Bottom-Up Learning of Logic Programs for Information Extraction from Hypertext Documents

Bernd Thomas

Universität Koblenz-Landau, Institut für Informatik
bthomas@uni-koblenz.de

Abstract. We present an inductive logic programming bottom-up learning algorithm (BFOIL) for synthesizing logic programs for multi-slot information extraction from hypertext documents. BFOIL learns from positive examples only and uses a logical representation for hypertext documents based on the document object model (DOM). We briefly discuss several BFOIL refinements and show very promising results of our IE system LIPX in comparison to state of the art IE systems.

1 Introduction

In the last decade several techniques and systems based on relational learning in the area of information extraction (IE) have been developed [10] . Though a handful approaches [1, 2, 5] exist which capture the idea of bottom-up and top-down rule learning inspired by inductive logic programming (ILP) [12], it is surprising that almost no system [8] tries to follow a pure logical ILP based approach. ILP in general offers broad varieties to be adapted to different problem domains by simply changing the problem representation and/or the hypothesis language. Our aim is to develop an algorithm for learning multi-slot wrappers for hypertext documents, based on logic programming and ILP concepts. This technology can easily be extended with additional information on the representational level (document pre-processing and hypothesis language) and algorithmic level (semantic least general generalization operators).

In Section 2 and 3 we introduce a DOM [4] based representation for hypertext documents and relational representation of text examples. Section 4 briefly explains the hypothesis language and derived example descriptions used for latter bottom-up learning. The **B**ottom-up **F**irst **O**rder **I**nductive **L**earning algorithm and results are presented in Section 5 and 6.

2 Document Representation

Throughout this paper we will focus on HTML documents. It should be noted that the approach presented in this paper is easily adaptable to XML or similar tag-based languages. In order to capture and model the syntactical and hierarchical aspects of HTML and XML documents we define the concept of TDOM-trees, which is strongly related to that of a *document object model* (DOM-tree). A node in a TDOM-tree consists of

four features: a document reference (D_{id}), a node identifier (n_{id}), the corresponding token t describing the document text denoted by the node and an ordered list of child node identifiers ($[ch_1, \dots, ch_n]$). Thus we represent a node in a TDOM-tree as a term $node(D_{id}, n_{id}, t, [ch_1, ch_2, \dots, ch_n])$. The basic intention of tokens is, like in most other approaches, to group symbols from the text separated by white spaces or other separators to typed words like integer, date, html-tags etc. Each token is represented as a term with a list of feature-value pairs, which is given by: $token([f_1, v_1], \dots, [f_n, v_n])$, where f_i is an arbitrary feature name and v_i is an arbitrary feature value with $i = 1, n$. For example $Tok(<img\ src = "a.jpg">) = \{token([(ttype, html), (value,'<img\ src = "a.jpg">'), (spos, 0), (epos, 16), (tag, img), (src,'a.jpg')])\}$.

Node identifiers are terms representing a path from the root node to a node in the TDOM. To illustrate the idea of node identifiers assume every node in a tree is assigned a unique number. The function $child : N_0 \times N_0 \rightarrow N_0$ computes for a given node number i and $n \in N_0$ the n-th unique child node of i. For example the term $child(child(child(root, 1), 0), 3)$ refers to the fourth child of the first child of the second child of the root node in the TDOM. For better readability and later handling we use a prolog list notation $[1, 0, 3]$, leaving out the root node, to denote node identifiers. Hence a *node identifier* is used to assign a unique term to each node in a TDOM. It also provides information about the position in the TDOM-tree. In fact, the notation of node identifiers is strongly related to the *Dewey-Notation* [18]. A leaf node in a DOM-tree

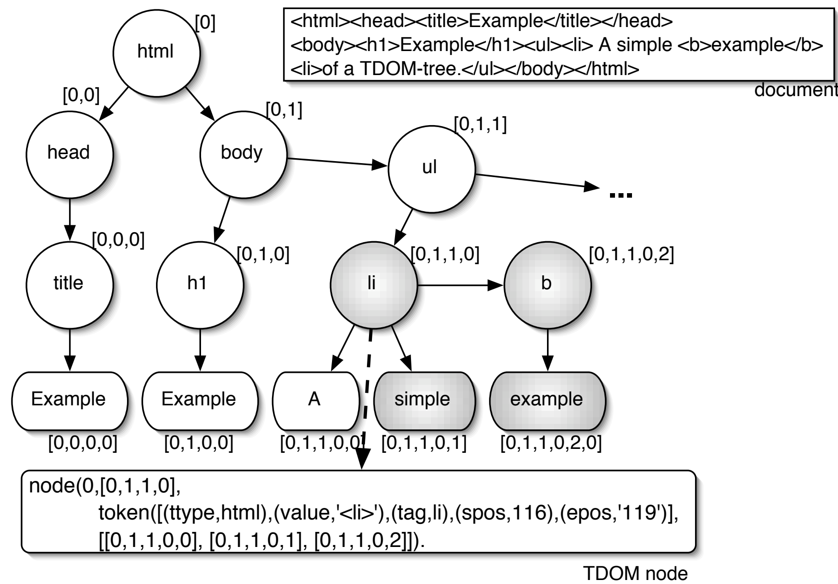

Fig. 1. HTML document, simplified TDOM-tree, TDOM node and span ([0,1,1,0],1,2)

represents text appearing at the "surface" of the hypertext document. For example a whole paragraph may be associated with one leaf node in a DOM-tree. In many cases, this representation is not accurate enough for IE tasks. We modify the concept of a DOM-tree such that a leaf node in a DOM-tree becomes many leaf nodes in a TDOM-tree. Each of these nodes represent one token from the text.

Given this notation, an arbitrary HTML document D can be represented as a set of ground unit clauses describing a TDOM model of D. $\mathcal{T}(D_i)$ denotes the TDOM of D with $D_{id} = i$. A $\mathcal{T}(D_0)$ representation for an example HTML page is shown in Figure 1. To be able to compare node identifiers we define the following order relation. A node identifier n_i is smaller than a node identifier n_j written $n_i < n_j$ iff $\exists x \in N_0 : n_j.x > n_i.x \wedge \forall y \in N_0 : y < x$ it holds that $n_j.y = n_i.x$ where $n_i.n$ denotes the n-th child number (starting from left) of a node identifier. Two node identifiers n_i and n_j are equal if they have the same length and $n_i \not< n_j \wedge n_j \not< n_i$. For example: $[0,0,3] < [0,2]$.

Node identifiers have nice properties for wrapper-learning. Similar to expressions in the XPATH language [19] node identifier expressions can be used to refer to more than one node by the use of variables. The node identifier $[0,1,1,X]$ refers to every child node of the `<ul>` environment of Figure 1. For example, the term $[X,3]$ refers to all child nodes of the root nodes with at least 3 child nodes. It is important to point out that variables can only be substituted by one value and not by partial node identifier expression like $[0,1]$. Furthermore additional constraints can be introduced by using one variable more than once (e.g. $[0,X,2,X,0]$) or more than one variable (e.g. $[Y,X,2,X,Y]$. Then pattern variables with the same name are not treated disjunctively and thus have to be instantiated with the same value. In fact, in the XPATH query language such expressions can only be expressed by means of iterative programming language constructs like for-loops and thus are not as elegant and compact and easy to handle.

This notation makes it easy to generalize on node identifiers by means of lgg operations [15]. Assume one text example is located in one document in node $[0,1,1,0,0]$ and in the other document in node $[0,1,1,1,0]$. A reasonable first step in learning an extraction rule is the assumption that all nodes described by the generalized node identifier $[0,1,1,X,0]$ are good extractions.

3 Example Representation

One essential concept of our approach is that of *span*. Informally spoken a span determines a subtree in a TDOM-tree. We pick up the idea mentioned by [3] where a span is defined as a triple consisting of a node identifier N and a left and right delimiter L,R. Delimiters determine the left and right boundaries of an interval of child nodes contained in a span. For example the span $([0,1,1,0],1,2)$ of the example TDOM (Figure 1) refers to the set of node identifiers $\{[0,1,1,0,1],[0,1,1,0,2],[0,1,1,0,2,0]\}$. More precise: a span $S = (N,R,L)$ is the set of all reachable descendant nodes starting at the i-th child node of node N with $i = R..L$. In general we assume a depth first traversal to enumerate all nodes of a span to ensure the left to right order of the text at the surface of a document.

A *minimal example span MS* for a given text T is the span with the least cardinality including the text T. For example let T be a text fragment from the document (Figure 1) like `simple example` and S_1 be a span with $([0,1,1],0,1)$ and S_2 be the span from

our previous example. Clearly both S_1 and S_2 contain T but $card(S_1) > card(S_2)$ and therefore S_2 is the only existing minimal example span of T with respect to the example TDOM because: $\neg\exists S' : card(S') < card(S_2)$ where S' is a span including T.

For the rest of the paper we focus on multi-slot extraction tasks, where a text example t with n slots consists of a tuple of texts $<t_1,\ldots,t_n>$ taken from a document D. The initial example set of text tuples is denoted by E_T^D. Given D and t we define the *example representation* of t with respect to D as $e_t^D :=<<s_1,\ldots,s_n>,<t_1,\ldots,t_n>>$ where s_i is the minimal example span of t_i with $i = 1..n$ in $\mathcal{T}(D)$. For later purposes we define the notion of a *validation set* given by $VS(E_T^D, p) = \{p(D_{id},[s_1,\ldots,s_n],[t_1,\ldots,t_n]) \mid t \in E_T^D \wedge e_t^D =<<s_1,\ldots,s_n>,<t_1,\ldots,t_n>>\}$

Further we take some assumptions according to the presentation of examples: 1) t_1 to t_n do not define a particular order of occurrences of t_i in D (i.e. we can not follow that t_i occurs before t_{i+1} in D). 2) Each t_i is associated with an intended semantics (e.g. t_i describes the ZIP code) 3) Missing slot fillers in the text (e.g. no ZIP field or placeholder stated in the text) or empty slot fillers (e.g. there is a ZIP placeholder but no code is given) are represented by the empty string "".

4 Hypothesis Language

This section will cover three questions: given an example representation which important relational properties can be observed (Section 4.1)? How can these observations be represented? How are these representations used to define a hypothesis language for inductive learning of extraction rules (Section 4.2)?

4.1 Observing Example Properties

We write $s.n$, $s.l$ and $s.r$ to refer to the components of a span $s := (n,l,r)$. Given an example representation e_t^D we investigate each tuple argument t_i and its span s_i according to the following four levels. Note, the following predicates can be exchanged by arbitrary other ones describing relational information regarding the training examples.

structural level: the position of a span s_i and its neighbor nodes are investigated:
$xpath(D_{id},s,tl)$ holds if D_{id} is a document id (Section 2), s is a span and tl is the list of tokens associated with each node following the path from the root node to the node of s.
$xspan(D_{id},s,tl)$ holds if tl is the associated list of tokens of all nodes of span s.
$xright_brother(D_{id},n,t_r)$ holds if n is a node identifier and t_r is the associated token of the right neighbor node of n. Analogously we define a left brother predicate.
textual or content level: a relation between the example text, its tokens associated with the leaf nodes and its span is defined:
$span_text_and_tokens(D_{id},s,t,tl)$ holds if tl is the list of tokens associated with all leaf nodes of span s for text t.

delimiter level: predicates to incorporate a widespread idea of IE approaches to learn right and left delimiters of relevant text parts are defined:

$start_end_nodes(D_{id}, t, n_l, n_r)$ holds if n_l is the start node and n_r the end node of text t in $\mathcal{T}(D)$ referred to by D_{id}.

$xpredecessor(D_{id}, n, n_i, tl)$ holds if the token list tl contains the tokens associated with the first n nodes we meet going backwards in a depth first search[1] to n_i. Analogously we define *xsuccessor* to collect all n successor tokens we meet by a depth first traversal after having met n_i. We call n the *context distance*.

```
extract(D, [[0,1,0,9,X,5] : 0 : R], [[E|ER]]) :-
    xpath(D, [0,1,0,9,X,5],
          [token([(ttype,html), (value,'<html>'), (tag,html), (spos,'0'), (epos,'5')]),
           token([(ttype,html), (value,'<center>'), (tag,center), (spos,'136'), (epos,'143')]),
           token([(ttype,html), (value,'<nobr>'), (tag,nobr), (spos,'146'), (epos,'151')]),
           token([(ttype,html), (value,'<table border cellpadding,2>'), (tag,table), (border,''),
                       (cellpadding,'2'), (spos,'1090'), (epos,'1117')]),
           token([(ttype,html), (value,'<tr>'), (tag,tr), (spos,V19), (epos,V20)]),
    unify(C1, 0), unify(C2, R), member(C1, [0]), member(C2, [0, 1, 2]),
    xspan(D, [0,1,0,9,X,5] : 0 : R,
          [token([(ttype,html), (value,V1), (tag,td), (align,V2), (spos,V3),(epos,V4)]), T1|TR1]),
    xleft_brother(D, [0,1,0,9,X,5],
          [token([(ttype,html), ((value,'<td align,right>')), (tag,td), (align,right), (spos,V5), (epos,V6)])]),
    xright_brother(D, [0,1,0,9,X,5],
          [token([(ttype,html), (value,V7), (tag,td), (align,V8), (spos,V9), (epos,V10)])]),
    span_text_and_tokens(D, [0,1,0,9,X,5] : 0 : R, [E|ER], [T1|TR1]),
    start_end_nodes(D, [0,1,0,9,X,5] : 0 : R, [0,1,0,9,X,5,0], [0,1,0,9,X,5,R]),
    xpredecessor(D, 7, [0,1,0,9,X,5,0],
          [token([(ttype,html), value,V1, tag,td, align,V2, spos,V3, epos,V4),
           token([(ttype,html), (value,'<td align,right>'), (tag,td), (align,right), (spos,V5), (epos,V6)]),
           token([(ttype,html), (value,'<td align,right>'), (tag,td), (align,right), (spos,V11), (epos,V12)]),
           token([(ttype,html), (value,'<td align,right>'), (tag,td), (align,right), (spos,V13), (epos,V14)]),
           token([(ttype,html), (value,'<td align,left>'), (tag,td), (align,left), (spos,V15), (epos,V16)]),
           token([(ttype,html), (value,'<td>'), (tag,td), (spos,V17), (epos,V18)]),
           token([(ttype,html), (value,'<tr>'), (tag,tr), (spos,V19), (epos,V20)])]),
    xsuccessor(D, 7, [0,1,0,9,X,5,R],
          [T0, token([(ttype,V21), (value,V22), (spos,V23), (epos,V24)]), T2, T3, T4, T5, T6]),
    xsmallest_common_span(D, [[0,1,0,9,X,5] : 0 : R], [0,1,0,9,X,5] : 0 : R,
          [token([(ttype,html), (value,V1), (tag,td), (align,V2), (spos,V3), (epos,V4)])]).
```

Fig. 2. learned single-slot rule for QS-vol

relational span level: to figure out relations between spans we define:

$xsame_span_node(D_{id}, s_i, s_j)$ holds if n_i and n_j of spans $s_i = (n_i, l_i, r_i)$ and $s_j = (n_j, l_j, r_j)$ are unifiable.

$xnode_less(D_{id}, n_i, n_j, dist)$ holds if $n_i < n_j$. Where *dist* is the list of differences between the components of n_j and n_i (e.g. $xnode_less(0, [1,4,0], [2,3,0,2], [1,-1,0])$). Analogously we define *xnode_greater*.

[1] This captures the idea to interpret the document as a sequence of tokens rather than a tree, and we investigate the n preceding tokens of the token associated with n_i.

$overlapping_span(D_{id}, s_i, tl_i, s_j, tl_j)$ holds if $(s_i.l < s_j.l) \wedge (s_i.r \geq s_j.l) \wedge (s_i.r \leq s_j.r)$ where tl_i and tl_j are the corresponding token lists of s_i and s_j.
$span_in_span(D_{id}, s_i, s_j)$ holds if span s_i is a subtree of span s_j.
$xsub_related_span(D_{id}, s_i, s_j)$ holds if $s_j.n$ is a prefix of $s_i.n$ (e.g. $[1,2]$ is a prefix of $[1,2,3]$).
$xsmallest_common_span(D_{id}, [s_1, \ldots, s_n], s_x, tk_x)$ holds if s_x is the smallest span (wrt. to its number of nodes) in D_{id} such that each span s_i with $i = 1..n$ is a subtree of s_x and tk_x is the associated token with $s_x.n$.

4.2 Clause Descriptions of Examples

Now that we have defined predicates for the description of text example properties based on the representation of a TDOM, we introduce the concept of a clause description $CD(e_t^D)$ for an example representation e_t^D. In terms of *extensional* and *intentional object languages* a clause description of an example is an *intentional object description*. Furthermore we use the same language for the description of objects and hypotheses.

Let $\mathcal{L}_H$ be the set of predicates introduced in Section 4.1. We call this the *hypothesis language* which is used later for construction of rules. This is in analogy to standard ILP algorithms like FOIL [16]. It should be noted that the hypothesis language can be freely chosen. Furthermore let us assume that a logic program $P_{\mathcal{L}_H}$ is given that implements the intended semantics of the predicates in $\mathcal{L}_H$. To denote the union of $P_{\mathcal{L}_H}$ and $\mathcal{T}(D_i)$ we write $P^D_{\mathcal{L}_H}$. Now we can define $CD(e_t^D) = \{l'_\sigma \mid P^D_{\mathcal{L}_H} \vdash l'_\sigma$ with $l \in \mathcal{L}_H$ and l' is l instantiated according to its given semantics with $(s_i, t_i) \in e_t^D$ and σ calculated answer substitution$\}$. Here $\vdash$ denotes the logical derivation operator and we assume a standard logical calculus (e.g. SLD-Resolution [11]).

Finally we define E_u^+ to be the set of clause descriptions for a given set of examples as $E_u^+ = \bigcup_{t \in E_T^D} CD(e_t^D)$. Additionally we extend every $CD(e_t^D)$ with a special predicate, the *rule head* defined as $extract(D_{id}, [s_1, \ldots, s_n], [t_1, \ldots, t_n])$, where every s_i and t_i is instantiated with the associated argument from e_t^D. Then $CD(e_t^D)$ forms a ground instantiated rule of the form $extract(D_{id}, [s_1, \ldots, s_n], [t_1, \ldots, t_n]) \leftarrow l_1, \ldots, l_n$ with $l_i \in \mathcal{L}_H$. Since we focus only on learning non recursive horn clauses, we do not have to use negation operators for the body literals and consider a marked predicate (e.g. *extract*) to build the head and all other literals in $CD(e_t^D)$ to form the body of a rule. Thus every $CD(e_t^D)$ is one rule describing exactly one text example with respect to D and $\mathcal{L}_H$. Accordingly computing all answers to the query $P^D_{\mathcal{L}_H} \cup E_u^+ \vdash extract(D_{id}, [s_1, \ldots, s_n], [t_1, \ldots, t_n])$ provides the validation set $VS(E_T^D, extract)$ (Section 3).

5 BFOIL Algorithm

The central idea of BFOIL is to learn in a bottom-up fashion from positive examples only a set of rules by means of least general generalization techniques [15]. In contrast to the standard *Top-Down* learning approaches starting with the most general hypothesis BFOIL starts with a set of ground rules (clause descriptions) as initial hypothesis and tries to generalize these clause sets by means of lgg operations. The term *clause-lgg* denotes the lgg of two clauses C_1 and C_2 defined as $clause\text{–}lgg(C_1, C_2) = \{lgg(l, m) \mid$

Algorithm 5.1 Basic BFOIL algorithm

Require: $P =$ logic program ; $E_T^D =$ positive examples
$E_{learn}^+ \subseteq E_T^D$; $E_u^+ = \bigcup_{t \in E_{learn}^+} CD(e_t^D)$
1: $LearnedRules \leftarrow \emptyset$
2: **while** $E_u^+ \neq \emptyset$ **do**
3: $Rule \in E_u^+$
4: $E_u^+ \leftarrow E_u^+ \setminus \{Rule\}$
5: $ProblemSet \leftarrow \emptyset$
6: **while** $E_u^+ \neq \emptyset$ **do**
7: $X \in E_u^+$
8: $R \leftarrow clause_lgg(Rule, X)$
9: **if** $apply(R, P, E_T^D).fp > 0$ **then**
10: $ProblemSet \leftarrow ProblemSet \cup \{X\}$
11: **else**
12: $Rule \leftarrow R$
13: $E_u^+ \leftarrow E_u^+ \setminus \{X\}$
14: $LearnedRules \leftarrow LearnedRules \cup \{Rule\}$

$l \in C_1 \wedge m \in C_2 \wedge lgg(l,m)$ is defined$\}$. In general the clause-lgg of two clauses has to be reduced, in the sense that redundant literals under θ-subsumption have to be removed. Since it is obvious that calculating the clause-lgg of E_u^+ results in one rule that over-generalizes with high probability, BFOIL inductively tries to partition E_u^+ into sets of clauses $C_i \subseteq E_u^+$ such that the clause-lgg of each C_i forms a new rule that does not produce any false positive predictions (extractions). Since we only learn from positive

Function 5.2 apply(R,P,V) with false positive calculation

Require: R :=rule ; $P =$ logic program ; $V =$ examples
1: $A \leftarrow \{R_{head}\sigma \mid P \cup R \vdash R_{head}\sigma$ with σ answer subst.$\}$
2: $fp \leftarrow |A \setminus (VS(V, R_{head}) \cap A)|$

examples, standard techniques to determine false predictions during the learning phase (validation on negative example sets) are not applicable. To yield good rules anyhow, it is essential to estimate the correctness of rules during learning. Thus we assume that the set E_T^D is exhaustively enumerated. This means every intended extraction from D is contained in E_T^D. Then we can conclude that if a rule extracts a tuple t from D with $t \notin E_T^D$ it is false positive. This introduces a *closed world assumption* [17] similar view on extraction examples and the absence of negative training data.

This seems to be a very strong restriction which requires tedious labeling. But since our approach does not need many examples (5-30 training examples Section 6 Figure 6) only a small number of documents have to be labeled.

In general an IE learning task has to deal with multiple documents $D_1 \ldots D_n$ and examples drawn from $D_1 \ldots D_n$ then we define $E_T^D = \bigcup_{i=1}^n E_T^{D_i}$. Additionally we assume that the logic program P is an implementation of $\mathcal{L}_H \cup (\bigcup_{i=1}^n \mathcal{T}(D_i))$. Algorithm 5.1 shows the basic BFOIL algorithm and Function 5.2 the function *apply* for calculating false positives. In the best case basic BFOIL returns one rule, the clause-lgg of E_u^+. Experiments showed that this happens if examples are identical wrt. to their structural

properties in a TDOM. In the worst case basic BFOIL just memorizes each clause in E_u^+. This might happen if examples are too different wrt. to the expressiveness of $\mathcal{L}_H$ and the clause-lgg leads to over-generalized rules.

5.1 BFOIL Refinements

The results of using basic BFOIL to multi-slot extraction are not satisfying. Imagine a clause $C_1 = CD(e_t^D))$ with $\{xpath(0,[1,2],[\ldots]),\ldots,xpath(0,[1,3,1],[\ldots]),\ldots\}$. The intention of this clause is that the first literal describes path features of the first argument and the second literal describes path features of the second argument of an example.

Algorithm 5.3 Consistent BFOIL algorithm

Require: $P =$ logic program ; $E_T^D =$ positive examples
$E_{learn}^+ \subseteq E_T^D$; $E_u^+ = \bigcup_{t \in E_{learn}^+} CD(e_t^D)$

1: $LearnedRules \leftarrow \emptyset$
2: **while** $E_u^+ \neq \emptyset$ **do**
3: $Rule \in E_u^+$
4: $E_u^+ \leftarrow E_u^+ \setminus \{Rule\}$
5: $ProblemSet \leftarrow \emptyset$
6: $C \leftarrow \emptyset$
7: **while** $E_u^+ \neq \emptyset$ **do**
8: $X \in E_u^+$
9: $R \leftarrow clause_lgg(Rule, X)$
10: **if** $(apply(R,P,E_T^D,E_{learn}^+,C).fp > 0)$
or (not $apply(R,P,E_T^D,E_{learn}^+,C).consistent$) **then**
11: $ProblemSet \leftarrow ProblemSet \cup \{X\}$
12: **else**
13: $Rule \leftarrow R$
14: $C \leftarrow C \cup \{X\}$
15: $E_u^+ \leftarrow E_u^+ \setminus \{X\}$
16: $LearnedRules \leftarrow LearnedRules \cup \{Rule\}$

Calculating the clause-lgg of C_1 and C_2 generalizes each *xpath* literal in C_1 with each *xpath* literal in C_2. This is not what we want. Only the lgg of *xpath* literals describing the same argument i should be calculated from both clauses. With a simple syntactic transformation before the calculation of an lgg and re-transformation before evaluation of a generalized clause (rule) we can still use the standard lgg operation for learning. Adding a prefix arg_i to every predicate symbol of each literal in E_u^+ prevents the lgg to generalize from non-intended literals. This *prefix protection* is more an issue of representation than a refinement of the BFOIL algorithm.

The basic BFOIL algorithm is not consistent (e.g. learned rules may not cover examples from E_{learn}^+). Imagine two examples e_1 and e_2. The second argument of e_1 is empty. Due to P and $\mathcal{L}_H$ the clause description for e_1 would not contain literals for the description of argument 2 to reduce the complexity associated with *empty substitutions*. Because of the absence of these literals the clause-lgg eliminates the literals for argument 2 from clause two. It is possible that the new rule still covers e_1 and does not produce any false positives, but does not cover e_2 anymore. For this reason, we

keep track of examples that had been used successfully for learning the current rule (line 14 Algorithm 5.3). Every rule refinement (line 9) must cover all examples that have been successfully used in previous learning steps (line 10). Function 5.4 implements this test. A third refinement of BFOIL is the modification of the clause-lgg

Function 5.4 apply(R,P,V,L,C) with consistency check

Require: R :=rule ; P = logic program ;
$L \subseteq V$ = examples ; C = example descriptions
1: $A \leftarrow \{R_{head}\sigma \mid P \cup R \vdash R_{head}\sigma$ with σ answer subst.$\}$
2: $fp \leftarrow |A \setminus (VS(V, R_{head}) \cap A)|$
3: $consistent \leftarrow true$
4: **while** $C \neq \emptyset \wedge consistent$ **do**
5: $c_e \in C$
6: $e \in L \wedge e$ is described by c_e
7: **if** $e \notin A$ **then**
8: $consistent \leftarrow false$
9: **else**
10: $C \leftarrow C \setminus \{c_e\}$

operator. Therefore we introduce the concept of a *semantic lgg operator*. Semantic lgg operators are closely related to the chosen hypothesis language and example representation in general. The key idea is to guide the lgg operation by additional knowledge to prevent over-generalization. For example the lgg of spans and the generalization of *xspan* literals tend to blow up the search space. The lgg of $xspan(0,([1,2,3],3,6),[...])$ and $xspan(0,([1,2,3],1,10),[...])$ is $xspan(0,([1,2,3],X,Y),[...])$ which is obviously to general from a practical point of view. For this reason we define additional semantical lgg operators. These operators provide semantical based generalization by adding special literals to the lgg of two clauses. We denote a semantic lgg operator similar to an inference rule:

$$\frac{C1 \setminus \{xspan(D_1,(N_1,L_1,R_1),TL_1)\} \quad C_2 \setminus \{xspan(D_2,(N_2,L_2,R_2),TL_2)\}}{\{member(L,[L_1,...,L_2]), member(R,[R_1,...,R_2])\} \cup CL}$$

with $CL = clause_lgg(C1,C2)$ and $xspan(D,(N,L,R),TL) \in CL$.

Extending the standard clause-lgg with semantic lgg operators can reduce the search space significantly, resulting in faster learning and extraction times. Especially if spans in a document are huge, the insertion of the member predicates are of practical relevance. Instead of considering all possible instances for the left and right delimiter of the span, they are constrained to take only values between the smallest and the greatest value seen so far. All results presented in this paper have been generated by using only one semantic lgg operator, that is for the *xspan* literal.

6 Results and Conclusion

We tested the BFOIL algorithm with our extraction system LIPX on the RISE repository [13]. RISE contains document resources with an extraction task description taken from various IE research papers and projects. Most publications refer to these problem cases as kind of standard tests. Unfortunately not all approaches give a complete overview

of their results with respect to *precision* and *recall* values. We focused on extraction tasks from HTML documents only and learned multi-slot extraction rules for HTML resources as described in the RISE repository.

All tests were ran using a fixed number of randomly drawn examples to perform 20 learning and test runs for each problem class. The settings are shown in the first table of Figure 6 where t = no. total tuples; e^+ = no. examples and r = average no. of learned rules. For each problem class the learning examples were randomly drawn from one half of the available documents. The testing set consisted of all documents, but only the data tuples not used for learning were considered. An extraction was counted as correct when all of its slots where correctly extracted. Values for precision, recall and F1 are displayed in percentages, all other values in totals. For all tests we used the hypothesis language described in Section 4.1 with context distance $n = 7$. Figure 6 shows the best F1 (harmonic mean of precision and recall) values.

LIPX	multi-slot			single-slot			
	t	e^+	r	slot	t	e^+	r
CS				name	1151	20	7.2
Bigbook	204	30	1.6				
IAF	84	30	13.3	altname	11	6	4.7
				org	53	20	8.7
LA	157	5	1.4	CC	144	10	2.4
Okra	3335	25	1				
Quote	25	14	6.8	date	21	6	1
				vol	25	10	1.6
Zagats	140	30	1.3	addr	140	30	1.3

Fig. 3. test settings

Comparing LIPX results with other multi-slot IE-systems is not straightforward, because almost all systems set up different evaluation scenarios with respect to the number of examples, their selection criteria and the number of test iterations. The first table of Figure 6 shows the results (median) for single and multi-slot learning in comparison.[2] to the systems SoftMealy [7], Stalker [14] and Wien [9]. Even though LIPX is developed for multi-slot tasks we tested it on single slot extraction tasks to provide a comparison to one state of the art single slot extraction approach (BWI) of [6]. These results are listed in the second part of Figure 6. While learning single slot wrappers supersedes the *relational span level* predicates the single slot learning results also underline the high precision values observed with multi-slot learning. In 5 out of 7 cases LIPX shows better or equal precision values than BWI and BWI HMM. This is not too surprising, because in the worst case BFOIL only memorizes the examples. This does not happened with these test cases, but in two cases the F1 rates due to the low recall rate are not acceptable. There are mainly two interacting reasons for this behavior, which build a general observation for multi and single-slot learning. First, BFOIL seems to yield bad recall rates if only a few examples are present and those differ strongly regarding their relational description (Quote, IAF-altname). Secondly, some tests where run with only a few training examples, because of the bad runtime behavior caused by BFOIL's

LIPX multi-slot	t	e^+	r	Pre	Rec	F1
Bigbook	204	30	1	100	98.3	99.1
IAF	84	30	12	85	31.5	46
LA	157	5	2	100	25.1	40.2
Okra	3335	25	1	100	95.4	97.7
Quote	25	14	7	100	63.6	77.8
Zagats	140	15	1	98.2	97.3	97.7

Fig. 4. best F1 multi-slot results

[2] All values for SoftMealy, Stalker and Wien are taken from [7].

naive answer set computing for each generalized rule. Consequently the recall rate was low (LA, LA-cc, CS-name).

The presented approach offers a wide variety for extensions by modifying the token representation of text units for richer semantic text pre-processing. This allows to incorporate linguistic or additional general semantic information. By modification of the underlying hypothesis language we can adapt the presented approach to other mark up languages or focus on different relationships than those stated in this paper. Using natural language tools (e.g. part of speech tagger) for the pre-processing of documents in combination with an XML representation of such pre-processed documents also allows us to apply our methods to natural language texts. By extending the BFOIL algorithm with additional semantic lgg operators the hypothesis search space can be constrained and runtime behavior improved. An additional modification to increase the recall rate is, to accept rules that cover a small number of false positives. This modification was not tested yet. But it is easily accomplished by incorporating a threshold (e.g. if the percentage of false positive extractions is below 0.03 % (algorithm 5.1 line 9)). These observations show that all results presented in this paper depend strongly on the chosen hypothesis language and the degree of additional information chosen for the representation of TDOM nodes. So far we only made experiments with the one mentioned in Section 4.1 without any fine tuning (e.g. context distance, sem-lgg). LIPX shows partially bad learning time results, which clearly stems from the combinatorial explosion while applying a rule that became too general during the learning process. In fact, evaluating each new rule by computing the answer set for it leads to this problem. Thus we are doing research on using more efficient proof procedures than SLD-Resolution, clustering of example description rules and extending BFOIL with specification operators to minimize this problem. To summarize the capabilities: LIPX can learn single and multi slot wrappers for HTML or XML documents. It can handle slot fillers occurring in varying orders in the texts and it

multi-slot comparison	Pre	Rec	F1	Pre	Rec	F1
	BigBook			IAF		
LIPX	100	89.3	94.4	84	19.4	32
SoftMealy		100			41 to 99	
Stalker		97			85 to 100	
Wien	100	100	100	too hard		
	Quote			Okra		
LIPX	100	24	38.8	100	88.1	93.7
SoftMealy		85			100	
Stalker		79			97	
Wien	too hard				100	
	LA			Zagats		
LIPX	100	12.3	21.8	97.9	85.5	90.6
single-slot comparison	Pre	Rec	F1	Pre	Rec	F1
	CS name			LA cc		
LIPX	95.6	9.7	16.9	96.7	54.8	70.4
BWI	77,1	31.4	44.6	99.6	100	99.8
BWI HMM	41.3	65	50.5	98.5	100	99.2
	IAF altname			IAF org		
LIPX	100	20	33.3	77.5	37.7	51.3
BWI	90.9	43.5	58.8	77.5	45.9	57.7
BWI HMM	1.7	90	3.3	16.8	89.7	28.3
	QS date			QS vol		
LIPX	100	100	100	100	92.9	96.3
BWI	100	100	100	100	61.9	76.5
BWI HMM	36.3	100	53.3	18.4	96.2	30.9
	Zagats addr					
LIPX	98	93.6	95.4			
BWI	100	93.7	96.7			
BWI HMM	97.7	99.5	98.6			

Fig. 5. median results and comparison

can handle slots that may be empty, missing or nested. Though the presented approach shows very promising results its runtime behavior is a major subject for improvement. Nevertheless the pure logic programming motivated and based technique to learn multi-slot wrappers, the general method of lgg operations for learning and its independency of the application domain, are auspicious properties.

References

1. M. E. Califf. *Relational Learning Techniques for Natural Language Information Extraction*. PhD thesis, University of Texas at Austin, August 1998.
2. F. Ciravegna. Learning to Tag for Information Extraction from Text. In *Workshop Machine Learning for Information Extraction, European Conference on Artifical Intelligence ECCAI*, August 2000. Berlin, Germany.
3. W. Cohen, M. Hurst, and L. S. Jensen. A flexible learning system for wrapping tables and lists in html documents. In *The Eleventh International World Wide Web Conference WWW-2002*, 2002.
4. W3C, Document Object Model (DOM) Level 2 Core Specification , 2000. Version 1.0 `http://www.w3.org/TR/DOM-Level-2-Core/`.
5. D. Freitag. *Machine Learning for Information Extraction in Informal Domains*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, November 1998.
6. D. Freitag and N. Kushmerick. Boosted Wrapper Induction. In *Proceedings of the Seventh National Conference on Artificial*, pages 577–583, July 30 - August 3 2000. Austin, Texas.
7. C.-N. Hsu and C.-C. Chang. Finite-State Transducers for Semi-Structured Text Mining. In *Workshop on Text Mining IJCAI 99*, 1999.
8. M. Junker, M. Sintek, and M. Rinck. Learning for Text Categorization and Information Extraction with ILP. In *Proc. Workshop on Learning Language in Logic*, June 1999. Bled, Slovenia.
9. N. Kushmerick. *Wrapper Induction for Information Extraction*. PhD thesis, University of Washington, 1997.
10. N. Kushmerick and B. Thomas. *Intelligent Information Agents R&D in Europe: An AgentLink perspective*, chapter Adaptive Information Extraction: A Core Technology for Information Agents. Springer, 2002.
11. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2 edition, 1987.
12. S. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 1994.
13. I. Muslea. The RISE Repository, 1999. `http://www.isi.edu/~muslea/RISE/`.
14. I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In O. Etzioni, J. P. Müller, and J. M. Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 190–197, Seattle, WA, USA, 1999. ACM Press.
15. G. Plotkin. A note on inductive generalization. *Machine Intelligence*, (5):153–163, 1970. Edinburgh Univ. Press.
16. J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
17. R. Reiter. On Closed World Data Bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*. Plenum Press, New York, 1978.
18. M. L. Scott. *Dewey Decimal Classification: A Study Manual and Number Building Guide*. Libraries Unlimited, 1998.
19. W3C, *xpath specification*, 1999. `http://www.w3.org/TR/xpath`.