


Fig. 3. An unranked tree (left) and its representation using lists, as a ranked binary tree without list constructors, and using the $encode_f$ function (right).

distance between each pair of nodes. Given that extraction is to be based on the local context of nodes, this is a serious drawback. Obviously, it is not possible to preserve all distances. We opt for a transformation that preserves the distance between a node and all its siblings and between a node and its first child. Such a transformation can be formally defined with the following recursive function $encode$ (with $encode_f$ for the encoding of forests):

$$\begin{aligned}
 encode(T) & \stackrel{\text{def}}{=} encode_f(T) \\
 encode_f(F) & \stackrel{\text{def}}{=} \begin{cases} a & \text{if } F = a(\epsilon) \\ a_{right}(encode_f(F_2)) & \text{if } F = a(\epsilon), F_2 \\ a_{left}(encode_f(F_1)) & \text{if } F = a(F_1) \\ a(encode_f(F_1), encode_f(F_2)) & \text{if } F = a(F_1), F_2 \end{cases}
 \end{aligned}$$

As can be seen in Fig. 3, the first child of a node v in an unranked tree T is encoded as the left child of the corresponding node v' of T' , the binary encoding of T , while the right sibling of a node v in tree T is encoded as the right child of v' in T' . To distinguish between a node with one left child and a node with one right child, the node is annotated with left and right respectively. For example, the unranked tree $a(b, c(a), d)$ is encoded into the binary tree $a_{left}(b_{right}(c(a), d))$ as shown in the figure. Note that the binary tree has exactly the same number of nodes as the original tree.

As mentioned above, we can now directly use the tree automata inference algorithms that have been proposed in the literature [21,46,7,3]. In this paper we explore the application of the k -testable algorithm [21,46]. We choose the k -testable algorithm because it requires fewer examples than the other tree-based algorithms, and hence less effort from the user (who needs to provide these labeled examples). A drawback of converting an unranked tree to a binary tree is that the distance between the distinguishing context and the target node can increase. A so far less explored alternative is to work directly with unranked trees. Unranked tree automata [43,52] have transition rules of the form $(v, e) \rightarrow q$, where e is a regular expression that describes a sequence of states. A first step towards using them for information extraction has recently been made [29].

3.3. Approach

Our approach for information extraction has the following characteristics:

- Strings stored at the nodes are treated as a whole. If extracted, the whole node is returned. For example, our method is able to extract the whole node “*att global services*” in Fig. 2, but is not able to extract the substring “*att*” only.
- One automaton is learned for one type of field to be extracted, e.g., the field following “*Organization*” in Fig. 2.
- In the examples used during learning, one target field is replaced by x . When a document contains several fields of the same type, then several examples are created from it, one for each occurrence of the target field.

One characteristic of our tree automata wrappers is that they do *single-slot* (or single-field) extraction. A single-slot IE system extracts isolated facts from the text, while a *multi-slot* IE system groups the related extracted fields together into correctly ordered multi-slot facts. A group of related ex-

tracted fields is commonly called a *case frame*. There are some domains where multi-slot extraction is a necessity. For example, a webpage may contain a list of house addresses with their corresponding prices. Unless the address and the price are combined in a pair, the extracted information is rather useless because we do not know which price should be the correct attribute for a particular address.

Multi-slot extraction can be achieved by extracting also the location of the extracted fields. Knowing the locations, one can combine the extracted fields into the correct tuples. This simple post-processing method works reasonably well for structured documents such as HTML/XML documents, since the order of the fields in a case frame typically follows the order of their position in the document. However, we did not collect the extracted fields in a case frame in our experiments.

The learning procedure is as follows:

1. Replace in the examples the target field by “*x*”, the distinguishing context (if present) by “*ctx*” and all other text fields by CDATA.
2. Convert the example trees to binary trees.
3. Run a tree automaton inference algorithm on the examples and return the inferred automaton.

The extraction procedure is as follows:

1. Replace the distinguishing context (if present) by “*ctx*” and all other text fields by CDATA.
2. Convert the tree to a binary tree.
3. Repeat for all CDATA nodes:
 - Replace the label of one CDATA node by the special label ‘*x*’.
 - Run the inferred tree automaton.
 - If the tree is accepted by the automaton, then extract the original text of the node labeled with *x*.

The automaton can succeed for zero, one or more text nodes. The text nodes for which it succeeds are the extracted fields. Only the first step of the learning procedure requires user intervention. The second step of the learning procedure and the whole extraction procedure are done automatically. The above procedures are repeated for each field of interest in the dataset. If one wants to do a novel extraction task on a novel dataset, then the learning procedure above has to be done for this novel task. That is, the user should mark the fields of interest and distinguishing context if they exist, then run the learning algorithm on the novel data.

3.4. Tree automaton inference algorithms

The *k*-testable algorithm is a basic tree automaton inference algorithm. As we will see in Section 4, the algorithm is precise, but is sometimes too specific, as indicated by the low recall. Hence we develop two generalization algorithms: *g*-testable and *gl*-testable. We start with some definitions.

3.4.1. Definitions

The basic idea underlying *k*-testable languages is that a language is defined by a set of small fragments (of size at most *k*). Any structure built from only those fragments is an element of

the language. To learn such a language, one simply extracts all the fragments of size at most k from the positive examples. An unseen example is classified as a member of a given language if and only if its set of fragments is a subset of the fragments representing the language.

In the case of k -testable tree languages, the fragments are tree structured, and we divide them into roots, forks and subtrees, depending on whether they occur at the top of the example trees (roots), at the bottom (subtrees), or anywhere (forks). To define them formally, we first introduce the notion of *height* of a tree: $\text{height}(t)$ is the number of nodes on the longest path from the root.

The k -root $r_k(t)$ of a tree t is the tree of height at most k obtained from t by cutting off branches longer than k :

$$r_k(v(t_1, \dots, t_m)) = \begin{cases} v & \text{if } k = 1 \\ v(r_{k-1}(t_1), \dots, r_{k-1}(t_m)) & \text{if } k > 1 \end{cases} \quad (1)$$

The set $f_k(t)$ of k -forks is the set of all trees of height k obtained from t by taking all subtrees of height at least k and cutting off branches longer than k :

$$f_k(v(t_1, \dots, t_m)) = \begin{cases} \emptyset & \text{if } \text{height}(v(t_1, \dots, t_m)) < k \\ \bigcup_{j=1}^m f_k(t_j) \cup \{r_k(v(t_1, \dots, t_m))\} & \text{otherwise} \end{cases} \quad (2)$$

Finally, the set $s_k(t)$ of k -subtrees is the set of all subtrees at the bottom of t of height at most k :

$$s_k(v(t_1, \dots, t_m)) = \bigcup_{j=1}^m s_k(t_j) \cup \begin{cases} \emptyset & \text{if } \text{height}(v(t_1, \dots, t_m)) > k \\ v(t_1, \dots, t_m) & \text{otherwise} \end{cases} \quad (3)$$

Note that the k -root and the k -subtrees have height at most k , and that the k -forks have height exactly k . An example is shown in Fig. 4.

The *level* of a node is defined as the number of edges on the path from the node to the root. The *skeleton* of tree t , $\text{skeleton}(t)$, is defined as t with all of its labels, except the root label, changed to a wildcard $*$. For example, $\text{skeleton}(a(b(d), c(e, f))) = a(*(*), *(*, *))$. A *partition* of a set S is a set of disjoint non-empty subsets of S (called *classes*) such that the union of the subsets is S . The *children* of a tree $v(t_1, \dots, t_m)$ are t_1, \dots, t_m . A tree t *covers* a tree t' if t' can be derived from t by replacing some of the wildcards in t .

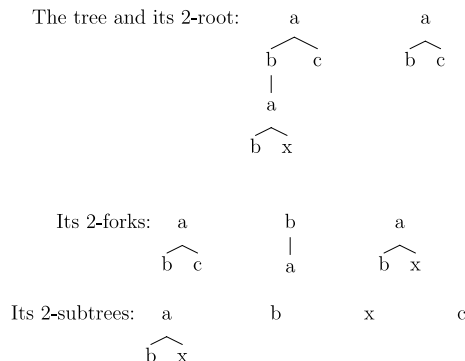


Fig. 4. A tree $a(b(a(b, x)), c)$, its 2-root, 2-forks and 2-subtrees.

3.4.2. The k -testable algorithm

The k -testable algorithm [46] is parameterized by a natural number k ; its name comes from the notion of a “ k -testable tree language”. Informally, a tree language (set of trees) is k -testable if membership of a tree in the language can be determined just by looking at its $(k - 1)$ -root, k -forks, and $(k - 1)$ -subtrees. The k -testable algorithm is capable of identifying in the limit any k -testable tree language from positive examples only. We have selected it because the information to be extracted typically has a locally testable character. Intuitively, given an example, the right value of k is the minimal value that ensures that the target x and the distinguishing context are in the same fork.

The choice of k is performed automatically using cross-validation, choosing the smallest k giving the best results. Our cross-validation approach takes randomly one half of the dataset for training and uses the rest for testing. First calculating a score for $k = 2$, the value for k is increased until the score shows a decrease. The least k -value giving a maximal score is then selected as the best value. As argued in Section 4, the FI -score used to measure the quality of the extraction first increases and then decreases, hence the selected k is indeed the optimal one.

The procedure to learn the tree automaton [46] is shown in Algorithm 1. Essentially, the automaton is an efficient procedure to check whether a tree contains only fragments that occurred in the training examples. The algorithm uses the $(k - 1)$ -roots, k -forks and $(k - 1)$ -subtrees occurring in the examples to derive states and transitions.

The final state FS of the automaton is the $(k - 1)$ -root of the tree. The set of states Q is the union of the $(k - 1)$ -subtrees (\mathcal{S}), the $(k - 1)$ -forks (computed as the $(k - 1)$ -roots of the k -forks \mathcal{F}) and the $(k - 1)$ -roots (which are a subset of the $(k - 1)$ -forks). Using trees as states ensures that the state associated with a node not only depends on its label, but also on the states of the children. Finally, there is a transition for each $(k - 1)$ -subtree (the state of the node at the root of the subtree is the subtree itself) and a transition for each k -fork (the state of the node at the root of the fork is the $(k - 1)$ -root of the fork).

Algorithm 1. k -testable.

Input: A set T of positive examples (ranked trees over V) and a positive integer k .

Output: A tree automaton (V, Q, Δ, FS) .

- 1: $\mathcal{F} := \bigcup \{f_k(t) \mid t \in T\}$
- 2: $\mathcal{S} := \bigcup \{s_{k-1}(t) \mid t \in T\}$
- 3: $FS := \{r_{k-1}(t) \mid t \in T\}$
- 4: $Q := \mathcal{S} \cup FS \cup \{r_{k-1}(f) \mid f \in \mathcal{F}\}$
- 5: $\Delta := \{(v, t_1, \dots, t_m) \rightarrow v(t_1, \dots, t_m) \mid v(t_1, \dots, t_m) \in \mathcal{S}\}$
- 6: $\Delta := \Delta \cup \{(v, t_1, \dots, t_m) \rightarrow r_{k-1}(v(t_1, \dots, t_m)) \mid v(t_1, \dots, t_m) \in \mathcal{F}\}$

It was observed that it was difficult to obtain optimal results. For small k , too many targets are extracted. Increasing k eliminates false positives but also true positives as the generalization tends to be insufficient for large k . Therefore, two (ad hoc) variants were developed that generalize better for large k . Their development was motivated by the intuition that the forks that contribute to the recognition of the target are close to it while forks further away are merely contributing to recognizing that the document belongs to the class containing targets. As it is assumed that all documents belong to the right class, the latter is not part of the extraction task. Moreover, learning the class tends to require more examples than learning

the target, hence one can better generalize the forks mainly contributing to the recognition of the class.

3.4.3. The g -testable algorithm

The basic idea of the g -testable algorithm is to generalize the transitions originating from forks that are not important for the recognition of the target. Important forks are those that contain the target label x as they belong to the local context of the target. In Algorithm 2, they are collected in the set \mathcal{TF} (target forks), the other ones in the set \mathcal{OF} (other forks). The generalization is parameterized by a level l that measures the distance to the root. It replaces the label of a node by a wildcard when its distance to the root is at least l ; hence generalization increases with decreasing l . The algorithm uses a function $gen(f, l)$ for this. Fig. 5 shows a fork f (left), $gen(f, 1)$ (middle) and $gen(f, 2)$ (right). To prevent overgeneralization and to keep the automaton deterministic, we require that the generalization of a fork does not interfere with forks close to the target. Formally, the condition is that it does not cover any target fork. The value of parameters k and l are determined by the same cross-validation method as explained above.

The meaning of a generalized fork is the set of all trees that can be obtained by instantiating labels for the wildcards. Generalized forks yield transitions with wildcards. For example, with $k = 3$ and $l = 2$, the fork $gen(f, 2)$ from the figure would yield the transition $a(b, c(*, *)) \rightarrow a(b, c)$, which on a 5-label alphabet $V = \{a, b, c, d, e\}$ effectively stands for the $5^2 = 25$ possible transitions obtained by instantiating labels for the wildcards. Similarly, with $k = 3$ and $l = 1$, the fork $gen(f, 1)$ from Fig. 5 would yield the transition $a(*, *(*, *)) \rightarrow a(*, *)$, which then stands for 5^4 possible transitions obtained by instantiating labels for the wildcards on the left-hand side of the transition. Since the right-hand side stands for the 2-root of the fork, the wildcards on the right-hand side are instantiated in accordance with the left-hand side. Some concrete example instantiations of the transition are:

- $a(b, c(d, e)) \rightarrow a(b, c)$,
- $a(d, e(b, c)) \rightarrow a(d, e)$,
- $a(a, a(b, b)) \rightarrow a(a, a)$.

The detailed g -testable procedure is shown in Algorithm 2. \mathcal{F}_0 , the set of k -forks, is divided in \mathcal{TF} , the target forks, $\mathcal{OF}_{\text{nogen}}$, the other forks that cannot be generalized because the generalization would interfere with the target forks, and the forks to be generalized. After generalization, the latter yield $\mathcal{OF}_{\text{gen}}$. Together, the three sets make up the \mathcal{F} , the final set of forks. The remaining difference with the k -testable algorithm is that the $(k - 1)$ -subtrees of k -forks are explicitly added as states. In the k -testable algorithm, they were already present as $(k - 1)$ -subtrees or as $(k - 1)$ -roots of other forks. For generalized k -forks, this is no longer the case due to the wild-

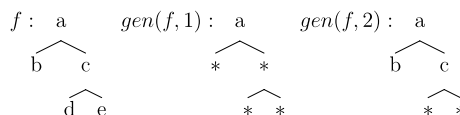


Fig. 5. Generalizing a fork.

cards. Note also that with $l = k$, the g -testable algorithm will output exactly the same automaton as the k -testable algorithm as no wildcards are introduced.

Algorithm 2. g -testable algorithm.

Input: A set T of positive examples, parameters k and l .

Output: A tree automaton (V, Q, Δ, FS)

- 1: $\mathcal{F}_0 := \bigcup \{f_k(t) \mid t \in T\}$
- 2: $\mathcal{T}\mathcal{F} := \{f \in \mathcal{F}_0 \mid f \text{ contains } x\}$
- 3: $\mathcal{O}\mathcal{F}_0 := \mathcal{F}_0 - \mathcal{T}\mathcal{F}$
- 4: $\mathcal{O}\mathcal{F}_{\text{nogen}} := \{f \in \mathcal{O}\mathcal{F}_0 \mid \text{gen}(f, l) \text{ covers one of } \mathcal{T}\mathcal{F}\}$
- 5: $\mathcal{O}\mathcal{F}_{\text{gen}} := \{\text{gen}(f, l) \mid f \in \mathcal{O}\mathcal{F}_0 - \mathcal{O}\mathcal{F}_{\text{nogen}}\}$
- 6: $\mathcal{F} := \mathcal{T}\mathcal{F} \cup \mathcal{O}\mathcal{F}_{\text{gen}} \cup \mathcal{O}\mathcal{F}_{\text{nogen}}$
- 7: $\mathcal{S} := \bigcup \{s_{k-1}(t) \mid t \in T\}$
- 8: $FS := \{r_{k-1}(t) \mid t \in T\}$
- 9: $Q := \mathcal{S} \cup FS \cup \{r_{k-1}(f) \mid f \in \mathcal{F}\} \cup \bigcup \{s_{k-1}(f) \mid f \in \mathcal{F}\}$
- 10: $\Delta := \{(v, t_1, \dots, t_m) \rightarrow v(t_1, \dots, t_m) \mid v(t_1, \dots, t_m) \in \mathcal{S}\}$
- 11: $\Delta := \Delta \cup \{(v, t_1, \dots, t_m) \rightarrow r_{k-1}(v(t_1, \dots, t_m)) \mid v(t_1, \dots, t_m) \in \mathcal{F}\}$

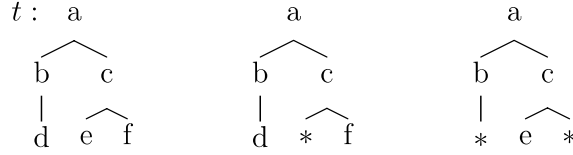
3.4.4. The gl -testable algorithm

Like the k -testable algorithm, the gl -testable algorithm has a single parameter k whose optimal value is determined by the same cross-validation method. Like the g -testable algorithm, the gl -testable algorithm divides forks in *target forks* $\mathcal{T}\mathcal{F}$ and *other forks* $\mathcal{O}\mathcal{F}$ and generalizes the other forks. However, the amount of generalization is not determined by a second parameter l , but by a more exhaustive exploration of possible generalizations.

Checking whether a (generalized) fork f is overly general is done with a test *overly-general* $(f, \mathcal{T}\mathcal{F}, \mathcal{T}\mathcal{S})$. This includes a cover test against the target forks $\mathcal{T}\mathcal{F}$ as in the g -testable algorithm but also checks that the children of f (which are also states) do not cover states containing the target x . Such states can originate from subtrees of height $k - 1$. These states (target subtrees) are collected in the set $\mathcal{T}\mathcal{S}$ which is the last argument of the procedure.

To avoid an exhaustive search over all possible generalizations of a fork, some heuristics are used. As a first heuristic, the other forks are partitioned according to their skeleton by means of a procedure *partition* (not shown). For instance, with $\mathcal{O}\mathcal{F} = \{a(d), a(c, d), a(b), a(c), a(b, c), a(c, d, e)\}$, $\text{partition}(\mathcal{O}\mathcal{F}) = \{\{a(b), a(c), a(d)\}, \{a(b, c), a(c, d)\}, \{a(c, d, e)\}\}$. Then the procedure *pgen* (also not shown) computes a single generalization of the forks in the same class (with the same skeleton): common labels are preserved but all other labels are replaced by a wildcard $*$. For instance, $\text{pgen}(\{a(b(c), d), a(b(c), e), a(b(f), e)\}) = a(b(*), *)$. If this generalization is not overly general, it is used as initial value for a search of further generalizations. Otherwise, each fork of the partition is considered for generalization. This further generalization is performed by the procedure *genl* (Algorithm 4) with as inputs a set of target subtrees, a set of target forks (both used to check against overgeneralization) and the set of forks to be generalized. This procedure returns the most general forks that are allowed by the overgeneralization check. It only considers generalizations at the bottommost level (i.e., introducing wildcards only for leaves that are at depth n , with n the height of the tree); this is a heuristic decision inspired by earlier experimental results.

Consider for instance a fork $a(b(d), c(e, f))$. It has seven possible generalizations for the labels at the bottommost level. They are partially ordered by the covers relation. The most specific ones are $a(b(*), c(e, f))$, $a(b(d), c(*, f))$ and $a(b(d), c(e, *))$ (the ones tested by *gentree* when passed the initial fork), the more general ones are $a(b(*), c(*, f))$, $a(b(*), c(e, *))$ and $a(b(d), c(*, *))$ and $a(b(*), c(*, *))$ is the most general one. The fork and two of its generalizations are shown below.



The detailed *gl*-testable procedure is shown in Algorithm 3. Lines 1–3 compute the forks and split them in $\mathcal{O}\mathcal{F}$, the other forks and $\mathcal{T}\mathcal{F}$, the target forks. The latter initialize the \mathcal{F} , the final set of forks, while the former are partitioned into \mathcal{P} (line 5). Besides the final state *FS* and the $(k - 1)$ -subtrees, the algorithm now computes also $\mathcal{T}\mathcal{S}$, the target subtrees (line 6). Then each set C in \mathcal{P} is generalized. If $\text{pgen}(\mathcal{C})$ is not overly general, it is further generalized by the *genl* procedure (line 14), otherwise, all elements of C are further generalized (line 12). The generalizations are added to the forks \mathcal{F} and states and transitions are computed as before (lines 17–19). The used heuristics were determined by experiment.

Algorithm 3. *gl*-testable.

Input: A set T of positive examples and a positive integer k

Output: A tree automaton (V, Q, Δ, FS)

- 1: $\mathcal{F}_0 := \bigcup \{f_k(t) \mid t \in T\}$
- 2: $\mathcal{T}\mathcal{F} := \{f \in \mathcal{F}_0 \mid f \text{ contains } x\}$
- 3: $\mathcal{O}\mathcal{F} := \mathcal{F}_0 - \mathcal{T}\mathcal{F}$
- 4: $\mathcal{F} = \mathcal{T}\mathcal{F}$
- 5: $\mathcal{P} = \text{partition}(\mathcal{O}\mathcal{F})$
- 6: $FS := \{r_{k-1}(t) \mid t \in T\}$
- 7: $\mathcal{S} := \bigcup \{s_{k-1}(t) \mid t \in T\}$
- 8: $\mathcal{T}\mathcal{S} = \{s \in \mathcal{S} \mid s \text{ contains } x, \text{height}(s) = k - 1\}$
- 9: **for each** $\mathcal{C} \in \mathcal{P}$ **do**
- 10: $c = \text{pgen}(\mathcal{C})$ % candidate generalization
- 11: **if** $\neg \text{overly_general} \{(c, \mathcal{T}\mathcal{F}, \mathcal{T}\mathcal{S})\}$ **then**
- 12: $\mathcal{F} = \mathcal{F} \cup \text{genl}(\mathcal{T}\mathcal{S}, \mathcal{T}\mathcal{F}, \mathcal{C})$
- 13: **else**
- 14: $\mathcal{F} = \mathcal{F} \cup \text{genl}(\mathcal{T}\mathcal{S}, \mathcal{T}\mathcal{F}, \{c\})$
- 15: **end if**
- 16: **end for**
- 17: $Q := \mathcal{S} \cup FS \cup \{r_{k-1}(f) \mid f \in \mathcal{F}\} \cup \bigcup \{s_{k-1}(f) \mid f \in \mathcal{F}\}$
- 18: $\Delta := \{(v, t_1, \dots, t_m) \rightarrow v(t_1, \dots, t_m) \mid v(t_1, \dots, t_m) \in \mathcal{S}\}$
- 19: $\Delta := \Delta \cup \{(v, t_1, \dots, t_m) \rightarrow r_{k-1}(v(t_1, \dots, t_m)) \mid v(t_1, \dots, t_m) \in \mathcal{F}\}$

Example 1. Applying Algorithm 3 on the term of Fig. 4 for $k = 3$, we obtain:

- $FS = r_2(t) = \{a(b, c)\}$, $\mathcal{O}\mathcal{F} = \{a(b(a), c)\}$, $\mathcal{T}\mathcal{F} = \{b(a(b, x))\}$ and $\mathcal{S} = s_2(t) = \{a(b, x), b, x, c\}$,
- $\mathcal{T}\mathcal{S} = \{a(b, x)\}$,
- $\mathcal{P} = \{\{a(b(a), c)\}\}$,
- $\mathcal{F} = \{a(b(*), c), b(a(b, x))\}$,
- $Q = \{a(b, c), b(a), b(*), a(b, x), b, x, c\}$,
- transitions:
 - $a(b, x) \in \mathcal{S} : (a, b, x) \rightarrow a(b, x)$,
 - $b \in \mathcal{S} : (b) \rightarrow b$,
 - $x \in \mathcal{S} : (x) \rightarrow x$,
 - $c \in \mathcal{S} : (c) \rightarrow c$,
 - $a(b(*), c) \in \mathcal{F} : (a, b(*), c) \rightarrow a(b, c)$,
 - $b(a(b, x)) \in \mathcal{F} : (b, a(b, x)) \rightarrow b(a)$.

Algorithm 4. genl.

Input: Sets $\mathcal{T}\mathcal{S}$ of target subtrees, $\mathcal{T}\mathcal{F}$ of target forks, and T of trees

Output: A set of trees G (a generalization of T)

- 1: $G := \emptyset$
- 2: **while** $T \neq \emptyset$ **do**
- 3: select t from T and remove it
- 4: $C := \{t' \mid t' \text{ is derived from } t \text{ by replacing one bottommost label } \neq * \text{ by } * \text{ and } \neg \text{overly_general}(t', \mathcal{T}\mathcal{F}, \mathcal{T}\mathcal{S})\}$
- 5: **if** $C = \emptyset$ **then**
- 6: $G := G \cup \{t\}$
- 7: **else**
- 8: $T := T - \{t \mid t \in T \text{ and } t \text{ is covered by some } c \in C\}$
- 9: $T := T \cup C$
- 10: **end if**
- 11: **end while**

Note again that the use of wildcards in the representation of the sets \mathcal{F} , Q and Δ in the example is really just an abbreviation; e.g., when Q contains $b(*)$, this really means it contains the states $b(a)$, $b(b)$, $b(c)$ and $b(x)$.

4. Experimental results

We performed two series of experiments. The first series focuses on comparing our methods on existing methods, using benchmark datasets where results for other systems are available. The second series tests the performance of our approach on larger datasets; comparison with other systems is more difficult there due to greater variation in the tasks and performance metrics used in the literature.

4.1. Test on the benchmark datasets

We evaluated our method on some semi-structured data sets commonly used in the IE research¹: a collection of webpages containing people's contact addresses (the Internet Address Finder (IAF) database) and a collection of webpages about stock quotes (the Quote Server (QS) database). There are 10 example documents in each of these datasets. The number of fields to be extracted is respectively 94 (IAF-organization), 12 (IAF-alt.name), 24 (QS-date), and 25 (QS-vol). The motivation to choose these datasets is as follows. Firstly, they are benchmark datasets that are commonly used for research in information extraction, so we can compare the results of our method directly with the results of other methods. Secondly, they are the most difficult benchmarks we are aware of that require the extraction of a whole node of the document tree. In fact, one of the authors in [40] has tried to build a handcrafted extractor given all available documents from the QS dataset and achieved only 88% accuracy.

We also test the k -testable, g -testable, and gl -testable algorithms on a small and simplified Shakespeare² data set. This dataset is a significantly reduced version of Jon Bosak's Shakespeare XML dataset, which can be found from <http://www.ibiblio.org/bosak/>; its DTD is shown in Fig. 6. In words: an act consists of a title followed by one or more scenes; a scene consists of a title followed by one or more speech acts; a speech act consists of a speaker followed by one or more lines; finally title, speaker and line are normal strings of text.

We use this dataset to test the expressiveness of our methods. The task is to extract the title of the second scene of every act, in a particular play. This extraction task is very difficult for string-based methods even on the simplified data, because each scene has a complex structure of varying length. We used the simplified Shakespeare dataset because our three algorithms above performed very poorly on the full dataset.³ The latter is a consequence of the conversion to ranked trees: an 'act' field, for instance, can have many children, many of which precede the second scene of the act. After the conversion, this second scene then ends up very deep in the subtree below the 'act' tag, making it very difficult to identify using a k -testable automaton. Thus, we expect the simplified data set to be a good example of a task that is difficult for string-based wrappers but manageable for those based on ranked trees. The full Shakespeare dataset is currently out of reach for all these methods.

The training and the testing processes follow the procedures outlined in Section 3.3. For evaluating our method, we use criteria that are commonly used in the information retrieval research community. Precision P is the number of correctly extracted objects divided by the total number of extractions, while recall R is the number of correct extractions divided by the total number of objects present in the answer template. The $F1$ -score is defined as $2PR/(P + R)$, the harmonic mean of P and R .

4.1.1. Summary of the results

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) [20], the Stalker

¹ Available from <http://www.isi.edu/~muslea/RISE/>.

² Available from <http://www.cs.kuleuven.ac.be/~ml/ie/>.

³ Also available from <http://www.cs.kuleuven.ac.be/~ml/ie/>.

```

<!ELEMENT ACT      (TITLE, SCENE+)>
<!ELEMENT SCENE   (TITLE, SPEECH+)>
<!ELEMENT SPEECH  (SPEAKER, LINE+)>
<!ELEMENT TITLE   (#PCDATA)>
<!ELEMENT SPEAKER (#PCDATA)>
<!ELEMENT LINE    (#PCDATA)>
    
```

Fig. 6. A DTD of the small Shakespeare dataset.

wrapper induction algorithm [41] and BWI [19]. We also include the results of the k -testable algorithm (as we reported in [31]) and the g -testable algorithm (as we reported in [30]). The results of HMM, Stalker and BWI are adopted from [19]. All tests are performed with standard cross-validation performed ten times following the splits used in [19], except in the small Shakespeare dataset where standard cross-validation was performed two times only. Each split has 5 documents for training and 5 for testing. We refer to the related work section for a brief description of the other methods.

Table 1 shows the results of the k -testable, g -testable and gl -testable algorithms for the optimal k -value. More specifically, k was optimized with standard cross-validation, which is performed ten times, using a separate dataset not used for training and testing.

As can be seen, our 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 algorithm performs better in the IAF-alt.name, IAF-organization and small Shakespeare data. Compared to the results of g -testable, the gl -testable performs better in the IAF-alt.name and IAF-organization data but equal in the small Shakespeare data. We shall discuss these results below.

Table 2 shows the parameters k , (k, l) and k that were used by the k -testable, g -testable, and gl -testable algorithms respectively to produce the results in Table 1. As mentioned before, these

Table 1
Comparison of the results

	<i>IAF-alt.name</i>			<i>IAF-organization</i>			<i>QS-date</i>			<i>QS-volume</i>		
	<i>P</i>	<i>R</i>	<i>F1</i>	<i>P</i>	<i>R</i>	<i>F1</i>	<i>P</i>	<i>R</i>	<i>F1</i>	<i>P</i>	<i>R</i>	<i>F1</i>
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

	<i>P</i>	<i>R</i>	<i>F1</i>
k -testable	56.2	90	69.2
g -testable	66.7	80	72.7
gl -testable	66.7	80	72.7

Table 2
Parameters used for the experiments

	<i>IAF-alt.name</i>	<i>IAF-org.</i>	<i>QS-date</i>	<i>QS-volume</i>	<i>Shakespeare</i>
<i>k</i> -testable (<i>k</i>)	4	4	2	5	3
<i>g</i> -testable (<i>k, l</i>)	(5, 2)	(5, 2)	(3, 2)	(6, 5)	(4, 2)
<i>gl</i> -testable (<i>k</i>)	4	4	2	6	4

parameters were optimized using cross-validation. A distinguishing context was discovered and used in the datasets *IAF-alt.name* and *IAF-organization*.

4.1.2. More detailed discussion of the results

In these datasets some of the best results with the *gl*-testable algorithm (i.e. in *QS-volume* and small *Shakespeare* data) are obtained with a *k*-value bigger than the value used in the *k*-testable algorithm. This means that our goal, performing more generalization while using bigger contexts, is achieved. Some other best results (i.e., in *IAF-alt.name*, *IAF-organization* and *QS-date* data) are obtained with using the same *k*-value. These results indicate that: (1) The wildcards are useful for our IE tasks as they can improve the results of the *k*-testable algorithm. (2) The two step generalization, done by the procedure *pgen* that generalizes all forks in each partition and by the procedure *genl* that searches the generalization of the bottommost labels more thoroughly, is useful for our IE tasks. This is shown by the better results of the *gl*-testable algorithm compared to the results of the *g*-testable algorithm in the two *IAF* datasets that are obtained with a smaller value of *k*.

Despite the improvements of both *gl*-testable and *g*-testable algorithms in *IAF-alt.name*, *IAF-organization* and small *Shakespeare* datasets, they were not able to improve the results of the *k*-testable algorithm in the two *QS* datasets. For the *QS-volume* data, the reason is not clear to us. One explanation is that the result might be optimal for these learners given a certain set of training examples. For the *QS-date* data, the reason is that the inferred automaton is not general enough, as can be seen in Fig. 7. In that figure, the recall of the most general automaton inferred ($k = 2$) is not very high and the maximum precision is already reached with $k = 2$. Thus we cannot improve the *FI*-score by increasing the *k*.

Fig. 7 shows how the *FI*-score of the *gl*-testable algorithm changes with *k*. The solid line is the *FI*-score, the dotted line precision and the dashed line recall. In this figure, we can clearly see the trade-off between precision and recall. The behavior of the three tree-based algorithms that we test is quite similar. With small value of *k*, the precision of these tree-based methods tend to be low because the automaton inferred is relatively general. As the value of *k* increases, the precision rises until a certain value of *k* where the maximum precision is reached. The recall of these tree-based methods behaves the other way around. At the low value of *k*, the recall of these tree-based methods tends to be high. However, as the value of *k* becomes higher, the recall decreases gradually. As the harmonic mean of the precision and recall, typically the *FI*-score curve starts with a low value at $k = 2$, increases, reaches a maximum and then starts to decrease.

Fig. 8 shows the average training time of the *k*-testable and the *gl*-testable algorithms for different values of *k*. Overall, both algorithms show somewhat similar training time on our datasets. By using a suitable data structure for looking up forks and subtrees, the *k*-testable algorithm can

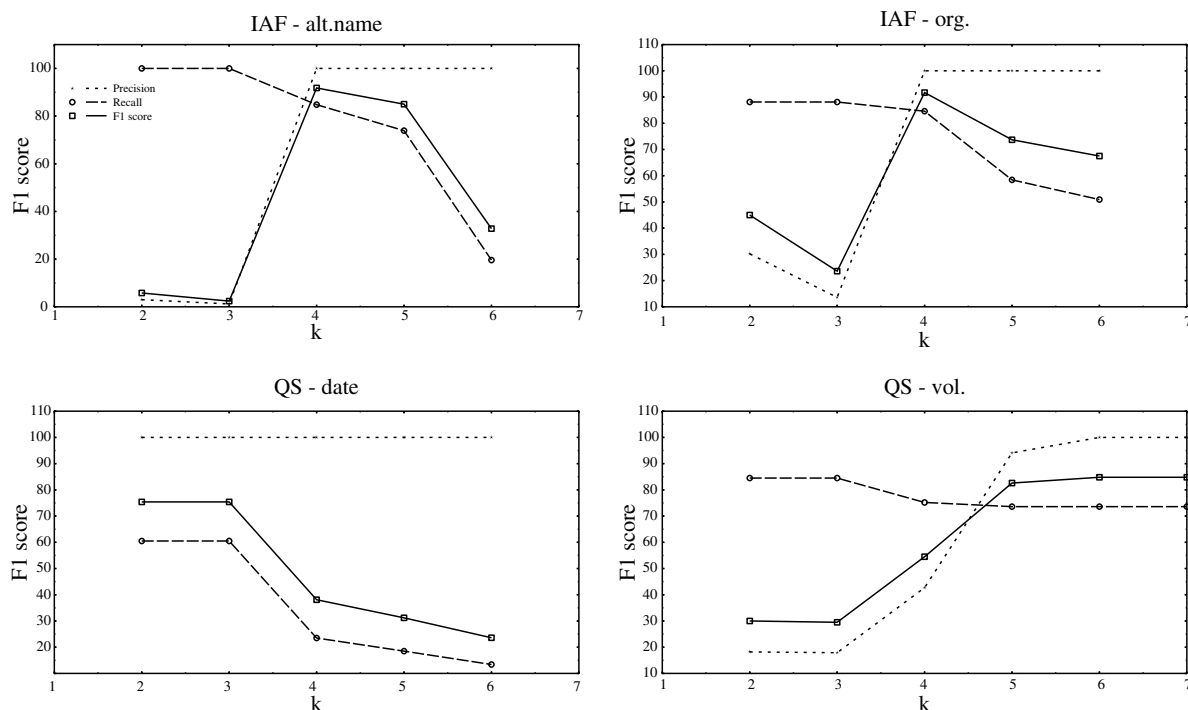


Fig. 7. The graphs of $F1$ -score versus k , for the gl -testable algorithm.

be made to run in $O(kn)$ time, where n is the total size of the examples. A similar upper bound holds for the time complexity of the g -testable algorithm. The theoretical running time of the gl -testable algorithm is exponential in the size of the subtrees, in the worst case, due to the finer search in the generalization lattice. In our experiments, the gl -testable algorithm is still feasible to run if the k -value used is less than 8. The preprocessing consists of parsing, conversion to the binary tree representation (both processes take time linear in the size of the document) and the manual insertion of the label x . Our prototype implementation was implemented in Prolog and tested on a Pentium 1.7 GHz PC. The figure shows that the actual training time (after preprocessing) needed to infer the automaton is more or less linear in k . One exception is the training time of the gl -testable algorithm for IAF-alt.name which looks non-linear. The reason is that with $k < 6$ most candidate generalizations (the result of p_{gen} function) do not suffer from overgeneralization. Thus only one candidate generalization is input to the $genl$. This is also the reason why in this task the gl -testable algorithm is slightly faster than the k -testable algorithm for $k < 6$. At $k = 6$ several candidate generalizations suffer from overgeneralization.

Actually the theoretical training time of the k -testable and the g -testable algorithms is better than that of BWI [19], one of the string-based methods that are used for comparison. The training time of BWI increases exponentially with the increase of the look-ahead parameter. As reported in [19] the IAF-alt.name, IAF-organization and QS-volume datasets need a long lookahead. They used lookaheads of 8 because these tasks need very long boundary detectors. We cannot compare the actual training time of BWI to ours in these datasets, as it was not reported.

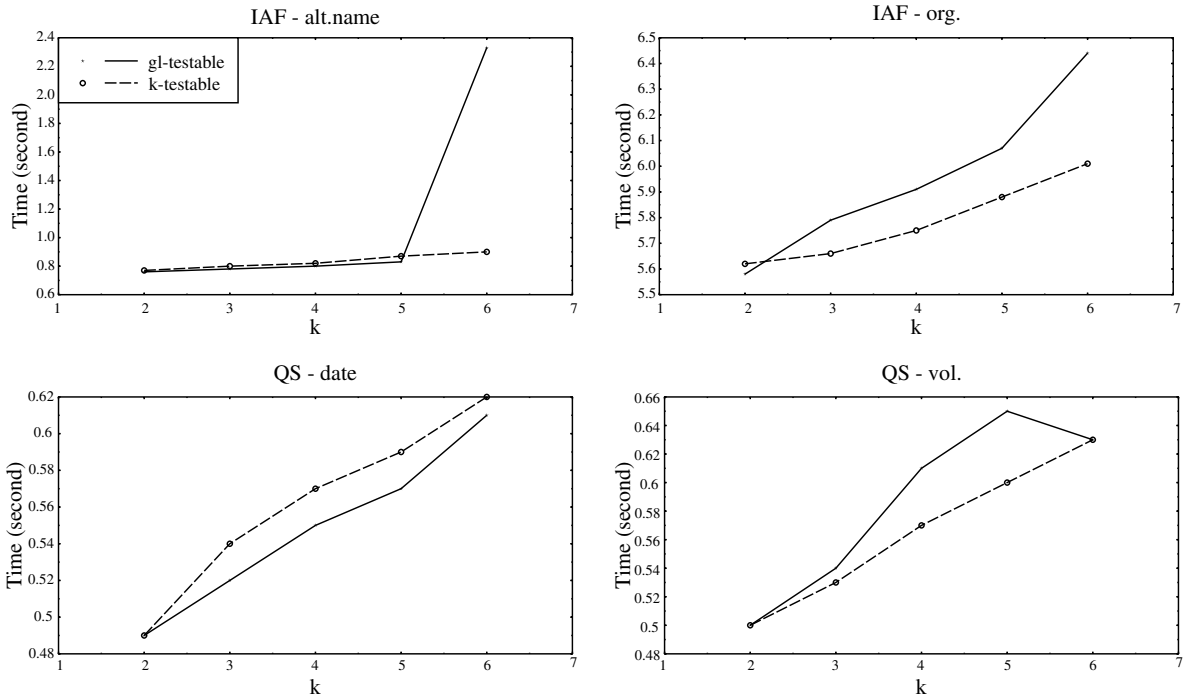


Fig. 8. The graphs of training time versus k .

Fig. 9 shows the average extraction time per document of the gl -testable algorithms for different values of k . The theoretical time complexity of the extraction procedure is $O(n^2)$ where n is the number of nodes in the document. Indeed, the time of a single run is linear in the number of nodes (using suitable data structures), while the automaton has to run for each replacement of a node by the target symbol x . Just for comparison, the extraction time of the tree automaton inferred by the k -testable algorithm (not shown here) is about two times faster than the extraction time of the generalized automaton inferred by the gl -testable algorithm. The reason is that the latter automaton needs additional time to match the wildcards.

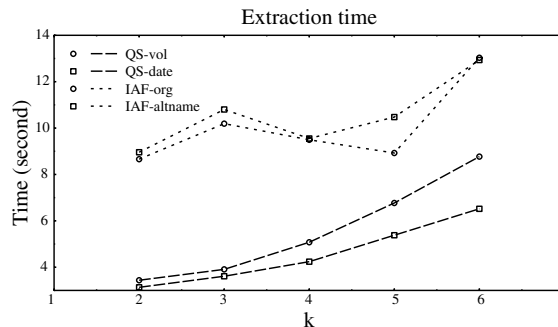


Fig. 9. The graph of extraction time versus k , for the gl -testable algorithm.

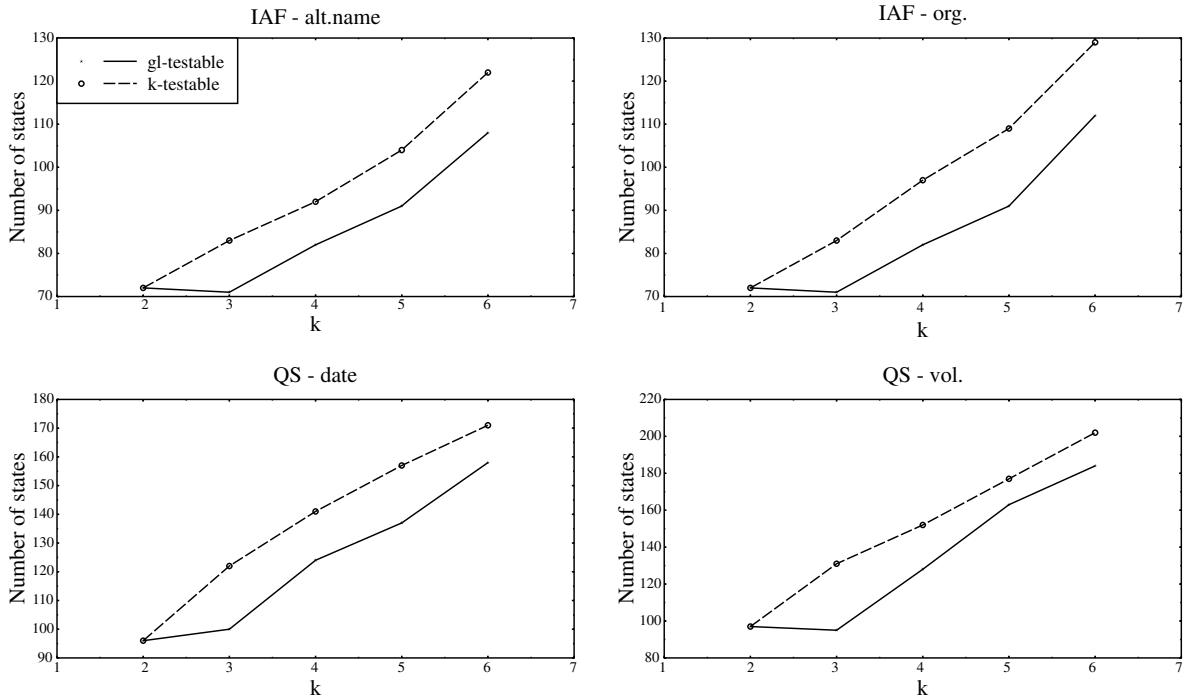


Fig. 10. The graphs of the number of states versus k .

Fig. 10 shows the number of states inferred by the k -testable and gl -testable algorithms for different values of k . As we can see from the figure, the number of states inferred by the gl -testable algorithm is always smaller than the number of states inferred by the k -testable algorithm for $k > 2$. For $k = 2$, the number of states inferred is equal as the gl -testable algorithm performs no generalization in this case.

4.2. Test on larger datasets

The previous experiments were on benchmark datasets, which made a more precise comparison with other systems possible but did not allow us to explore the limitations of our approach with respect to, for instance, the size of the database, or the computational complexity of the method.

In this second series of experiments, we test the gl -testable algorithm on larger datasets. For the experiments below we use the Bigbook and the Okra datasets that are also available online from RISE.⁴ In the Bigbook dataset we train the automaton to extract the 'name' and 'address' fields, and in the Okra dataset we train the automaton to extract the 'name' and 'email' fields. The Bigbook and Okra datasets contains 235 and 252 files respectively. The number of the name and address fields to be extracted from the Bigbook dataset is 4299 while the number of the name and email fields in the Okra datasets is 3334.

⁴ <http://www.isi.edu/~muslea/RISE/>.

All experiments in this section are done with ten-fold cross-validation and the experimental setting is as follows. We divide both datasets in two parts.

The first part, consisting of 10 files (or documents), is used to test the generalization ability of the *gl*-testable algorithm. From these 10 files we obtain 184 and 120 examples from the Bigbook and Okra datasets respectively. First we determine for each extraction task the optimal *k* by cross-validation in one random fold of training and test set. The best *k* found is 5 for the tasks in the Bigbook dataset and 4 for the tasks in the Okra dataset. Then, from the same set of 184 (120) examples, we take randomly ten examples, then twenty examples, then thirty, and so on, and give them as training examples while the rest is for testing. This process is performed ten times for every extraction task. We stop when the induced automaton has an average *FI*-score of 98% or better on the test examples. The fourth column of Table 3 shows the number of examples needed for good generalization. We can see that the number of examples needed to learn a good wrapper for these datasets is quite small.

In the ten documents of the Bigbook dataset in Table 3, the training time for each document ranges from 0.11 to 0.12 s and the average testing time for each document is 11.24 s (the average number of nodes in a document ≈ 513). In the ten documents of the Okra dataset in Table 3, the training time for each document ranges from 0.26 to 0.54 s and the testing time for each document varies from 0.08 to 38.67 s (the average number of nodes in a document ranges from 66 to 1299 respectively).

The second part of the experiment is to test the quality of the learned automata on unseen examples. The remaining data in the second part consists of 225 (=235 – 10) and 242 (=252 – 10) documents from the Bigbook and Okra datasets respectively. Using the learned tree automaton from the first part of the experiments, we perform extraction on the remaining documents in the data set. Note that none of these documents was used during the learning. Table 4 shows the results. The results for the Okra-name and Okra-email datasets are very good consid-

Table 3
The number of examples needed for good generalization

	<i>k</i>	#documents	#fields	#examples	<i>P</i>	<i>R</i>	<i>FI</i>
Bigbook-name	5	10	184	40	100	98.7	99.3
Bigbook-address	5	10	184	40	100	99.1	99.5
Okra-name	4	10	120	10	100	100	100
Okra-email	4	10	120	10	100	100	100

Table 4
The test on the rest of the larger datasets

	<i>k</i>	#documents	#fields	<i>P</i>	<i>R</i>	<i>FI</i>
Bigbook-name	5	225	4115	100	70.5	82.7
Bigbook-address	5	225	4115	100	71.7	83.5
Okra-name	4	242	3214	100	97	98.5
Okra-email	4	242	3214	100	97	98.5

Table 5
Comparison to the other IE systems on the larger datasets

	<i>Bigbook (4 fields)</i>		<i>Okra (6 fields)</i>	
	<i>R (%)</i>	<i>#Examples</i>	<i>R (%)</i>	<i>#Examples</i>
Stalker	97	8	97	1
WIEN	100	274	100	46
SoftMealy	100	6	100	1
WL ²	100	6	100	1
<i>gl</i> -testable	71.7 (2 fields)	40	97 (2 fields)	10

ering that it used only 10 examples for learning. However, the results for the Bigbook-name and Bigbook-address datasets are not as good. The reason is that the tree automaton is sensitive to the small variability in the document tree, even after generalization. In the Bigbook data, there is an index in every page that enables the user to ‘jump to’ the first company name beginning with a certain letter. This index sometimes contains full links but sometimes only partial links. The variability in this bottom part of the document is creating new states that were not seen before by the tree automaton, causing failure of the extraction task.

These results cannot be compared rigorously with any results published in the literature, due to differences in experimental setup. We nevertheless include some of those results, to give some idea of the relative quality of our results.

Hsu and Chang [26] list the performance of Stalker, WIEN, and SoftMealy systems on these datasets as shown in Table 5. Their setup differs from ours in that above wrapper systems extract six fields from the Okra dataset and four fields from the Bigbook dataset, while our system was only tested on two fields from each dataset. Due to other differences in their setup, what they report as “accuracy” corresponds to recall in our experiments. In the Bigbook dataset, Stalker [41] achieves 97% recall with 8 examples, WIEN [32] achieves 100% recall using an average of 15 documents containing approximately 274 examples, and SoftMealy [26] achieves 100% recall given 6 examples. In the Okra dataset, Stalker achieves 97% recall with only 1 example, WIEN achieves 100% recall using an average of 3.5 documents containing approximately 46 examples, and SoftMealy achieves 100% recall given 1 example.

Recently [12], a novel approach called WL² was compared to WIEN and Stalker. We include the results of WL² on the Bigbook and Okra datasets in Table 5. WL² has also been tested on the IAF and QS datasets and is able to extract all four tasks in the IAF and QS datasets with 100% recall. We should note that WL² is not string-based or tree-based, like the other approaches mentioned here, but uses a variety of representations. This comparison makes clear that the *gl*-testable algorithm needs more examples to learn from than the string-based methods we compare with. This is not unexpected: our tree-based methods search a larger hypothesis space, looking also for patterns further away from the field to be extracted, whereas the other methods look for patterns that narrowly enclose this field. As our methods consider more possible hypotheses, they need more examples to eliminate the incorrect ones. This is an example of the well-known bias-variance trade-off: a larger hypothesis space decreases bias but increases variance.

5. Related work

A lot of methods have been used for IE problems. Many are described in [39,51,33]. As mentioned above, the work on IE can be classified into three main categories: IE from unstructured texts, IE from semi-structured texts and IE from structured texts. Within each of these categories, the work can be further divided into manually built systems and (semi-)automatic systems. Within the domain of extraction from structured documents, the work on IE can be divided into:

- *Manual systems.* Examples of manually built systems can be found in [3,24]. They apply knowledge engineering techniques for building wrappers. Manually building a wrapper for each data source becomes infeasible when confronted with the variety of Web sources. A separate area aims at the development of query languages for HTML/XML, e.g. [5,57]. While these query languages are suitable for expressing complex extraction problem, their use remains time consuming and requires non-trivial skill. The advantage over other manual approaches is that they provide the user with a sophisticated user interface that simplifies the wrapper specification process as the user neither need the ability to program nor to know the HTML syntax. Most of the work in this area originated from the database community, other work originated from the document and logic programming communities. Some recent systems developed in this area are W4F [47], XWrap [36], and Lixto [4]. Related to this work is the large body of work on Web information integration. This addresses the problem of integrating heterogeneous data on the Web with the purpose of allowing users to pose queries to these integrated data. A typical information integration system consist of: wrappers that transform data from the original sources into a form that can be further processed by the system, a mediator consisting of a query planner and an execution engine, and a user interface for entering queries. Some examples of information integration systems are: Tsimmis [9], and Jedi [28].
- *Semi-automatic systems.* Our work is situated in the domain of semi-automatically built systems for IE from structured documents. Such systems make use of machine learning and data mining techniques, as well as other algorithms. The process is known as *wrapper induction*. With the recent attention for the Web, this line of work has been more popular than the work on manually-built IE system. It can be noted, that some wrapper induction systems (not ours) are also able to work on semi-structured data and even on unstructured texts.
- *Automatic systems.* We classify an IE system as an automatically-built system if the wrapper is built only once and can be used for new extraction tasks directly, or if wrappers can be built for each new task using unsupervised training only. Some examples of the IE systems in this category are as follows. WHIRL [11] is a ‘soft’ logic system that incorporates a notion of textual similarity developed in the information retrieval community. WHIRL has been used to implement some heuristics that are useful for IE [13]. Hemnani and Bressan [25] proposed a tree alignment algorithm that are based on two heuristics for extracting multiple record Web documents. IEPAD [8] is a system that automatically discovers extraction rules for identifying record boundaries from webpages.

Other work, e.g. [54,42], on Web structure mining aims at finding structural similarity between webpages. It is known as schema discovery and DTD inference and is different in nature from ours. They aim at mining the frequent or common structure of webpages. The problem that we

are facing goes beyond finding the common structure of Web documents. We also try to find the pattern of the field to be extracted inside the common document structure.

In what follows, we restrict our attention to work on wrapper induction techniques that, similar to our work, use machine learning or data mining techniques for IE from structured texts. We refer to other articles for reviews of classical IE and the issues of IE from unstructured [6,15] or semi-structured [39,51,33] texts.

The term wrapper induction was first introduced in [34]. As mentioned in the introduction, much work on wrapper induction learns wrappers based on regular expressions. BWI [19] is a boosting approach in which the weak learner learns a simple regular expression with high precision but low recall. The HMM approach reported in Table 1 was proposed by Freitag and McCallum [20]. They learn a hidden Markov model, solving the problem of estimating probabilities from sparse data using a statistical technique called shrinkage. This model has been shown to achieve state-of-the-art performance on a range of IE tasks. The Stalker algorithm [41] induces extraction rules that are expressed as simple landmark grammars. The latter are a class of finite automata. Stalker performs hierarchical extraction guided by an *embedded catalog* tree. This tree describes the structure of the fields to be extracted from the documents.

Freitag [16] describes several techniques based on naive-Bayes, two regular language inference algorithms, and their combinations for IE from unstructured texts. His results demonstrate that the combination of grammatical inference techniques with naive-Bayes improves the precision and accuracy of the extraction. WHISK [51] is a system that learns extraction rules with a top-down rule induction technique. The extraction rules of WHISK are based on a kind of regular expression patterns. To make the rules more powerful, WHISK has some built-in semantic classes and in addition allows for user-defined semantic classes. A semantic class is basically a set of terms that are considered to be equivalent. Chidlovskii et al. [10] describe an incremental grammar induction approach; their language is based on a subclass of deterministic finite automata that do not contain cyclic patterns. Hsu and Dung's SoftMealy system [27] learns separators that identify the boundaries of the fields of interest. These separators are described by strings of fixed height in which each symbol is an element of a taxonomy of tokens (with fixed strings on the lowest level and concepts such as punctuation or word at higher levels). Hsu and Chang [26] propose two classes of SoftMealy extractors: single-pass, which is biased for tabular documents such as the QS dataset, and multi-pass, which is biased for tagged-list document such as the IAF dataset. Although their systems were tested on the same datasets as ours, their results cannot be compared directly because the experimental setting is different. Their evaluation gives only numbers for recall and uses a different set of examples.

The above mentioned methods learn string languages while our method learns a more expressive tree language. Compared to HMMs and BWI our method does not require the manual specification of the windows height for the prefix, suffix and the target fragments. Compared to Stalker and BWI our method does not require the manual specification of the special tokens or landmarks such as ">" or ",". Compared to SoftMealy extractors in [26] our method is generally applicable to any type of document formatting without requiring different classes of wrappers for different categories of documents.

Despite the above advantages, our method also has some limitations. A first one is that our method only outputs a whole node. This seems to limit its applicability. For data-centric documents such as XML documents, this is not really the case since the data to be extracted is typically

a whole node. However, it is true for HTML formatted documents. One way to broaden the applicability of our method is to perform a two step extraction. A whole node of the tree can be extracted in a first step while a second step (using other techniques) can post-process the selected information to extract a part of it. A second limitation is that our method works only on structured documents. Indeed our method cannot be used for text-based IE, and is not intended for it. A third limitation is that our method is possibly slower (when extracting) than string-based methods because it has to parse the document tree and has to substitute each node with x when extracting information from the document. Despite these limitations, our results suggest that our method works better in the four structured domains than the more generally applicable string-based IE methods.

Some other approaches that exploit the structure of the documents have been proposed. WL² [12], a logic-based wrapper learner that uses multiple (string, tree, visual, and geometric) representations of the HTML documents, consists of one master learner and several specific builders that are created for specific page formats. The learning method is an inductive logic programming algorithm similar to FOIL [45]. In fact, WL² is able to extract all four tasks in the IAF and QS datasets with 100% recall. The work of WL² suggests that indeed using task-specific document representation can yield a much better performance. Sakamoto et al. [50] propose a certain class of wrappers that use the tree structure of HTML documents and propose an algorithm for inducing such wrappers. They identify a field with a path from root to leaf, imposing conditions on each node in the path that relate to its label and its relative position among siblings with the same label (e.g., “second child with label ⟨B⟩”). Their hypothesis language corresponds to a subset of tree automata.

Besides the k -testable algorithm proposed in this paper, we have also experimented with Sakakibara’s reversible tree algorithm [48]. Preliminary results with this algorithm suggested that it generalizes insufficiently on our data sets. Hence, we did not further pursue its use.

6. Conclusion

We have explored induction of ranked tree automata for information extraction from Web documents. Our main findings and contributions are as follows.

We have argued for the use of tree automata, instead of string automata, for wrapper induction for Web documents. Our main argument is that string based methods cannot exploit the tree structure of these documents. We have conjectured that exploiting this tree structure should make it easier to correctly extract information.

Focusing on the induction of ranked tree automata, a subset of tree automata for which algorithms have been presented before, we have described an approach based on the existing k -testable algorithm. Several versions of a tree induction algorithm, together with a preprocessing procedure, have been presented.

Experimental results confirm our conjecture that exploiting the tree structure is often advantageous: on several benchmark problems, our methods perform better than string-based methods. But the results also indicate limitations of the approach. First, the conversion of unranked trees into ranked trees, which is part of the preprocessing phase, and necessary to apply ranked tree induction algorithms, destroys the tree structure to some extent. Second, by considering a more

global context for identification of the field to be extracted (i.e., a subtree of the document tree, instead of delimiters of the field as string-based methods typically do), the method becomes more sensitive to changes in non-relevant parts of the documents, which causes it to work less well on some data sets.

We have looked only into variants of the k -testable algorithm for induction of ranked tree automata. Other algorithms are conceivable for induction of ranked tree automata. However, all of these will suffer from the first limitation mentioned above. Instead of studying other algorithms for induction of ranked tree automata, a more promising direction is to study algorithms for induction of unranked tree automata. Early results on the latter, presented in another paper [29], confirm this impression.

Further limitations of the tree automata induction approach in general, are that one can extract only a whole node, and only single-field extraction can be performed. The first limitation can be removed by combining string-based methods with tree-based methods, that is, one could identify a node in the document tree using a tree automaton, then extract the relevant field from this node using a string automaton. The second limitation is somewhat more difficult to resolve in a principled manner. Both issues need further research.

A last direction that seems interesting to explore is to incorporate probabilistic information in the inference process; this could compensate for the lack of negative examples.

Acknowledgements

We thank Nicholas Kushmerick for providing us with the datasets used for the BWI experiments. This work is supported by the FWO project query languages for data mining. Hendrik Blockeel is a post-doctoral fellow of the Fund for Scientific Research of Flanders.

References

- [1] D. Angluin, Queries and concept learning, *Machine Learning* 2 (4) (1988) 319–342.
- [2] D. Angluin, C.H. Smith, Inductive inference: Theory and methods, *ACM Computing Surveys* 15 (3) (1983) 237–269.
- [3] P. Atzeni, G. Mecca, Cut & paste, in: *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ACM Press, 1997, pp. 144–153.
- [4] R. Baumgartner, S. Flesca, G. Gottlob, Visual web information extraction with lixto, in: *Proceedings of 27th international conference on very large data bases (VLDB 2001)*, 2001, pp. 119–128.
- [5] F. Bry, S. Schaffert, Towards a declarative query and transformation language for XML and semistructured data: Simulation unification, in: *Proceedings of the international conference on logic programming*, 2002.
- [6] C. Cardie, Empirical methods in information extraction, *AI Magazine* 18 (4) (1997) 65–79.
- [7] R.C. Carrasco, J. Oncina, J. Calera-Rubio, Stochastic inference of regular tree languages, in: *Proceedings of the 3rd International Colloquium on Grammatical Inference, Lecture Notes on Artificial Intelligence*, vol. 1433, 1998, pp. 187–198.
- [8] C.-H. Chang, S.-C. Lui, IEPAD: Information extraction based on pattern discovery, in: *Proceedings of the Tenth International Conference on World Wide Web*, 2001, pp. 681–688.
- [9] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J. Widom, The TSIMMIS project: Integration of heterogeneous information sources, in: *Proceedings of the 10th Meeting of the Information Processing Society of Japan*, 1994, pp. 7–18.

- [10] B. Chidlovskii, J. Ragetli, M. de Rijke, Wrapper generation via grammar induction, in: 11th European Conference on Machine Learning, ECML'00, 2000, pp. 96–108.
- [11] W. Cohen, Whirl: A word-based information representation language, *Artificial Intelligence* 118 (2000) 163–196.
- [12] W. Cohen, M. Hurst, L.S. Jensen, A flexible learning system for wrapping tables and lists in HTML documents, in: The Eleventh International World Wide Web Conference (WWW2002), 2002.
- [13] W.W. Cohen, Recognizing structure in web pages using similarity queries, in: Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, 1999, pp. 59–66.
- [14] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree Automata Techniques and Applications, Available from: <<http://www.grappa.univ-lille3.fr/tata>>, 1999.
- [15] J. Cowie, W. Lehnert, Information extraction, *Communications of the ACM* 39 (1) (1996) 80–91.
- [16] D. Freitag, Using grammatical inference to improve precision in information extraction, in: ICML-97 Workshop on Automata Induction, Grammatical Inference, and Language Acquisition, 1997.
- [17] D. Freitag, Information extraction from HTML: Application of a general learning approach, in: Proceedings of the Fifteenth Conference on Artificial Intelligence AAAI-98, 1998. pp. 517–523.
- [18] D. Freitag, Machine learning for information extraction in informal domains, *Machine Learning* 39 (2/3) (2000) 169–202.
- [19] D. Freitag, N. Kushmerick, Boosted wrapper induction, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Innovative Applications of AI Conference, AAAI Press, 2000, pp. 577–583.
- [20] D. Freitag, A. McCallum, Information extraction with HMMs and shrinkage, in: AAAI-99 Workshop on Machine Learning for Information Extraction, 1999.
- [21] P. Garcia, Learning k -testable tree sets from positive data, Technical Report DSIC-ii-1993-46, DSIC, Universidad Politecnica de Valencia, 1993.
- [22] E.M. Gold, Language identification in the limit, *Information and Control* 10 (5) (1967) 447–474.
- [23] G. Gottlob, K. Koch, Monadic datalog over trees and the expressive power of languages for web information extraction, in: 21st ACM Symposium on Principles of Database Systems, 2002, pp. 17–28.
- [24] J. Hammer, H. Garcia-Molina, J. Cho, A. Crespo, R. Aranha, Extracting semistructured information from the Web, in: Proceedings of the Workshop on Management of Semistructured Data, 1997, pp. 18–25.
- [25] A. Hemnani, S. Bressan, Information extraction-tree alignment approach to pattern discovery in web documents, in: Database and Expert Systems Applications, 13th International Conference, DEXA 2002, 2002, pp. 789–798.
- [26] C.-N. Hsu, C.-C. Chang, Finite-state transducers for semi-structured text mining, in: Proceedings of IJCAI-99 Workshop on Text Mining: Foundations, Techniques and Applications, 1999.
- [27] C.-N. Hsu, M.-T. Dung, Generating finite-state transducers for semi-structured data extraction from the web, *Information Systems* 23 (8) (1998) 521–538.
- [28] G. Huck, P. Fankhauser, K. Aberer, E.J. Neuhold, Jedi: Extracting and synthesizing information from the web, in: Conference on Cooperative Information Systems, 1998, pp. 32–43.
- [29] R. Kosala, M. Bruynooghe, H. Blockeel, J.V. den Bussche, Information extraction from web documents based on local unranked tree automaton inference, in: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 2003, pp. 403–408. <http://www.cs.kuleuven.ac.be/cgi-bin/dtai/publ_info.pl?id=40757>.
- [30] R. Kosala, M. Bruynooghe, H. Blockeel, J. Van den Bussche, 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 and Services (IIWAS), 2002, pp. 105–109.
- [31] R. Kosala, J. Van den Bussche, M. Bruynooghe, H. Blockeel, 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), 2002, pp. 299–310.
- [32] N. Kushmerick, Wrapper induction for information extraction, Ph.D. thesis, University of Washington, 1997.
- [33] N. Kushmerick, Wrapper induction: Efficiency and expressiveness, *Artificial Intelligence* 118 (2000) 15–68.
- [34] N. Kushmerick, D. Weld, R. Doorenbos, Wrapper induction for information extraction, in: Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-97, 1997, pp. 729–737.
- [35] A. Levy, C. Knoblock, S. Minton, W. Cohen, Trends and controversies: Information integration, *IEEE Intelligent Systems* 13 (5) (1998).

- [36] L. Liu, C. Pu, W. Han, Xwrap: An XML-enabled wrapper construction system for web information sources, in: Proceedings of the 16th International Conference on Data Engineering, IEEE Computer Society, 2000, pp. 611–621.
- [37] MUC-6, Proceedings of the Sixth Message Understanding Conference, Morgan Kaufmann, San Francisco, CA, 1995.
- [38] K. Murphy, Learning finite automata, Technical Report 96-04-017, Santa Fe Institute, 1996.
- [39] I. Muslea, Extraction patterns for information extraction tasks: A survey, in: AAAI-99 Workshop on Machine Learning for Information Extraction, 1999.
- [40] I. Muslea, S. Minton, C. Knoblock, A hierarchical approach to wrapper induction, in: Proceedings of the 3rd International Conference on Autonomous Agents, 1999.
- [41] I. Muslea, S. Minton, C. Knoblock, Hierarchical wrapper induction for semistructured information sources, Journal of Autonomous Agents and Multi-Agent Systems 4 (2001) 93–114.
- [42] S. Nestorov, S. Abiteboul, R. Motwani, Inferring structure in semistructured data, SIGMOD Record 26 (4) (1997).
- [43] C. Pair, A. Quere, Définition et étude des langages réguliers, Information and Control 13 (6) (1968) 565–593.
- [44] R. Parekh, V. Honavar, Automata Induction, Grammar Inference, and Language Acquisition, Handbook of Natural Language Processing, Marcel Dekker, New York, 1998.
- [45] J.R. Quinlan, Learning logical definitions from relations, Machine Learning 5 (1990) 239–266.
- [46] J. Rico-Juan, J. Calera-Rubio, R. Carrasco, Probabilistic k -testable tree languages, in: A. Oliveira (Ed.), Proceedings of 5th International Colloquium, ICGI 2000, Lisbon (Portugal), Lecture Notes in Computer Science, vol. 1891, Springer, 2000, pp. 221–228.
- [47] A. Sahuguet, F. Azavant, Looking at the web through XML glasses, in: Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems, 1999, pp. 148–159.
- [48] Y. Sakakibara, Efficient learning of context-free grammars from positive structural examples, Information and Computation 97 (1) (1992) 23–60.
- [49] Y. Sakakibara, Recent advances of grammatical inference, Theoretical Computer Science 185 (1) (1997) 15–45.
- [50] H. Sakamoto, H. Arimura, S. Arikawa, Knowledge discovery from semistructured texts, in: S. Arikawa, A. Shinohara (Eds.), Progress in Discovery Science—Final Report of the Japanese Discovery Science Project, LNAI, vol. 2281, Springer, 2002, pp. 586–599.
- [51] S. Soderland, Learning information extraction rules for semi-structured and free text, Machine Learning 34 (1–3) (1999) 233–272.
- [52] M. Takahashi, Generalizations of regular sets and their application to a study of context-free languages, Information and Control 27 (1975) 1–36.
- [53] L. Valiant, A theory of the learnable, Communications of the ACM 27 (11) (1984) 1134–1142.
- [54] K. Wang, H. Liu, Discovering structural association of semistructured data, IEEE Transactions on Knowledge and Data Engineering 12 (3) (2000) 353–371.
- [55] G. Wiederhold, Intelligent Information Integration, Kluwer, 1996.
- [56] XML. Extensible markup language (XML) 1.0, second ed., W3C Recommendation 6 October 2000, Available from: <<http://www.w3.org>>, 2000.
- [57] XQL. XQuery 1.0: An XML query language. W3C Working Draft 16 August 2002, Available from: <<http://www.w3.org/TR/xquery>>.



Raymond Kosala obtained a B.Sc. degree in Electronics Engineering from the University of Atma Jaya, Indonesia (1994), Master in Artificial Intelligence (1999) and Ph.D. in Informatics (2003) degrees from the Katholieke Universiteit Leuven. He is currently a faculty member in the Computer Science Department of the Bina Nusantara University, Indonesia. His research interests include Data Mining and Machine Learning.



Hendrik Blockeel obtained the degrees of Engineering in Computer Science (1993), Master of Artificial Intelligence (1994), and a Ph.D. in Applied Sciences (1998) from the Katholieke Universiteit Leuven. He currently leads a machine learning research group at the same University. His research interests are in machine learning and data mining, with a focus on learning from structured and relational data.



Maurice Bruynooghe obtained a degree of Engineer in Computer Science at the K.U. Leuven. He received a Ph.D. from the same University in 1979; the subject of the thesis was logic programming. He started a logic programming group at the K.U. Leuven that later became the group Declarative Languages and Artificial Intelligence of which he is still the head. His research interests covers the three research areas of the group: Design, Analysis and Implementation of Declarative Programming Languages, Knowledge Representation and Reasoning and Machine Learning and Data Mining. From 1991 to 2000 he was Editor-in-Chief of the Journal of Logic Programming, and from 2001 to 2005 of Theory and Practice of Logic Programming, the low budget successor of JLP. He is a fellow of ECCAI, the European Coordinating Committee for Artificial Intelligence.



Jan Van den Bussche received his Ph.D. from the University of Antwerp in 1993, under the advising of Jan Paredaens, and became a faculty member at the Limburg University Centre in 1996, where he is now professor of theoretical computer science. His research interests are in databases, broadly construed.