Made available by Hasselt University Library in https://documentserver.uhasselt.be

Learning (k, l)-contextual tree languages for information extraction from web pages

Non Peer-reviewed author version

Raeymaekers, Stefan; Bruynooghe, Maurice & VAN DEN BUSSCHE, Jan (2008) Learning (k, l)-contextual tree languages for information extraction from web pages. In: MACHINE LEARNING, 71(2-3). p. 155-183.

Handle: http://hdl.handle.net/1942/8318

Learning (k,l)-contextual tree languages for information extraction from web pages

Stefan Raeymaekers[1], Maurice Bruynooghe[1], and Jan Van den Bussche[2]

[1] K.U.Leuven, Dept. of Computer Science, Celestijnenlaan 200A, 3001 Leuven, Belgium
{stefanr,maurice}@cs.kuleuven.ac.be
[2] Universiteit Hasselt and Transnationale Universiteit Limburg, Agoralaan D, 3590 Diepenbeek, Belgium jan.vandenbussche@uhasselt.be

Abstract. This paper introduces a novel method for learning a wrapper for extraction of information from web pages, based upon (k, l)-contextual tree languages. It also introduces a method to learn good values of k and l based on a few positive and negative examples. Finally, it describes how the algorithm can be integrated in a tool for information extraction.

1 Introduction

The World Wide Web is an indispensable source of information. Extracting its content for further processing, however, is difficult because it is formatted in HTML, which is primarily focused on presentation. A wrapper is a general name for a procedure that extracts data (often from machine generated HTML-pages) based on the structure of the documents, commonly without the use of linguistic knowledge. Various tools have been designed to facilitate wrapper building, but the process remains tedious. Hence the efforts [4–9, 14, 16–19, 22, 30] to create algorithms that learn wrappers from examples.

Several approaches [4, 6–9, 14, 19, 22, 30] process documents in a string representation. HTML-code is encoded as a sequence of text and tags. But implicitly the tags define a tree structure on the document. Flattening the tree structure of the document to a string representation though can project sibling nodes arbitrarily far from one another. Take for example a node with two children. During flattening, all the descendants of the first child (the first child might be the root of a large subtree), will be put between the first child and the second, hiding the sibling relation between the two. Learning a wrapper that can express relations between such distant nodes turns out to be a quite difficult task. To improve locality of the neighborhood nodes of the target nodes, [16, 18] represent documents as (ranked [3]) binary trees; they show that this indeed improves the quality of the results. In [17], the same authors further exploit the idea that a representation where the nodes in the neighborhood of the target node are close to each other, makes the extraction task easier and introduce an unranked tree representation. Combined with some ad-hoc design decisions that compensate for some relevant context information that is not captured by their approach, they indeed obtain superior results.

[3] In ranked trees, the number of children of a node is fi xed and a function of its label. Nodes in unranked trees can have an arbitrary number of children.

One limitation of these tree based methods is that they extract whole nodes (or subtrees) in the document; most string-based approaches, on the contrary, can extract a substring of a text node. This limitation is easy to overcome. Indeed, if a task needs sub-node extraction, it is very natural to learn in a first step a wrapper that retrieves the relevant node(s), and in a second step another wrapper that extracts the required information from the text in the retrieved node(s). This text is much smaller than the text corresponding to the whole document and hence this second task much better fits the capabilities of existing string based approaches.

The current paper further explores the idea of using an unranked tree as a representation of the document. Its contributions are as follows:

– The introduction of the notion of a (k,l)-contextual tree language for unranked trees and an algorithm to infer such a language from positive examples (trees) only. A major virtue is that this algorithm needs very few examples to learn. This algorithm is then applied on marked trees to induce wrappers. We obtain better results than [17] while avoiding its ad-hoc design decisions.
– Whereas [17] needed cross validation to learn the parameters (i.e., a fully annotated data set), we introduce a method to learn the parameters with only a few negative examples (Section 6).
– We integrate our results into an interactive system that guides the user in building a wrapper by posing equivalence queries. For example, if a user wants to extract book prices from www.amazon.com, he clicks on an example page (in the browser of the GUI-front end) on one or more prices. The algorithm then learns a wrapper from these (positive only) examples and highlights all elements that are extracted by this wrapper. When the current hypothesis is erroneous, the user can either click on a highlighted item to indicate it as a false positive or click on an item that is not yet highlighted to indicate that it is a false negative. The application then adjusts the wrapper. This interaction continues (possibly with other example pages), until the user is satisfied.

A preliminary version of this paper appeared as [27].

We have organized the paper as follows. We describe the application of information extraction from semi-structured data in more detail and present some running examples in Section 2. In Section 3, we introduce and discuss the (k,l)-contextual tree languages, a subclass of the regular unranked tree languages, as a solution for learning from positive examples only. We detail the application of this new class of languages as a wrapper representation in Section 4. Section 5 contains an overview of related work, and reports on an experimental evaluation of four different wrappers. In Section 6 we provide an algorithm able to learn the parameters for the wrapper induction from a few negative examples. We present an interactive system based on the previous algorithms, and evaluate it in Section 7. We indicate possible further work in Section 8 and we conclude in Section 9.

2 Information Extraction from Semi-Structured Data

The aim of Information Extraction(IE) systems is to extract specific information from human readable documents. In free text, the information is embedded in sentences and extraction techniques are rooted in the field of Natural Language Processing. In web pages (HTML documents), all information is embedded in markup tags that indicate the document's structure and layout. Such documents are said to be semi-structured. Full sentences are often lacking, but, in a same page or a set of similar pages, the local structure around the elements of interest is often very regular, in particular when the page is generated from a table by a script. Hence it is feasible to use grammars to identify the elements of interest.

A typical information extraction task does not intend to extract information from all possible HTML documents. Its aim is to extract a certain kind of information from a set of documents where the information of interest is organized in a similar way. The set of relevant documents is called the domain of the extraction task. For example, a wrapper can be learned for extracting addresses of houses offered for rent by a particular agency. Once learned it can be applied on all house offering pages of the agency. It makes no sense to apply it on a page containing weather forecasts or on pages of another rental agency. Indeed, while the latter holds similar data, the internal representation can be completely different.

Below, we introduce as running examples, two simplified information extraction settings.

Example 1. A database of articles can be queried for articles containing a given search term. The query returns a list that contains for each selected article its title and author. This result is converted to HTML by a script and visualized in a browser. Long lists are split over multiple documents. A schematic example web page together with the corresponding HTML code is shown in Figure 1.

The domain for this setting consists of all possible pages that can be generated by the script. One extraction task could be to extract the search term of the query for which that page presents the results. Other tasks are to extract the list of titles or the list of authors. A more complicated extraction task is to extract a list of tuples, each tuple containing a title field and an author field.

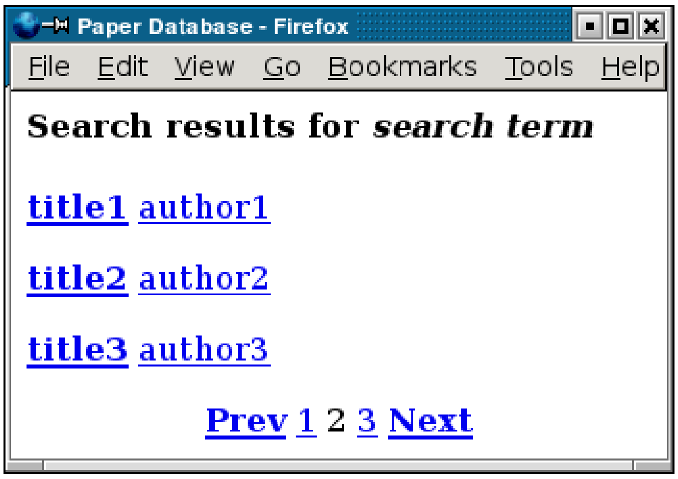
Example 2. Research groups at some university maintain a list of their PhD students. For each student, the page indicates the supervisor. An example web page together with the corresponding HTML code is shown in Figure 2.

One can define different kinds of extraction tasks. A basic distinction is between extraction of a single field and the extraction of a group of fields organized in a tuple or a more complex data structure. Moreover, a single field can correspond to a single node in the document tree, a part of a single node, or can be spread over a number of adjacent nodes. In the latter case, the field can or cannot correspond to all leaf nodes of a single subtree. In this paper, the focus is on extraction of a single field that corresponds to a single (text) node in the document tree. In Section 8, we elaborate on how our method can be a building stone in methods for more complex extraction tasks.

Wrapper learning is typically based on positive examples. A positive example consists of a page —containing one or more occurrences of the target field— where one node is marked as the target node. Other examples can come from different pages or from the same page. It is not necessary to give all the target nodes in a given page as

a)
```
<html><body>
   <h4>Search results for <i>search term</i></h4>
   <p><b><a>title1</a></b> <a>author1</a></p>
   <p><b><a>title2</a></b> <a>author2</a></p>
   <p><b><a>title3</a></b> <a>author3</a></p>
   <center>
      <b><a>Prev</a></b> <a>1</a> 2 <a>3</a> <b><a>Next</a><b>
   </center>
</body></html>
```

b)



Search results for *search term*

title1 author1

title2 author2

title3 author3

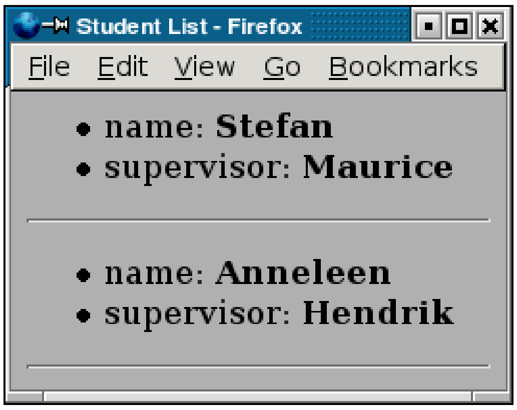
Prev 1 2 3 **Next**

Fig. 1. Paper Database (Example 1): a) HTML code; b) screen shot.

a)

```
<html><body>
   <ul><li>name: <b>Stefan</b></li>
       <li>supervisor: <b>Maurice</b></li></ul><hr>
   <ul><li>name: <b>Anneleen</b></li>
       <li>supervisor: <b>Hendrik</b></li></ul><hr>
</body></html>
```

b)



- name: **Stefan**
- supervisor: **Maurice**

- name: **Anneleen**
- supervisor: **Hendrik**

Fig. 2. Student List (Example 2), HTML code a) and screen shot b).

examples. As an example, learning a wrapper for extracting the author field in Example 2 can be based on two examples from that page, one example marking author1, the other marking author3 as the target page. When used for extraction, the learned wrapper should extract all three author fields from the page.

Some approaches expect completely annotated data. When a certain page is completely annotated, i.e., all positive examples for that page are given, we have in fact also an implicit collection of negative examples. Indeed, every node that is not marked in one of the positive examples is negative. It is tedious though to annotate pages completely. Giving a few negative examples is not a viable alternative because it is difficult to select useful negative examples as most negative examples do not affect the set of extracted fields. Therefore it is common to start from positive examples only. After the learner has inferred a hypothesis it can be used for extraction. False positives are sensible candidates for negative examples and can be exploited to refine the hypothesis iteratively.

3 (k,l)-Contextual Tree Languages

Unfortunately, the whole class of regular languages cannot be learned from positive examples only [12]. Intuitively the reason is that there is no boundary to end the generalization, and therefore the resulting language will accept everything. A common workaround is to focus on a subclass of the regular languages that is learnable from positive examples only. Examples of such subclasses of string languages are k-reversible languages [2], k-contextual languages [21] and k-testable languages [11]. The latter two are often referred to as k-local languages as they are equivalent [1]. Similar developments occurred for tree languages. Algorithms for induction of string automata have been upgraded for tree automata. Several works exist for ranked trees, e.g., [10, 15] (k-testable tree languages) and [28] (probabilistic extensions). HTML or XML documents are clearly *not* ranked; hence some encoding as ranked trees is needed in order to apply k-testable tree language learning to Web information extraction; the authors of [16, 18] use a binary encoding for this purpose. This binary encoding still distorts the structural relationships between the target and its neighborhood, as the distance between a node and its sibling can get arbitrary long.

Therefore, in this section, we introduce (k,l)-contextual tree languages, which are unranked, and therefore directly applicable. We start, in Section 3.1, with recalling some terminology about trees and introducing the concept of a (k,l)-fork. Section 3.2 introduces the notion of (k,l)-contextual tree languages and shows that a tree belongs to a particular (k,l)-contextual tree language when its (k,l)-forks belong to the so called representative set of the language. Finally, Section 3.3 shows that a (k,l)-contextual tree language is obtained by taking the (k,l)-forks in a set of positive examples as its representative set. It is also proven that the properties of (k,l)-contextual tree languages are similar to those of k-contextual string languages, namely, they are anti-monotone in the values of the parameters k and l and learnable in the limit from positive examples only.

3.1 Preliminary Definitions

We define the alphabet Σ as a finite set of symbols. The set $T(\Sigma)$ of all finite trees with nodes labeled by elements of Σ can be recursively defined as $T(\Sigma) = \{f(s) \mid f \in \Sigma, s \in T(\Sigma)^*\}$. We usually denote $f(\epsilon)$, where ϵ is the empty sequence, by f. The subtrees of a tree are inductively defined as $sub(f(t_1,\ldots,t_n)) = \{f(t_1,\ldots,t_n)\} \cup \bigcup_i sub(t_i)$. A *tree language* is any subset of $T(\Sigma)$. The set of *(k,l)-roots* of a tree $f(t_1,\ldots,t_n)$ is the singleton $\{f\}$ if l=1; otherwise, it is the set of trees obtained by extending the root f with $(k,l-1)$-roots of k successive children of t (all children if $k > n$). Formally, we have the following inductive definition. For sets $S_1, \ldots, S_n$ of trees, we use the notation $f(S_1,\ldots,S_n)$ for the set of trees $\{f(s_1,\ldots,s_n) \mid s_i \in S_i\}$.
$R_{(k,l)}(f(t_1 \ldots t_n)) =$

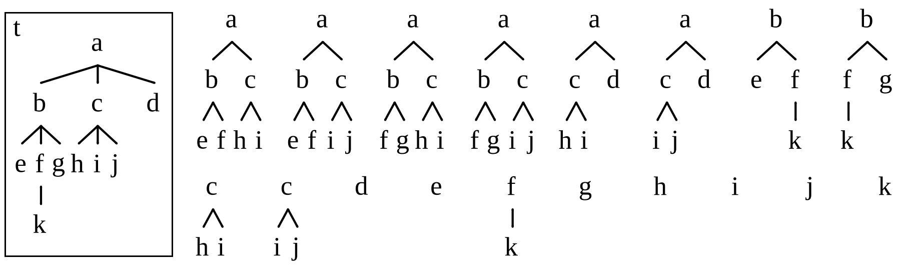
$$\begin{cases} \{f\} & \text{if } l=1 \\ f(R_{(k,l-1)}(t_1)\ldots R_{(k,l-1)}(t_n)) & \text{if } l>1 \text{ and } k>n \\ \bigcup_{p=1}^{n-k+1} f(R_{(k,l-1)}(t_p)\ldots R_{(k,l-1)}(t_{p+k-1})) & \text{otherwise.} \end{cases} .$$

As an extension, the (k,l)-roots of a set T of trees are defined as $R_{(k,l)}(T) = \bigcup_{t\in T} R_{(k,l)}(t)$. Finally, a (k,l)-*fork* of a tree t is a (k,l)-root of any subtree of t. Thus, the set of (k,l)-forks of t can be written as $R_{(k,l)}(sub(t))$ and we denote it by $F_{(k,l)}(t)$. The (k,l)-forks of a set of trees T are then defined as $F_{(k,l)}(T) = \bigcup_{t\in T} F_{(k,l)}(t)$.

Given these definitions we can state that $R_{(k,l)}(T)$ and $F_{(k,l)}(T)$ are monotone in T, or formally:

Proposition 1. *$T \supseteq T'$ implies $R_{(k,l)}(T) \supseteq R_{(k,l)}(T')$ and $F_{(k,l)}(T) \supseteq F_{(k,l)}(T')$*

Example 3. Below we show graphically the $(2,3)$-forks of a tree t. The first 6 of these forks, are the $(2,3)$-roots of t.



3.2 (k,l)-Contextual Tree Languages

Definition 1. *The (k,l)-contextual tree language based on the set G of trees is defined as $L_{(k,l)}(G) = \{t \in T(\Sigma) \mid F_{(k,l)}(t) \subseteq G\}$.*

Every given (k,l)-contextual language can be defined by an infinite number of different sets. For example adding elements to or removing elements from G with either height $> l$ or width $> k$ will not influence the definition of $L_{(k,l)}(G)$:

Proposition 2. *$L_{(k,l)}(G) = L_{(k,l)}(G \cap T_{(k,l)})$ with $T_{(k,l)}$ the set of trees with height at most l and width at most k.*

Note that $T_{(k,l)}$ is finite, so we can always assume that G is finite. But even addition or removal of trees with height $\leq l$ and width $\leq k$ will not always influence the definition of $L_{(k,l)}(G)$. The following proposition shows that there is always a unique smallest G:

Proposition 3. *If L is (k,l)-contextual, then $N_L := F_{(k,l)}(L)$ is the smallest set G (with respect to set inclusion) such that $L = L_{(k,l)}(G)$. We call N_L the* representative set *for L.*

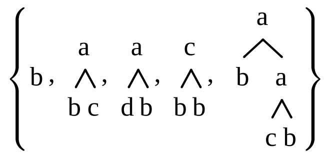
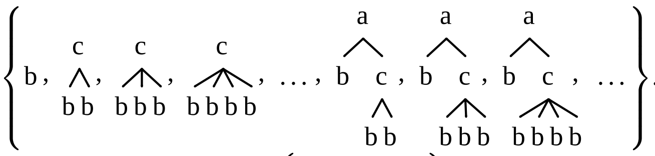
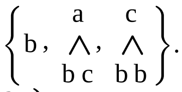
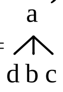
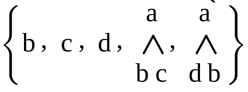
Proof First, we show that $L = L_{(k,l)}(N_L)$. The inclusion from left to right is trivial. For the converse inclusion, we know that $L = L_{(k,l)}(G)$ for some G (since L is given to be (k,l)-contextual). Clearly, $N_L \subseteq G$ for any such G. Hence, if $F_{(k,l)}(t) \subseteq N_L$ for some tree t, then also $F_{(k,l)}(t) \subseteq G$ and thus $t \in L$.

Since we observed that $N_L \subseteq G$ for any G such that $L = L_{(k,l)}(G)$, the minimality of N_L is established as well and the proposition is proved. □

As an immediate corollary we obtain:

Corollary 1. *For any two (k,l)-contextual languages L_1 and L_2, we have $L_1 \subseteq L_2$ if and only if $N_{L_1} \subseteq N_{L_2}$.*

Proof The implication from left to right is trivial. For the other direction, we have $L_1 = L_{(k,l)}(N_{L_1}) \subseteq L_{(k,l)}(N_{L_2}) = L_2$. □

Example 4. To illustrate these definitions we show an example with $(k,l) = (2,2)$. Given a set of trees $G =$  , the associated language is $L_{(2,2)}(G) =$ . And the representative set for this language is $N =$ . The tree $t =$  $\notin L_{(2,2)}(G)$ because $F_{(2,2)}(t) =$  $\not\subseteq G$. Note that $N \not\subseteq L_{(2,2)}(G)$.

Our definition generalizes to unranked trees the notion of k-testable string language "in the strict sense". To generalize the more expressive notion of k-testable, studied by McNaughton [20], one should consider a set of sets of forks for G (one for each example). Then a tree is accepted if its forks are a subset of those from one example. Our experiments (Section 5.3) indicate that k-testable languages in the strict sense are sufficiently expressive, hence we explore only this notion.

The local unranked tree automata of [17] correspond to the special case $l = 2$ in our approach. The lack of expressiveness in vertical direction was remedied with some extra preprocessing (see Section 5.2).

3.3 Learning (k,l)-Contextual Tree Languages from Positive Examples

To learn a (k,l)-contextual tree language from a set of positive examples E, we collect the (k,l)-forks of these examples and use them as a representative set for the language to be learned. In other words, we assume the language to be learned equals $L_{(k,l)}(F_{(k,l)}(E))$ (see Algorithm 1). Note that the representative set for this language equals $F_{(k,l)}(E)$. This way, overgeneralisation is avoided as, for a given k and l, the algorithm finds the most specific (k,l)-contextual language that accepts all the examples. Intuitively we can state that the generalization is constrained by restricting the minimal granularity of the building blocks that can be used. These building blocks are the forks found in the examples.
We note that this learning method is anti-monotonic in the parameters k and l:

Proposition 4. *If $k' \geq k$ and $l' \geq l$ then $L_{(k',l')}(F_{(k',l')}(E)) \subseteq L_{(k,l)}(F_{(k,l)}(E))$.*

Proof Let t be a tree such that $F_{(k',l')}(t) \subseteq F_{(k',l')}(E)$. We must show that $F_{(k,l)}(t) \subseteq F_{(k,l)}(E)$. Consider a (k,l)-fork r of t. Then r can be extended to a (k',l')-fork r' of t (it is possible that r' equals r). By the given, r' also appears as a (k',l')-fork of some tree t' in E. But then r appears as a (k,l)-fork of t' as well, and thus $r \in F_{(k,l)}(E)$ as had to be shown. $\square$

Algorithm 1 learnWrapper

```
Input: The set of positive examples $E$, and the parameters $k$ and $l$.
Output: The learned $(k,l)$-contextual language.
1: $Forks = \emptyset$
2: for each $Example \in E$ do
3:     $Forks = Forks \cup F_{(k,l)}(Example)$
4: end for
5: return $L_{(k,l)}(Forks)$
```

This anti-monotonicity is typical for local languages such as k-contextual languages [21, 1], k-testable languages [11], and k-testable tree languages [10, 15].

In the remainder of this section we formally prove that the class of (k,l)-contextual tree languages is learnable in the limit [12] from positive examples only. This proof is structured in the same way as similar proofs in [2, 21]. Hence we start by proving that there exists a characteristic sample for every language.

Definition 2. *A characteristic sample of a (k,l)-contextual tree language L is a finite subset S of L, such that L is the smallest (k,l)-contextual tree language, given k and l, that contains S.*

Proposition 5. *Every (k,l)-contextual tree language has a characteristic sample.*

Proof Call two trees t_1 and t_2 "equivalent" if $F_{(k,l)}(t_1) = F_{(k,l)}(t_2)$. Since there are only finitely many different trees of width k and height l, there are also only finitely many different sets of such trees, and as a consequence, there are only finitely many different equivalence classes. Moreover, any (k,l)-contextual language L is closed under equivalence, i.e., can be written as a (finite) union of equivalence classes. We now claim that it suffices to pick a representative from each class in L to obtain a characteristic sample S for L.

To prove this claim, consider any other (k,l)-contextual language L' such that $L' \supseteq S$. We show $L \subseteq L'$. Thereto, let $t \in L$. Then t is equivalent to some representative t' from S. Since S is contained in L' and L' must be closed under equivalence, also $t \in L'$ as desired. □

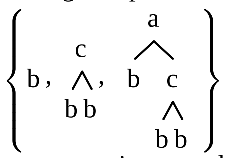
A positive presentation of a language L is an infinite sequence of trees $T = t_1, t_2, t_3, \ldots$, such that every element of the sequence is an element of L and vice versa. We define an inference operator KL, which given an infinite sequence of trees $t_1, t_2, t_3, \ldots$ and parameters k and l, produces an infinite sequence of tree languages $L_1, L_2, L_3, \ldots$ in which $L_n = L_{(k,l)}(F_{(k,l)}(\{t_1, t_2, \ldots, t_n\}))$ for all $n \geq 1$.

Observe, by Proposition 3, that $L_n \subseteq L$ for each n, if L is (k,l)-contextual. The following proposition now shows that (k,l)-contextual languages are indeed identifiable in the limit:

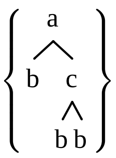
Proposition 6. *If L is (k,l)-contextual, then $L_n = L$ for n sufficiently large.*

Proof Let n be sufficiently large such that $\{t_1, \ldots, t_n\}$ includes a characteristic sample S of L. Then L_n is a (k,l)-contextual language containing S and thus $L_n \supseteq L$. Since also $L_n \subseteq L$, we conclude $L_n = L$. □

Example 5. The language $L_{2,2}(G)$ in Example 4 is divided in 3 equivalence classes. Taking a representative of each class results in the following characteristic sample:



. We can reduce this to



, as all the forks of the first two representatives are also forks of the last one.

Hence the class of (k, l)-contextual tree languages is learnable from positive examples only. Choosing larger parameters allows for more expressive languages, at the cost of bigger representative sets and bigger characteristic samples for these languages. For learning we want to choose the parameters small, to minimize the number of examples needed, but big enough such that the resulting language is expressive enough for the problem at hand.

4 Information Extraction with (k,l)-Contextual Tree Languages

In Section 4.1, we describe how the marking of nodes of interest provides a foundation for the use of a (k, l)-contextual tree language as a wrapper that can extract the nodes of interest. Section 4.2 introduces two generalizations to the basic (k, l)-contextual tree construction. The first one introduces wildcards to generalize over irrelevant text nodes and the second one ignores forks that are not in the neighborhood of the node of interest.

4.1 Representing Wrappers with Tree Languages

In this section we show how tree languages can be used to extract information from (semi)structured documents like HTML. Basically we need to be able to select specific nodes (be it leaves or internal nodes) from a given tree. This selection is represented by marking the tree. A marked tree is an element of $T(\Sigma_X)$ with Σ a given alphabet and X a marker. The marked alphabet Σ_X is defined as $\Sigma_X = \Sigma \cup \{s_X \mid s \in \Sigma\}$. Hence the nodes of a marked tree have either a marked or an unmarked label. Marking a node s consists of replacing it by a marked equivalent s_X.

A marking of a tree is correct (with regard to the extraction task) when each node that should be selected, and only those, are marked. Each tree has a unique correct marking. A marking is partially correct when every node that is marked is indeed a node that should be selected. A given tree can have multiple partially correct markings. Tree languages defined over Σ_X will accept marked trees. Given some extraction task, one can construct a tree language that accepts all correctly marked trees as well as a language that accept all partially correct marked trees.

Given a tree and a tree language that accepts correct markings for a given task, every possible marking of the tree can be checked and the single one that is contained in the language, indicates the requested nodes. The exponentially high number of possible markings makes this approach infeasible though. Using a tree language that accepts only partially correct markings allows for a more practical extraction method. One can mark each time a single node in the tree. If this marked tree is accepted, the marked node is one of the target nodes. Hence only *n* markings have to be checked, with *n* the number of nodes. This approach is also used in [16–18]. In [24] an even more efficient extraction method is proposed that extracts the nodes in a single run. We will not discuss the latter method here as it falls out of the scope of this paper. Given that a practical method for extraction exists, we can conclude that representing wrappers by tree languages is a valid approach.

4.2 Wrapper Induction

To learn a wrapper, we need to learn a language that accepts the partially correct marked trees for the given task. The training examples given to the learning algorithm consist of trees with a single node marked. Each marked node corresponds to an example of a target node of the task. We make two extra changes to the basic algorithm. The first change is to generalize over text nodes, the second one is to generalize over the set of forks.

As the text nodes come from an infinite alphabet we cannot learn them from a small number of examples. To solve this, we follow [16–18]: All text nodes in the examples are replaced by a wildcard (@). During extraction the wildcard matches with every text node, even those not seen during the learning phase. Sometimes this leads to overgeneralisation, when a text node close to the target is needed to disambiguate between a positive and a negative example. We call such a text node a distinguishing context. A set of distinguishing contexts can be given, to keep text nodes with these context from being replaced. Figure 3 shows a positive example, for the author extraction task of Example 1 (using a marker ’A’) as well as a positive example for the name extraction task of Example 2 (using a marker ’N’). One node is marked in both examples; a distinguishing context is used for the name extraction task.

The heuristic we use to determine the set of distinguishing contexts is different (and simpler) than the one used in [16–18]. We inspect all positive examples in a preprocessing step. For each positive example we collect the set of text nodes that occur in the (k, l)-forks for that example that contain the marked node. The set of distinguishing contexts is then the intersection of these sets. This way text nodes are only generalized when there is a positive example for which they do not occur in its parameterized neighborhood. This procedure guarantees that (given sufficient examples) all the strings remaining in the resulting set are true context for the target node. It is possible though that some discriminative context string is not found (for example, when the target is a node with as context either c1 or c2). So far we have not encountered the need for a more elaborate procedure. A final remark is that the use of distinguishing contexts can be turned off. The boolean which controls this feature can be considered as a third parameter of our algorithm; the others being k and l.

When learning marked (k, l)-contextual tree languages, the set of forks in the representative set splits naturally in two; a set of forks with a single node marked, and a set of forks without any marked node. One can argue that the forks containing the marker provide the local context needed to decide whether a node should be extracted or not, while the other forks describe the general structure of the document. The latter merely serve to decide whether the document is in the domain of the extraction task. Learning the domain typically requires substantially more examples than learning the

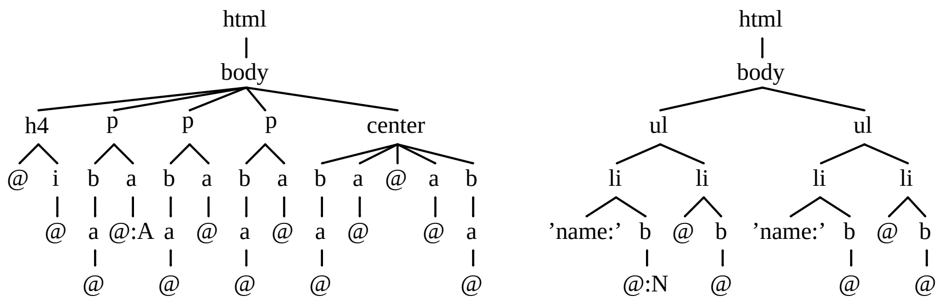


Fig. 3. Positive example for author extraction (left) in the Article Database (see Figure 1) and for student extraction (right) in the Student List (see Figure 2). In both examples, one node is marked (with respectively "A" and "N"). For the Article Database, all text nodes have been replaced by the wildcard "@"; for the Student List, $\{\ 'name\ :'\}$ is used as distinguished context and is not replaced by the wildcard symbol.

local context. However, in our setting, we assume all documents are from the correct domain; hence there is no need to learn the domain and we can ignore all forks that do not contain the marker during learning and extraction. This way a small set of examples will be able to cover the variance of forks in areas far away from the targets.

Example 6. Given the positive example for the author extraction task shown in Figure 3, and given the parameters $k = 1$ and $l = 3$, we learn the wrapper by collecting the marked forks. The representative set for the wrapper becomes $\left\{ \text{@:A},\ \begin{array}{c} \text{a} \\ | \\ \text{@:A} \end{array},\ \begin{array}{c} \text{p} \\ | \\ \text{a} \\ | \\ \text{@:A} \end{array} \right\}$.
Marking the node containing 'author2', and collecting its (1,3)-forks results in the set $\left\{ \text{'author2':A},\ \begin{array}{c} \text{a} \\ | \\ \text{'author2':A} \end{array},\ \begin{array}{c} \text{p} \\ | \\ \text{a} \\ | \\ \text{'author2':A} \end{array} \right\}$. Given that a wildcard matches every text node, each of these forks matches with one of the forks in the representative set, hence "author2" is extracted. For the text node containing '1', the collected set of (1,3)-forks is $\left\{ \text{1:A},\ \begin{array}{c} \text{a} \\ | \\ \text{1:A} \end{array},\ \begin{array}{c} \text{center} \\ | \\ \text{a} \\ | \\ \text{1:A} \end{array} \right\}$. The last fork does not match with the forks in the representative set and the marked text node is rejected.

For the student extraction task, the tree as shown in Figure 3 gives as representative set of the wrapper learned for $k = 2$ and $l = 3$ the following set of forks:

$\left\{ @{:}A,\ \begin{array}{c} b \\ \mid \\ @{:}A \end{array},\ \begin{array}{ccc} & li & \\ \diagup & & \diagdown \\ \text{'name:'} & & b \\ & & \mid \\ & & @{:}A \end{array} \right\}$. This wrapper will extract all students, and reject all other nodes.

5 Related Work and Experimental Comparisson

In this section we survey related work, starting with string based methods in Section 5.1 and continuing with tree based method in Section 5.2 and select the best existing methods for inclusion in an experimental study which is described in Section 5.3.

5.1 String Based Methods

Some string based methods [6] are node based, i.e., they extract whole text nodes, but the majority can extract a substring from a text node. Instead of finding a single node, they return the boundaries of the substring. Except for WIEN [19] which is character based, all systems are token-based. This means that the *start* and *end* boundaries are always between two tokens. The target values will usually not contain half a token, and the higher granularity speeds up the learning phase. The use of wildcards for different classes of tokens also drastically reduces the amount of examples needed for learning. Some systems were designed for information extraction from free text (BWI [8], HMM [9], SRV [7], RAPIER [4]) but are general enough to apply to semi-structured data as well. WHISK [30] is designed for both free text and semi-structured data. We have chosen two of the more performant systems (BWI and STALKER [22]) to compare experimentally with our approach. Experiments in [8] show that BWI outperforms HMM, SRV and RAPIER (the last one only tested on free text). It is argued in [22] that the rules used in WIEN, SoftMealy [14], SRV [7] and RAPIER [4] are strictly less expressive than STALKER's. In the same paper it is shown experimentally that STALKER outperforms WIEN. Below we describe the STALKER and BWI system a bit more in depth.

To extract a subsequence from a sequence of tokens, the STALKER system uses a start and an end rule, to find the boundaries of that subsequence. The start rules are executed in forward direction from the beginning of the sequence, the end rules are executed in backward direction. A STALKER rule is either a simple rule or a disjunction of simple rules. In the latter case the boundary is given by the first simple rule that does not fail. The simple rules are based on a list of so-called landmarks. A landmark is a sequence pattern consisting of tokens and/or wildcards. When a rule is executed, it searches for a part of the sequence that matches the first landmark. From the end of this part the search for the second landmark is started, and so on. The boundary that is finally returned is either the end or the beginning of the part that matched the last landmark. Which one is indicated by a modifier. This is respectively SkipTo and SkipUntil for using the end or the beginning (or BackTo and BackUntil for rules in the other direction). When the search for a landmark reaches the end/beginning of the sequence, the rule is said to fail. STALKER uses multiple types of wildcards that form a type hierarchy. This hierarchy is shown in Figure 4.

Example 7. The rule *SkipTo(*<`p`><`b`><`a`>*)* applied on the HTML sequence of Example 1 returns the position at the end of the first occurrence of these three consecutive tags, i.e., at the beginning of 'title1' While the rule *BackTo(*<`center`>*)* *BackTo(*<`/a`>*)* applied on the same sequence returns the position at the end of 'author3'. These rules will both fail on the HTML sequence of Example 2. The rule *SkipTo(*`name` *Punctuation) SkipUntil(Capitalized)* with two landmarks, each containing a wildcard, will, given the sequence of Example 2, return the beginning of 'Stefan'.

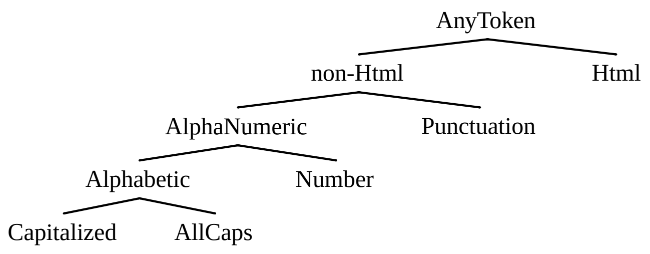


Fig. 4. Wildcard hierarchy. A token that matches a wildcard of a given type, will also match the wildcards of the ancestors of that type.

The STALKER induction algorithm starts from a set of positive examples (each consisting of a sequence wherein boundaries of a subsequence are given). As long as this set is not empty, a new simple rule is learned, those examples that are covered by this rule are removed from the set, and that rule is added to the disjunction of rules that will be the final result. The algorithm to learn a simple rule chooses one *seed* example (the shortest example in the set) to guide the induction, the other examples are used to test the quality of candidate rules. The algorithm does not search the entire rule space for the best rule but takes in each loop two rules from the current set of rules. One is the best solution in the set, the other is the best refiner in the set. Some heuristic rules are designed to define a ranking (best solution and best refiner) over a set of rules. This ranking is based on properties of the rules and on the number and quality of the extractions that each rule performs on the other examples. The refinements of the best refiner, together with the best solution, gives the new rule set for the next iteration. This loop continues until a perfect solution is found (one that either extracts correctly from an example or fails on that example) or until all refinements fail. The initial set of candidate rules are single landmark rules, with each landmark a single token or wildcard (occurring in the seed). The refinement step will either extend one of the landmarks of a rule with an extra token or wildcard (the extended landmark has to match within the seed), or add a new single token/wildcard landmark somewhere in the rule (the token or wildcard has to occur in the seed).

In contrast with other string based methods, STALKER implements a hierarchical extraction approach. An *Embedded Catalog (EC)* describes the structure of the data. This is a tree structure where the leaves are fields, and the internal nodes either tuples or lists. Figure 5 shows the EC for Example 1 and 2. Note that the EC formalism might not be expressive enough to represent some more complex data structures. To extract a specific field, first the parent has to be extracted, and the extraction rules are then applied on the subsequence extracted for the parent. To extract the author fields of Example 1, first the complete list of papers is extracted (the rules from Example 7 achieve this). Then the individual papers are extracted. And finally from each paper, the author field is extracted. The advantage of this approach is that complex extraction tasks are split into easier problems. Disadvantages are that during learning more examples are needed to learn for every level of the hierarchy[4], and that errors in the different levels will accumulate.

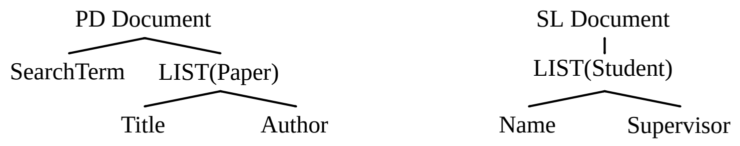


Fig. 5. Embedded Catalogs for the paper database ex. (left) and the student list ex. (right).

Like STALKER, the Boosted Wrapper Induction (BWI) extracts a subsequence using a start rule and an end rule. A BWI rule is a set of simple rules with an associated weight. During extraction, each simple rule in the set extracts a boundary and casts a weighted vote, to return a single winning boundary. Using the terminology from STALKER, a single BWI rule consists of two landmarks, called prefix and suffix. The rule searches for the first sequence that matches the concatenation of prefix and suffix. The boundary point is placed in between the tokens that match the prefix and those that match the suffix. This is less expressive than a simple STALKER rule. In BWI, both start rules and end rules go in the forward direction. The BWI system does not extract hierarchically. The rules are applied to find every matching point in the entire HTML sequence.

Example 8. The BWI rule $\langle$ [<`p`><`b`><`a`>], [*non-Html*] $\rangle$ looks for the first sequence of the three html tags that are followed by an non-html token. Applied on the HTML sequence of Example 1, it returns the start position of 'title1'.

The BWI learning algorithm uses a boosting approach. It learns repeatedly from a set of weighted examples. In each iteration, a simple rule is learned with a *weak learner*. After each iteration, the weights of the examples are changed according to the performance of the learned rules. Examples that are extracted well get a smaller weight while for the others the weight is increased. Hence, in each iteration, the weak learner focusses on the examples for which the results are poor. This technique is shown to give significant improvements over the use of the weak learner on its own [29]. The number of boosting iterations is given as a parameter T.

[4] To learn list extraction, each example should consist of two consecutive elements of the list.

The weak learner used in BWI, starts from an empty pair $\langle$ [], [] $\rangle$. The algorithm searches for the best possible extension (front) of the prefix and the best possible extension (back) for the suffix. A *lookahead* parameter L indicates the maximal length of the extensions. Every combination of the old and new prefix with the old and new suffix is given a score with a function that measures performance on the training set. The combination with the best score becomes the new rule. This process is repeated until the rule remains unchanged. The algorithm can work with extra negative examples next to the positive ones.

It is not straightforward to compare the expressiveness of the (k, l)-contextual tree languages with the wrapper representation languages of STALKER or BWI. Therefore, an experimental comparison is performed in Section 5.3. Note though that even when a correct wrapper can be expressed in the representation language, the heuristic search in STALKER does not guarantee to find it. For our induction algorithm holds that if a correct wrapper can be expressed as a (k, l)-contextual tree language, it will be found (given sufficient examples).

5.2 Tree Based Methods

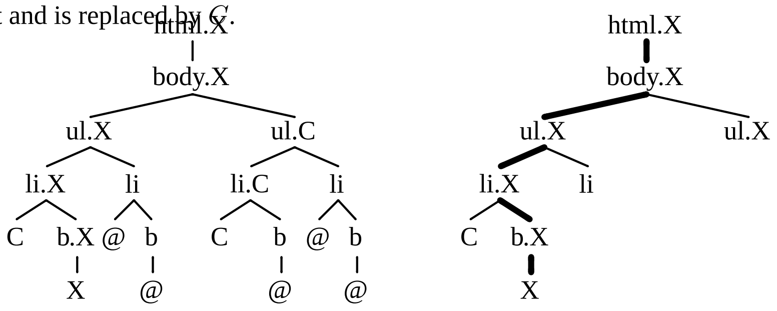
In [17], it is shown that wrappers learned directly from the unranked tree structure of the document perform better than wrappers that learn from ranked representations of the unranked document tree. The tRPNI algorithm [5] learns from unranked trees. Its hypothesis space consists of whole class of regular unranked tree languages. However, it needs completely annotated documents. As our focus is on learning from a few positive examples, we do not consider it further. The other tree based learning approach we are aware of is the the Local Unranked Tree Inference(LUTI) algorithm [17] which we describe in more detail below. Besides approaches that learn a wrapper, there is also a research line that explores wrapper programming languages and the visual specification of wrappers. See [13] for a representative example.

The Local Unranked Tree Inference algorithm is closely related to our approach. As mentioned in Section 4.2, both methods start with a preprocessing step to generalize over the text nodes in the training examples. Each text node becomes either a target (X), a distinguished context (C) or a generalized text node (@). Note that this approach uses only a single string as distinguishing context instead of a set. Basically this method infers a $(k, 2)$-contextual language (a special case $l = 2$ of our method). But some extra differences exist. We can describe these as two transformations that are performed in a preprocessing step on both training examples as on the documents to be extracted.

The first transformation replaces every node f into a node $f.X$, if its subtree contains the X-node. If the subtree does not contain the X-node but a C-node then it is replaced by $f.C$. Hence, limited information is passed infinitely upwards, making the method not purely local. However, the subclass remains inferable and the expressiveness is enhanced.

The second transformation in [17], although part of the inference algorithm, can also be explained as a preprocessing step. The automaton accepts everything below a node that is not of the form $f.X$, i.e., all subtrees below such nodes can be removed and only the path from the root to the X-node is left, together with the siblings of the nodes on that path; parts farther away from the marked node are ignored. This enhances the generalizing power of the resulting language (and reduces the expressiveness).

Example 9. The left tree below shows the document tree of Example 2 after the first transformation, while the tree on the right shows the result of applying the second transformation to that same tree. Note that the string "name:" is used as the distinguishing context and is replaced by C.



Thanks to the first transformation, the LUTI algorithm can express some global vertical relations. The relation between the target node or a context node and an ancestor that is an arbitrary number of levels higher can be described, despite that l is always 2. Our algorithm (KL) is purely local and does not have this extra expressiveness. Our experiments showed that local information in the vertical direction (a high enough l parameter) was sufficient for all data sets.

The second transformation in LUTI reduces its expressiveness with regard to KL as all information about the siblings of the target node is removed while KL retains this neighborhood. We encountered several data sets where that information was needed to disambiguate positive and negative examples. Consider, for instance, a table with bargains. The aim is to extract those with a picture of the item. The picture, when present, occupies the first cell of the row (a sibling of the cell containing the target).

5.3 Experiments

We evaluate our approach on the WIEN data sets, the bigbook data set and the okra data set[5]. Each of these sets contains a set of pages from a same domain, most of them have an associated set of annotations for a specific extraction task. We have split the tasks aiming at the extraction of a n-tuple in n single field extraction tasks. We refer to them with the name of the original data set and the index of the field in the tuple. We use a well-defined subset of 36 extraction tasks from the available WIEN data tasks, namely those that extract a complete text node and for which the information on the nodes to be extracted is available in the WIEN data. Each example is a single page with exactly one of the target values marked. The target concepts are given only through annotated pages. No rules are given for a correct wrapper. Hence we will assume that the annotated pages contain all possible exceptions, and that a wrapper is correct for the given domain when it correctly extracts every annotated page in the data set.

[5] These are available at the RISE repository: http://www.isi.edu/info-agents/RISE/index.html.

We use the F1 score as a fitness criterion. Given E, the number of text nodes extracted from the test set, C, the number of correctly extracted text nodes, and T, the total number of text nodes to be extracted from the test set, Precision(P) is defined as P=C/E, recall(R) as R=C/T. The F1 score is defined as the harmonic mean of precision and recall: F1=2PR/(P+R).

For the purpose of doing these experiments we have obtained an implementation of the BWI algorithm from the Fondazione Bruno Kessler. For the STALKER and Local Unranked Tree Inference algorithms we used our own implementation. Through extensive communication with the authors we have tried to stay as close as possible to the original implementations. We used for both STALKER and BWI the same set of wildcards, the one shown in Figure 4.

The experiments compare the ability of the different algorithms to learn from a small set of positive examples. In our setup, each experiment selects 5 random examples from a data set. Each algorithm learns from the same 5 examples, and the F1 score of the resulting wrapper on the whole data set is calculated. This experiment is not intended to measure the number of examples needed by each algorithm but to measure which one learns best from a given sample of (incomplete) data. Table 1 shows for each task the mean over 5 experiments. Note that due to the hierarchical nature of STALKER, it is given more information per example. Not only the boundaries of the target, but also the boundaries of its ancestors, and when one of the ancestors is a list element also the boundaries of one of its adjacent siblings. It also expects the embedded catalog for the induction task at hand. For the WIEN data sets, we use the embedded catalogs originally used in the STALKER papers. We did not run STALKER on bigbook and okra due to a lack of embedded catalogs for them. We did not run BWI on them as it was not worth the effort of making an extra converter.

The STALKER algorithm is parameterless. The (k, l)-contextual algorithm presented here needs three parameters: k, l, and whether to use distinguishing contexts or not. These parameters need tuning for every task. In column *KL(opt.)* in Table 1, the results are shown for the set of optimal parameters for each experiment (based on F1 score on the test set). To have a fair comparison though, the parameters have to be chosen without the extra annotations in the test set. We therefore use a parameterless version of our algorithm [25]. This version uses a heuristic to make an estimation for the parameters, based on unmarked pages. As unmarked pages we use the pages from the given examples with the markers removed. The results are given in column *KL(est.)*. The estimated parameters are sometimes suboptimal though. This is part of the motivation to learn these parameters interactively (see Section 7).

The BWI system has two parameters for the learning phase: lookahead(L) and the number of boosting iterations (T), and one parameter (τ) for the extraction phase that allows to make a tradeoff between precision($\tau = 1$) and recall($\tau = 0$). We used for T the values 10 and 100, but found them to make no difference. The explanation in [8] seems valid: in contrast to free text, semi structured documents are formatted highly regular, and therefore need only a few boundary detectors. For τ we used the values 0, 0.5, and 1, but found them to have very small influence. The training time increases exponentially with the lookahead. L is therefore a tradeoff between quality and time. We used all values from 2 to 7. The LUTI algorithm has also two parameters: k and whether

Table 1. Experimental comparison on how well the given algorithms perform with few examples. Each column shows the F1 score for the wrappers learned with the respective algorithms on a set of only 5 random examples (each experiment is performed 5 times and the results are averaged). The first column indicates the data set, while the second column indicates whether the algorithms LUTI, KL(opt.), and KL(est.) used distinguishing contexts.

Data set	ctx	LUTI	KL (opt.)	KL (est.)	STALKER	BWI
s1-1		89.1	**100.0**	**100.0**	92.2	**100.0**
s1-3		90.4	**98.7**	93.4	81.0	9.1
s1-4		78.8	**100.0**	**100.0**	93.1	42.7
s3-2		97.6	**100.0**	**100.0**	99.4	27.4
s3-3		98.2	**100.0**	**100.0**	96.3	6.4
s4-1		91.6	**100.0**	**100.0**	88.8	58.9
s5-2		93.8	**98.9**	94.7	91.3	27.6
s8-2		**100.0**	**100.0**	**100.0**	95.9	31.6
s8-3		**100.0**	**100.0**	**100.0**	91.3	96.6
s10-2		**100.0**	**100.0**	**100.0**	96.8	20.3
s10-4		**100.0**	**100.0**	**100.0**	96.3	10.8
s11-1	✔	**100.0**	**100.0**	**100.0**	91.7	1.9
s11-2	✔	**100.0**	**100.0**	89.4		8.1
s12-2		98.4	**98.5**	98.4	93.1	33.8
s13-2		**100.0**	**100.0**	**100.0**	95.9	38.7
s13-4		**100.0**	**100.0**	**100.0**	85.6	5.5
s14-3		99.5	**100.0**	**100.0**	78.0	14.1
s15-2		97.1	**100.0**	**100.0**	96.2	34.9
s19-4		**100.0**	**100.0**	**100.0**	**100.0**	17.2
s20-3	✔	98.5	**100.0**	**100.0**	**100.0**	97.7
s20-4	✔	97.5	**100.0**	**100.0**	99.6	**100.0**
s20-5	✔	97.5	**100.0**	**100.0**	**100.0**	86.4
s20-6	✔	98.5	**100.0**	**100.0**	84.0	**100.0**
s22-2		93.3	**100.0**	99.8	**100.0**	68.9
s23-1		97.6	**100.0**	**100.0**	87.5	99.5
s23-3		94.4	**100.0**	**100.0**	96.2	19.7
s25-2		97.2	**100.0**	**100.0**	93.5	20.9
s29-1		**96.6**	**96.6**	65.6	87.3	22.8
s29-2		**100.0**	87.8	36.8	60.7	28.4
s30-2		96.0	**100.0**	96.0	88.0	88.6
bigbook-2		94.3	**100.0**	97.3		
bigbook-3		88.0	**100.0**	96.9		
okra-1	✔	**100.0**	**100.0**	**100.0**		
okra-2	✔	99.3	**100.0**	**100.0**		
okra-3	✔	99.1	**100.0**	**100.0**		
okra-4	✔	99.1	**100.0**	**100.0**		

to use distinguishing contexts or not. The results shown for both LUTI and BWI, are those with the set of optimal parameters for each experiment (based on F1 score on the test set). The context parameter for LUTI and KL (both opt. and est.) was always the same, and is given in an extra column *(ctx)*. Note that no results for STALKER are included for data set s11-2, as this task contains alternative values, i.e., multiple elements to be extracted for the same field within the sequence extracted for the parent tuple. This cannot be represented in the embedded catalog formalism of STALKER.

Even though the parameterless version of the KL algorithm does not always reach optimal scores, it is only beaten in 4 of the 36 tasks. Three times by STALKER, and three times by LUTI, while it beats LUTI 22 times and STALKER 23 times (on 29 tasks). Note that using optimal parameters for LUTI and BWI is not fair when comparing them with STALKER, but makes the results for KL(est.) stronger. Recall that STALKER also has the advantage of getting extra information.

6 Learning the Parameters

As shown in Section 5.3, our (k, l)-contextual tree language improves upon the local unranked tree automata of [17] by being able to learn from fewer examples. However, a problem shared with [17] is that the method needs **parameter tuning** for each task. Selecting the optimal parameters requires to run the program on a set of completely annotated documents to obtain precision and recall. Hence parameter selection is in fact based on a large set of positive and negative examples.

Here, we describe how to learn parameters based on a small set of negative examples. In addition, it is indicated when (k, l)-contextual tree languages are not expressive enough to reach a 100% F1-score for the extraction task at hand.

Our algorithm finds the parameters k and l, such that the (k, l)-contextual language learned from the positive examples is the most general one that still rejects all the negative examples.

6.1 Algorithm

Order relations We will use two order relations on languages. The first is the standard set inclusion $L_1 \subseteq L_2$. Recall from Proposition 4 that this order is anti-monotonic in the parameters. The second order is defined using a finite set S of trees. Let $\#acc(S, L)$ be the number of trees from S that belong to the language L (the *count*). Then we define $L_1 \geq^{\#}_{S} L_2$ as $\#acc(S, L_1) \geq \#acc(S, L_2)$. Note that for any S we have $L_1 \supseteq L_2 \Rightarrow L_1 \geq^{\#}_{S} L_2$, hence $\geq^{\#}_{S}$ is also anti-monotonic in the parameters, i.e., the count decreases with increasing parameter values.

Solutions In what follows, we denote with $[k, l]$ the (k, l)-contextual language learned from the given positive examples Pos. So, $[k, l]$ equals $L_{(k,l)}(F_{(k,l)}(Pos))$. Any such set $[k, l]$, for some parameters k and l, is called a *potential solution*. If, moreover, $[k, l]$ is consistent with the negative examples Neg, i.e., if $Neg \cap [k, l] = \emptyset$, we call $[k, l]$ a *solution*. We define a solution L_1 to be better than L_2 when it extracts more solutions from the documents used to learn the wrapper; more formally, when $\#acc(S, L_1) \geq$

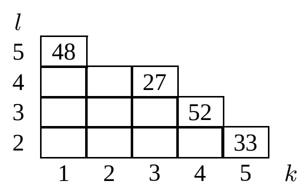
$\#acc(S, L_2)$ where S has a tree for each candidate node (with the candidate marked cnfr. Section 4.2). Hence the best solution is the solution that is maximal in the order $\geq^{\#}_{S}$.

Heuristic Due to the anti-monotonicity, we have that $\#acc(S, [k, l]) \leq \#acc(S, [k - 1, l])$ and $\#acc(S, [k, l]) \leq \#acc(S, [k, l-1])$, hence $\#acc(S, [k-1, l])$ and $\#acc(S, [k, l-1])$ are upper bounds on the value of $\#acc(S, [k, l])$. The algorithm uses them to estimate the value of $\#acc(S, [k, l])$ and, at each step, computes the count of the language with the best estimate. The search stops when the best estimate cannot improve upon the current best solution.

Example 10. Let *Pos* be the singleton with as element a tree *t* derived from the tree of the HTML example in Example 1, such that the node containing 'title1' is marked. Some examples of potential solutions, given *Pos* are $[1, 2] = L_{(1,2)}(\left\{ @{:}\mathrm{T}, \begin{array}{c} \mathrm{a} \\ | \\ @{:}\mathrm{T} \end{array} \right\})$, $[1, 3] = L_{(1,3)}(\left\{ @{:}\mathrm{T}, \begin{array}{c} \mathrm{a} \\ | \\ @{:}\mathrm{T} \end{array}, \begin{array}{c} \mathrm{b} \\ | \\ \mathrm{a} \\ | \\ @{:}\mathrm{T} \end{array} \right\})$, $[1, 4] = L_{(1,4)}(\left\{ @{:}\mathrm{T}, \begin{array}{c} \mathrm{a} \\ | \\ @{:}\mathrm{T} \end{array}, \begin{array}{c} \mathrm{b} \\ | \\ \mathrm{a} \\ | \\ @{:}\mathrm{T} \end{array}, \begin{array}{c} \mathrm{p} \\ | \\ \mathrm{b} \\ | \\ \mathrm{a} \\ | \\ @{:}\mathrm{T} \end{array} \right\})$, and $[2, 4] = L_{(2,4)}(\left\{ @{:}\mathrm{T}, \begin{array}{c} \mathrm{a} \\ | \\ @{:}\mathrm{T} \end{array}, \begin{array}{c} \mathrm{b} \\ | \\ \mathrm{a} \\ | \\ @{:}\mathrm{T} \end{array}, \begin{array}{ccc} & \mathrm{p} & \\ / & & \backslash \\ \mathrm{b} & & \mathrm{a} \\ | & & | \\ \mathrm{a} & & @ \\ | & & \\ @{:}\mathrm{T} & & \end{array} \right\})$. Defining S as the set of trees derived from *t* that have a single node marked, $\#acc(S, L)$ returns the number of nodes extracted from *t* by a wrapper based on L. For $[1, 2]$ the extractions from *t* are {title1, author1, title2, author2, title3, author3, Prev, 1, 3, Next}. Hence $\#acc(S, [1, 2]) = 10$. The extractions for $[1, 3]$ are {title1, title2, title3, Prev, Next}, and $\#acc(S, [1, 3]) = 5$. The extractions for $[1, 4]$ and $[2, 4]$ are {title1, title2, title3}, hence $\#acc(S, [1, 4]) = \#acc(S, [2, 4]) = 3$. For the set {author1, 1} as negative examples, we get that $[1, 3]$, $[1, 4]$, and $[2, 4]$ are solutions. From these, $[1, 3]$ is the best solution as it is the most general (k, l)-contextual tree language accepting *Pos* and rejecting all negative examples. With {Prev} as the the set of negative examples, only $[1, 4]$, and $[2, 4]$ are solutions. According to the heuristic, the count of $[2, 4]$ will be equal or smaller (equal in this example), hence there is no need for the algorithm to check $[2, 4]$ after $[1, 4]$ is found.

Initialization All $(k, 1)$-contextual languages extract all single node forks from the examples, hence are overly general and of no interest. Therefore, the search starts from the $(1, 2)$-contextual language as it has the largest count.

Algorithm To reduce the space requirements, our algorithm maintains for a given l-value the count of at most one (k, l)-contextual language. If $[k, l]$ is a solution, then the $(k + 1, l)$-contextual language is of no interest as it has a lower count; if it is not a solution, then its count is discarded as soon as the count of the $(k + 1, l)$-contextual language is computed. These counts are maintained in a *front* (of the search). For each l-value, the front maintains the k-value ($F.k[l]$), the count ($F.c[l]$) and whether it is a solution ($F.sol[l]$) (see the right of Figure6). In each step, the algorithm selects the minimal value l such that the language $[F.k[l], l]$ is most promising for exploration (the function BestRefinement): $[F.k[l], l]$ is not a solution and the estimation of its refinement has the highest bounds on its count. For $k > 1$, the refinement is the language $[F.k[l] + 1, l]$, however for $k = 1$, also $[1, l + 1]$ is a refinement.



l	k	c	sol
5	1	48	No
4	3	27	Yes
3	4	52	No
2	5	33	No

Fig. 6. Parameter Space and Data Representation

Example 11. Given the data in Figure 6, the languages $[1, 5]$, $[4, 3]$ and $[5, 2]$ are candidates for refinement. Although $[4, 3]$ has the highest count, its refinement $[5, 3]$ has a count bounded by 33 while both refinements of $[1, 5]$ have a count bounded by 48, hence the latter is selected for refinement.

A final point to remark is that it is useless to consider a language $[k, l]$ with k larger than $MaxK(Pos, Neg, l)$, the maximum branching factor for the forks of a given depth l (it depends on l because only the forks containing the target are considered). Indeed, an increase of k will not affect the number of extractions. The algorithm below achieves this by setting the k-value at level l to ∞ and the count to 0 when refining it. When this happens for all l values, then it means that no wrapper based on (k, l)-contextual tree languages is expressive enough to reach a 100% F1-score. Note that there is always a solution when all examples come from a single document. The final set of forks then becomes ultimately the set of marked versions of the whole document.

The algorithm is sketched in Algorithm 2. F is the array representing the front as shown in Fig. 6. For a given l value, the values $F.k[l]$, $F.c[l]$, and $F.sol[l]$ give respectively the k-value, the count and whether $[k, l]$ is a solution. It is initialized for $l = 2$ with k-value 1. The function $BestRefinement(F)$ returns the l-value of the best candidate for refinement (as described above) if it exists, otherwise it either returns the l-value of the solution or reports failure. The function $calc(Pos, Neg, k, l)$ updates $F[l]$ with the appropriate values. Note that two refinements are computed when the selected best candidate has a k-value of 1. As long as there are candidates for refinement (non-solutions) that have a larger bound than any of the solutions already encountered, the BestRefinement will return a non-solution. Hence the algorithm keeps searching for better (larger in the order $\geq^{\#}_{S}$) solutions even though some solutions are already found.

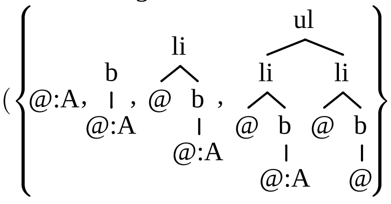
Algorithm 2 Learning the Parameters

```
Input: $Pos$ and $Neg$, The sets of positive and negative examples.
Output: The parameters $k$ and $l$ of the wrapper.
 1: calc($Pos$,$Neg$, 1, 2) // initialization
 2: $bestL$ = 2
 3: while not $F.sol[bestL]$ do
 4:     if $F.k[bestL]$=1 then
 5:         calc($Pos$,$Neg$, 1, $best$L+1)
 6:     end if
 7:     calc($Pos$,$Neg$, $F.k[bestL]$+1, $bestL$)
 8:     bestL = BestRefinement($F$)
 9: end while
10: return $F.k[bestL]$ and $bestL$
Function: calc($Pos$,$Neg$, $k$, $l$)
 1: if $k >$ maxK($Pos$,$Neg$, $l$) then
 2:     $F.k[l]$=$\infty$
 3:     $F.c[l]$=0
 4: else
 5:     $F.k[l]$=$k$
 6:     $W$ = learnWrapper($Pos$, $k$, $l$)
 7:     $F.sol[l]$=$W$ rejects all $N$
 8:     $F.c$ = cnt(extractions($W$,$Pos$,$Neg$))
 9: end if
```

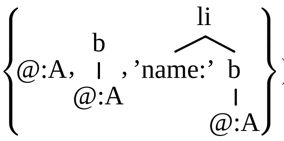
6.2 Learning with Context

The preprocessing step to collect a set of distinguishing contexts, as described in Section 4.2, ensures that the context increases with an increase in k or l. As the count of a wrapper decreases with increasing context, the anti-monotonicity property is still valid and our algorithm can easily be extended to learn a wrapper with context.

Example 12. Given the document of Example 2, with a positive example that has 'Stefan' marked, and a negative example that has 'Hendrik' marked. Using no distinguishing contexts the algorithm reaches a solution for $(k, l) = (2, 4)$, namely the language $L_{(2,4)}($



), while using distinguishing contexts a solution is reached for $k = 2$ and $l = 3$: $L_{(2,3)}($



).

Not all data sets need a context. In principle, one could learn the wrapper with context and the wrapper without context independently of each other. However, our system integrates both in one algorithm that maintains two fronts and selects the most promising point of both for refinement. Note that, for a given point (k, l), the count of the wrapper with context is bounded by the count of the wrapper without context; i.e., the latter value is used as an extra bound on the count of the former (hence the selection is such that the former will only be evaluated when that bound is already known).

7 Induction with Equivalence Queries

Arbitrary sets of positive and negative examples often contain redundant information. It is more efficient to use queries. The system will query the user for the information that it needs to improve its hypothesis. In this section we present a system based on the algorithms from previous section, that uses equivalence queries[3]. For a given set of examples, the system returns a hypothesis. The user is asked to indicate whether the hypothesis is correct or otherwise to give a counterexample (a false positive or a false negative). Most practical for the user is to check the already given pages or try out some new pages of his own choice. When he detects an error, he can signal that error to the system. The system keeps on updating its hypothesis until the user is satisfied.

In Section 7.1 we indicate how to adapt the algorithm of previous section for an efficient interactive use. In Section 7.2 we discuss some details of the implementation of our system and finally, in Section 7.3, we give an evaluation of its usability.

7.1 Interactive Algorithm

After each interaction the system updates its hypothesis. This is done by finding the $\geq_S^{\#}$-most general language that is consistent with the current set of examples. For this update step we can use the algorithm from Section 6. However, an incremental algorithm is feasible. This would certainly improve the timings in Table 2 (see Section 7.3).

Adding a positive example (a false negative) to the set of examples increases the set of forks, hence the counts of all wrappers. However, a (k, l)-wrapper that covers negative examples still does so and cannot become a solution. It means that the search of a solution can start from the current front. The initialization of the new search for parameters consists of updating the count fields ($F.c$) in the front.

Adding a negative example (a false positive) does not affect the set of forks. However the solution is invalid as it covers the new negative example. After updating the (true) solution fields ($F.sol$)[6], the search can resume from the current front.

In short, the algorithm from Section 6 can be used. When a new example is received, the values in the front are updated and the search resumes.

Example 13. Assume we want to learn the 'title' task from Example 1. The user gives an initial example. Let us assume he picks 'title1' (same as in Example 10, such that we can refer to the languages learned there). The system learns its first hypothesis and ends up with language $[1, 2]$. We see that in our example page, all title fields are marked, hence there are no false negatives. We could check on some other pages, however, the current page has several false positives: {author1, author2, author3, Prev, 1, 3, Next}. The user chooses one of them. The system (see the description of the implementation

[6] When the example is from a new document, also the counts are updated.

below) disallows the user to mark true negatives like ’search term’ and ’2’ as negative examples. The algorithm updates its hypothesis. Example 10 shows that not every choice is equally informative. Choosing ’Prev’ or ‘Next’ leads to the solution $[1, 4]$, while all other false positives lead to $[1, 3]$, necessitating an extra iteration. After reaching $[1, 4]$, the user cannot find other counterexamples in the given page. He may try other pages until he is convinced that the wrapper is indeed correct.

7.2 Implementation

Representing the wrappers as sets of forks is straightforward, and works fine most of the time. For some tasks (requiring large k and l-values, and with pages with a large branching factor), the time for learning and extraction becomes noticeable and becomes an annoyance in an interactive application.

We developed an implementation that represents the wrappers by unranked tree automata, enabling the use of extraction in a single run described in [24]. This substantially reduces the memory consumption and the execution time without affecting the language accepted by the wrapper. An algorithm capable of learning tree automata directly from the example trees instead of first extracting the set of all (k, l)-forks, is given in [26].

We added a graphical user interface to our application, which is basically a HTML-compliant browser, that allows the user to right-click on an element of the page to add an extra example. The system colors the background of all elements that are extracted by its hypothesis. A click on a colored element is interpreted as a false positive, a click on a plain element is interpreted as a false negative. This way the user is restricted to give only counterexamples to the equivalence query posed by the system.

7.3 Evaluation

In this section we evaluate the cost of this interactive system, and we compare with Aggressive Co-Testing (see [23]), an active learning approach on top of the STALKER algorithm.

Typically, a wrapper is found starting from a single page. Extra page views (random or those the user suspects to contain an exception) can help to learn possible exceptions. The ease of indicating a counterexample invites the user to choose the first error he spots. Checking whether on a page all targets and no other fields are marked can amount to a substantial amount of checks. However, with the graphical representation, and given that the layout of the target fields is mostly regular, an uncolored target field, or a colored element that is not a target field, really sticks out and the human pattern recognition ability is able to spot an anomaly in a glance (or after a quick scroll for larger pages). In our experimental evaluation we therefore use the number of counterexamples given by the user as a measure for the interaction.

The setup of our experiments is as follows. Initially a single random example is given to the algorithm. On every iteration, the algorithm learns a new hypothesis. The user input is simulated by taking a random element from the set of false positives and false negatives. The algorithm stops, when a 100% F1-score is obtained (no more false positives or false negatives). We use the same tasks as in the experiment of Section 5.3.

Each task is performed 30 times with random examples, the results are averaged. In Table 2 we show the number of interactions that are needed to learn the wrapper. The first column of the table contains the data sets. For the interactive (k, l)-contextual algorithm we include a column to indicate the numbers of positive and negative examples[7] needed (averaged), columns to show the learned k and l (these were the same for all 30 runs), and a last column to indicate the total time needed by all the learning steps in Algorithm 2, also averaged over the 30 runs.

The nodes in the local context of a field typically have modest branching factors. When the parameter l becomes larger, the forks will escape the local context around the target, and might include nodes with higher branching factors, leading to more complex tree automata. Even though a local solution is found, the algorithm keeps trying higher l-values, to make sure that no better solution exists. This explains some of the larger timings. For data set s29-2 the timings are exceptionally large (often larger than 30 minutes). We therefore used the following alteration to the basic algorithm: If a solution is already found, and the algorithm takes longer than 5 seconds, interrupt the algorithm and return the best solution so far. Hence for set s29-2 we show only the final parameters and the number of interactions, but no timings.

Before describing the experimental setup with Co-Testing, we briefly introduce this approach. In Co-Testing multiple views on the data are defined. A hypothesis is learned in each of the views. For a set of unseen, and unmarked data, contention points are defined as the examples on which hypotheses disagree. A query about a contention point will improve at least one of the hypotheses. Applied on STALKER, 2 views are used. Each boundary can be described by a forward rule or by a backward rule. The Naive Co-Testing approach picks a random contention point, while Aggressive Co-Testing tries to pick a contention point that is likely to be wrong for both views, such that both hypotheses can be improved. To order the contention points, patterns are learned on the content of the example fields. Content points that differ most from these patterns in both views, are chosen first.

The setup in [23] is to learn a wrapper for the extraction of the target field from its parent in the Embedded Catalog, and not to learn every extraction task in the hierarchy. We also refrain from using this setup, as the application of Co-Testing on list-extraction is not detailed in the paper, and the induction of the extraction tasks in the top of the hierarchy becomes impractically slow. Each example is therefore a small subsequence of the whole document containing exactly one target element. An example for the title extraction task from Example 1 could be the following sequence matching the paper field, with 'title1' marked.

```
title1</a></b> <a>author1
```

Hence the algorithm learns from, and the learned wrapper will extract from, the sequences extracted for the parent (paper) of the title field (see Figure 5).

Each induction task start with two random examples. As long as contention points exist, the Co-Testing approach asks the correct solution for one of the parent sequences (one containing the chosen contention point). Each induction task is performed 30 times, and the results are averaged. These results are also shown in Table 2. Column

[7] P/N = 1/0 means that the initial (1,2)-wrapper derived from only one positive example is a solution.

’P’ shows the average over all runs of the number of positive examples (the two random initial examples, plus the queries by the algorithm). Note that for data set s13-4, we stopped the algorithm after 100 queries, hence no 100% F1 score was reached. For data set s29-2, the algorithm did stop, but also no 100% F1 score was reached. This implies that in this data set both views made the same mistake, such that no extra contention points could be found. The last column holds the average time in milliseconds for the total of all induction steps in a single run. Again, these induction times are on subsequences of the document, while for the interactive KL, they are on the whole document.

Although the type of interactions are different (equivalence query opposed to a single query on a subsequence), we believe that the work for the user is almost the same. Hence we feel justified to point out that our proposed algorithm needs significantly less user interactions than Co-Testing. Also the timings show that the system is highly responsive and suited for interactive use.

Table 2. This table shows the number of interactions needed to learn the wrappers with either the interactive version of the (k, l)-contextual learning algorithm, and STALKER with Aggressive Co-Testing. The P/N column indicates the number of positive and negative examples needed to reach 100% F1 score, always starting from a single positive example. The results were averaged over 30 random runs. The resulting k and l parameters were the same in each run, and given in the columns k and l. The P column shows the number of positive examples requested by the Co-Testing algorithm, with a minimum of two for the two random initial examples. The results were also averaged over 30 random runs.

Data	Interactive KL				Co-Testing	
set	P/N	k	l	ms	P	ms
s1-1	1/1	1	3	21	2.5	35
s1-3	4/1.7	3	3	642	7.7	1595
s1-4	1/0	1	2	5	8.7	7376
s3-2	1/1	1	3	12	2.5	190
s3-3	1/0	1	2	4	2.5	77
s4-1	1/0	1	2	2	30.8	266528
s5-2	1/1	1	4	19	6.9	405
s8-2	1/1	1	3	12	2.3	9
s8-3	1/1.3	2	3	43	3.4	554
s10-2	1/1	1	3	9	3.0	34
s10-4	1/1	2	2	15	8.8	35072
s11-1	1/2.1	2	4	4191	80.3	4220
s11-2	1/1.6	2	4	732		
s12-2	2/1.4	1	4	43	8.8	515
s13-2	1/1	1	3	10	2.6	23
s13-4	1/1	2	2	15	100+	1387720
s14-3	1/0	1	2	3	5.6	441
s15-2	1/0	1	2	2	18.3	3912
s19-4	1/1	1	3	7	2.1	8
s20-3	1/0	1	2	3	2.8	29
s20-4	1/1.3	2	3	198	3.1	51
s20-5	1/2	2	3	1142	3.0	53
s20-6	1/1.3	2	3	44	3.1	3024
s22-2	1/1	1	4	26	3.77	88
s23-1	1/1	2	3	39	3.6	132
s23-3	1/1	1	3	12	8.8	382
s25-2	1/1	1	3	6	7.6	1246
s29-1	2/1.6	2	3	125	10.4	179879
s29-2	4.8/2.9	4	3		!15.9	114784
s30-2	2/1	1	3	12	2.5	5
bigbook-2	2/2	2	5	574		
bigbook-3	1/1.3	2	4	100		
okra-1	1/1.6	2	3	37		
okra-2	1/1	2	3	118		
okra-3	1/1.4	2	3	66		
okra-4	1/1	2	3	103		

8 Further Work

The system presented in Section 7, is rather a proof of concept and a research tool than an industrial strength wrapper induction system. The system lacks support for sub-node extraction, and the ability to extract the different fields of a tuple as a single entity. As mentioned before, our approach is intended to be modular. We believe it is easier and possible to solve these different problems separately.

We distinguish two cases in sub-node extraction. Either we have a text node, and we need to extract a substring of that text node, or we have an internal node, containing a subtree, and we need to extract a sequence that starts inside some text node in the subtree, contains some other nodes in the subtree, and ends in another text node in that subtree. A hybrid approach will learn to find the node that contains the field, and use a string approach to find the correct subsequence. In the second case, it might be interesting to learn to extract the first, and the last text node in two separate tasks, and then use an string approach to learn the starting boundary within the first text node, and the ending boundary in the second one. Some small scale experiments with manually crafted experiments to simulate a hybrid approach indicated a better performance on sub-node tasks by a hybrid approach using STALKER in comparison with STALKER alone.

In contrast to a hybrid approach we could also extend the tree formalism in the (k,l)-contextual approach. Every text node can be replaced by the root of a subtree that contains the tokens in that text node as children. The extraction task will become a dual extraction task. Extracting the initial token (which has become a node in the tree), and extracting the last token (also a node). Presumably extending the single wildcard towards a hierarchical wildcard type system as in string based methods will be beneficial.

The integrated approach in STALKER to extract fields and tuples, has some drawbacks, as the general structure of the document can only be described in terms of lists and tuples. We encountered the problem of alternative values (see data set s11-2 in Section 5.3). Also optional values are difficult, the algorithm extracts a single field from each element extracted as parent of that field (see embedded catalog in Section 5.1), and when no element is present, some false positive might be returned. The fields of different tuples might even appear interleaved in the sequence that represents the document (for example tuples occupying the columns of a table), which is also not representable as an embedded catalog.

We believe it to be more flexible to add a tuple aggregation procedure on top of a single field extraction approach. This introduces a new learning task. Based on extracted fields and a user giving some examples of which fields belong to which tuple, a tuple aggregator should be learned. Different approaches are possible. If the fields of an example tuple share a common ancestor different from the ancestors from other fields, this common node could be learned for each tuple as a node extraction task. The (k, l)-contextual approach seems perfectly fit, and the extraction of the common node would be independent of the extraction of single fields, preventing accumulation of errors. Another approach is to find a regularity in the sequence of extracted elements. For Example 2, this sequence is [N(Stefan)], [S(Maurice)], [N(Anneleen)], [S(Hendrik)] (with N the marker for the name field, and S the marker for the supervisor field). An aggregator induction algorithm has to learn that a tuple consists of a subsequence of fields starting with an N-field, and ending with an S-field. This approach works also when tuples share all the same ancestors. It seems that these two approaches alone would solve most of the aggregation tasks in the WIEN data sets. This is no general solution, and needs further investigation. At least this framework allows for an easy exchange of aggregation policies.

9 Conclusion

We have introduced a new subclass of the regular unranked tree languages, called (k, l)-contextual tree languages, that is learnable from positive examples only. We applied this class of languages to the problem of wrapper induction by representing a wrapper as a language of marked trees. We situated our approach between other wrapper induction approaches, and made an in-depth comparison with two string approaches (STALKER, and BWI) and another unranked tree approach [17]. The latter corresponds to $(k, 2)$-contextual tree languages; they lack expressivity and their authors tweak the representation of the documents by annotating the path from the root to the target node. An experiment learning wrappers from a small set of positive examples shows that wrapper induction based on (k, l)-contextual tree languages usually yields a better wrapper.

Both our new algorithm as [17] need to tune parameters for each task. In [17] this is solved by evaluating wrappers on a sufficiently large set of completely annotated documents (representing positive and negative examples) to find the optimal parameter setting for a given extraction task. We developed a technique that learns a good parameter setting from a small set of positive and negative examples.

Another limitation of [17] was the need for an ad-hoc preprocessing step to identify a so called distinguishing context that in some applications is needed to disambiguate positive from negative examples. We developed a technique that preserves text nodes close to the target node when they occur in all examples.

We integrated the algorithm in an interactive system that allows a user to build a wrapper by selecting an initial positive example, and possibly a small number of false positives or false negatives, in sample documents. Experiments show that the resulting system is indeed able to learn a wrapper from a few positive and negative examples for a large number of extraction tasks, and compares favorably with regard to an active learning approach based on STALKER. Interestingly, the system indicates failure when the extraction task is not expressible as a (k, l)-contextual tree language. In this case, one could switch to more expressive languages, e.g., the tRPNI algorithm [5] that needs a set of completely annotated documents (so far we have not met an existing data set requiring this).

Acknowledgments

We wish to express our gratitude to Ion Muslea for his feedback on our questions regarding the STALKER system, and the Co-Testing approach.

We acknowledge the Fondazione Bruno Kessler for granting us the use of their TIES software, containing an implementation of the BWI algorithm.

References

1. H. Ahonen. *Generating grammars for structured documents using grammatical inference methods*. PhD thesis, University of Helsinki, Department of Computer Science, 1996.
2. D. Angluin. Inference of reversible languages. *Journal of the ACM (JACM)*, 29(3):741–765, 1982.
3. D. Angluin. Queries and concept-learning. *Machine Learning*, 2:319–342, 1988.
4. M. E. Califf and R. J. Mooney. Relational learning of pattern-match rules for information extraction. In *AAAI/IAAI '99: Proc. of the 16th National Conference on Artifi cial Intelligence and the 11th Innovative applications of AI conference*, pages 328–334. American Association for Artifi cial Intelligence, 1999.
5. J. Carme, A. Lemay, and J. Niehren. Learning node selecting tree transducer from completely annotated examples. In *International Colloquium on Grammatical Inference*, volume 3264 of *Lecture Notes in Artifi cial Intelligence*, pages 91–102. Springer Verlag, Oct. 2004.
6. B. Chidlovskii, J. Ragetli, and M. de Rijke. Wrapper generation via grammar induction. In *Proc. 11th European Conference on Machine Learning (ECML)*, volume 1810 of *Lecture Notes in Computer Science*, pages 96–108. Springer, Berlin, 2000.
7. D. Freitag. Information extraction from HTML: Application of a general machine learning approach. In *AAAI/IAAI '98: Proceedings of the fi fteenth national/tenth conference on Artifi - cial intelligence/Innovative applications of artifi cial intelligence*, pages 517–523. American Association for Artifi cial Intelligence, 1998.
8. D. Freitag and N. Kushmerick. Boosted wrapper induction. In *Proceedings of the Seventeenth National Conference on Artifi cial Intelligence and Twelfth Innovative Applications of AI Conference*, pages 577–583. AAAI Press, 2000.
9. D. Freitag and A. McCallum. Information extraction with HMMs and shrinkage. In *AAAI-99 Workshop on Machine Learning for Information Extraction*, 1999.
10. P. García. Learning k-testable tree sets from positive data. Technical Report DSIC/II/46/1993, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 1993.
11. P. García and E. Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(9):920–925, 1990.
12. E. M. Gold. Language identifi cation in the limit. *Information and Control*, 10(5):447–474, 1967.
13. G. Gottlob and C. Koch. Logic-based web information extraction. *SIGMOD Rec.*, 33(2):87–94, 2004.
14. C.-N. Hsu and M.-T. Dung. Generating fi nite-state transducers for semi-structured data extraction from the web. *Information Systems*, 23(8):521–538, 1998.
15. T. Knuutila. Inference of k-testable tree languages. In H. Bunke, editor, *Advances in Structural and Syntactic Pattern Recognition: Proc. of the Intl. Workshop*, pages 109–120, Singapore, 1993. World Scientifi c.
16. R. Kosala, H. Blockeel, M. Bruynooghe, and J. Van den Bussche. Information extraction from structured documents using k-testable tree automaton inference. *Data and Knowledge Engineering*, 58(2):129–158, 2006.
17. R. Kosala, M. Bruynooghe, H. Blockeel, and J. Van den Bussche. Information extraction from web documents based on local unranked tree automaton inference. In *International. Joint Conference on Artifi cial Intelligence (IJCAI)*, pages 403–408, 2003.
18. R. Kosala, J. Van den Bussche, M. Bruynooghe, and H. Blockeel. Information extraction in structured documents using tree automata induction. In *PKDD*, volume 2431 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2002.

19. N. Kushmerick, D. S. Weld, and R. B. Doorenbos. Wrapper induction for information extraction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.
20. R. McNaughton. Algebraic decision procedures for local testability. *Math. Systems Theory*, 8(1):60–76, 1974.
21. S. Muggleton. *Inductive Acquisition of Expert Knowledge*. Addison-Wesley, 1990.
22. I. Muslea, S. Minton, and C. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:93–114, 2001.
23. I. Muslea, S. Minton, and C. Knoblock. Active learning with strong and weak views: A case study on wrapper induction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 415–420, 2003.
24. S. Raeymaekers and M. Bruynooghe. Extracting information from structured documents with automata in a single run. In *Proc. 2nd Int. Workshop on Mining Graphs, Trees and Sequences (MGTS 2004, Pisa, Italy)*, pages 71–82, Pisa, Italy, 2004. University of Pisa.
25. S. Raeymaekers and M. Bruynooghe. Parameterless information extraction using (k,l)-contextual tree languages. In *BNAIC 2004 - Proceedings of the 16th Belgian-Dutch Conference on Artificial Intelligence*, pages 211–218, 2004.
26. S. Raeymaekers and M. Bruynooghe. Wrapper induction: Learning (k,l)-contextual tree languages directly as unranked tree automata. In *Proc. Int. Workshop on Mining and Learning with Graphs (MLG-2006)*, pages 197–204, Berlin, Germany, 2006.
27. S. Raeymaekers, M. Bruynooghe, and J. Van den Bussche. Learning (k, l)-contextual tree languages for information extraction. In *European Conference on Machine Learning (ECML)*, volume 3720 of *Lecture Notes in Computer Science*, pages 305–316, 2005.
28. J. R. Rico-Juan, J. Calera-Rubio, and R. C. Carrasco. Probabilistic k-testable tree languages. In A. Oliveira, editor, *Proceedings of 5th International Colloquium on Grammatical Inference (ICGI 2000)*, volume 1891 of *Lecture Notes in Computer Science*, pages 221–228, 2000.
29. R. E. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.
30. S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.