Extracting partial structures from HTML documents

Hiroshi Sakamoto  Yoshitsugu Murakami  Hiroki Arimura  Setsuo Arikawa

Department of Informatics, Kyushu University
Hakozaki 6-10-1, Hizashi-ku, Fukuoka-shi 812-8581, Japan
{hiroshi, arim, arikawa}@i.kyushu-u.ac.jp
phone: +82-92-642-2693, fax: +82-92-642-2698

Abstract

The new wrapper model for extracting text data from HTML documents is introduced. In this model, an HTML file is considered as an ordered labeled tree. The learning algorithm takes the sequence of pairs of an HTML tree and a set of nodes. The nodes indicate the labels to extract from the HTML tree. The goal of the learning algorithm is to output the wrapper which exactly extracts the labels from the HTML trees.

keywords: information extraction, wrapper induction, semi-structured data, inductive learning

Introduction

The HTML documents currently distributed on the Internet can be regarded as a very large text database and the information extraction from the Web is widely studied. The problem of extracting the texts and attributes from HTML documents is difficult because we cannot construct the XML like database by only the limited number of HTML tags.

For this purpose Kushmerick introduced the framework of the wrapper induction (Kushmerick 2000). An HTML document is called a page and the contents of the page is called the label. The goal of the learning algorithm is, given the sequence of examples (Pn, Ln) of pages and labels, to output the program W such that Ln = W(Pn) for all n. Other extracting models, for example, are in (Hammer, Garcia-Molina, Cho, and Crespo 1997; Chun-Nan Hsu 1998; Muslea, Minton, Craig, and Knoblock 1998; Freitag 1998).

The program W is called Wrapper. Kushmerick defined several classes of Wrappers, in particular, we explain the LR-Wrapper (Kushmerick 2000). An LR-Wrapper is a sequence ((ε1, r1), ..., (εK, rK)), where the εi is called the left delimiter and the ri is called the right delimiter for the i-th attribute. The attribute is the unit of extraction data and we assume that HTML page is constructed by the finite repetitions of K attributes.

First, the LR-Wrapper finds the first appearance i of the εi in the page P and finds the first appearance j of the ri starting from the i. If such i and j are found, it extracts the string between the i and j as the first attribute in P and it continues to extract the next attribute.

The idea of the learning algorithm for LR-Wrapper is to find the εi as the longest common suffix of the strings just before the i-th attribute and the ri as the longest common prefix of the strings immediately after the i-th attribute. Thus, the string is so safe as to be long. However, in the following case, we can not get a sufficiently long delimiters.

Consider the case of extracting the attributes "arim@arim.com" and "saka@saka.co.jp". The learned left and right delimiters for the attributes are the "<" and ">", respectively. Now, let us extract the string "arim@arim.com". The first appearance of the the "<" is in the first line and the first appearance of ">"> from this point is in the third line. Thus the extracted string is the "www.arim.com".

The cause in this case is the HTML attribute values of the <a> tags for the email addresses. Since there is no common suffix of the strings, the LR-Wrapper can not determine the correct delimiters.

Thus, for overcome this difficulty, we propose the new data model for the HTML wrapper called Tree-Wrapper over the tree structures and present the learning algorithm of the Tree-Wrappers. Moreover, we experiment the prototype of our learning algorithm for more than 1,000 pages of HTML documents.

The introduced Tree-Wrapper W is the sequence (EP1, ..., EPK). The EPi, called the extraction path,
is the expansion of the notion of path to extract the i-th attributes from the HTML trees. Each EP, is of the form \langle EN_{L_i}, \ldots, EN_{R_i} \rangle, where EN_{L_i} is called the extraction node label. The most simple EN_{L_i} is the one that consists of only the node name. For a given HTML tree, the Tree-Wrapper tries to find a path matching with the path \langle EN_{L_i}, \ldots, EN_{R_i} \rangle and if it is found, then the Wrapper extracts the i-th attributes of the last node which matches with the EN_{L_i}.

Let us explain the Tree-Wrapper for the example of the HTML document in the above. In this case, the most simple Tree-Wrapper is W = \langle EP_i \rangle and EP_i = (\langle h3 \rangle, \langle a \rangle). This path matches with only the paths for the email addresses. In the next sections, we introduce the data model for such extraction and present the algorithm to learn more powerful Tree-Wrappers. In the final section, we show the expressiveness of the Tree-Wrapper by the experimental results.

The Data Model

In this section, we define the HTML tree which is constructed from an HTML file. First, we begin with the notations used in this paper. An alphabet \Sigma is a set of finite symbols. A finite sequence (a_1, \ldots, a_n) of elements in \Sigma is called string and it is denoted by w = a_1 \cdots a_n for short. The empty string of length zero is \epsilon. The set of all strings is denoted by \Sigma^* and let \Sigma^+ = \Sigma^* \setminus \{\epsilon\}. For string w, if w = a_0 a_1 \cdots a_n then the string \alpha (\beta) is called a prefix (suffix) of w, respectively.

For each tree T, the set of all nodes of T is a subset of N = \{0, \ldots, n\} of natural numbers, where the 0 is the root. A node is called a leaf if it has no child and another node is called an internal node. If n, m \in N has the same parent, then n and m are brother and n is a big brother of m if n \leq m. The sequence (n_1, \ldots, n_k) of nodes of T is called the path if n_1 is the root and n_i is the parent of n_{i+1} for all i = 1, \ldots, k - 1.

For each node n, the node label of n is the triple NL(n) = \langle N(n), V(n), HAS(n) \rangle such that N(n) and V(n) are strings called the node name and node value, respectively, and HAS(n) = \{HA_1, \ldots, HA_{n_k}\} is called the set of the HTML attributes of n, where each HA_i is of the form (ai, vi) and ai, vi are strings called HTML attribute name, HTML attribute value, respectively.

If N(n) \in \Sigma^+ and V(n) = \epsilon, then n is called the element node and the string N(n) is called the tag. If N(n) = TEXT for the reserved string TEXT and V(n) \in \Sigma^+, then n is called the text node and the value N(n) called the text value. We assume that every node n \in N is categorized to the element node or text node.

An HTML file is called a page. A page P is corresponding to an ordered labeled tree. For the simplicity, we assume that the P contains no comment part, that is, any string beginning the <! and ending the > is removed from the P.

Definition 1 For a page P, the P_i is the ordered labeled tree defined recursively as follows.

1. If P contains an empty tag \langle tag \rangle, P_i has the element node n such that it is a leaf P and N(n) = \epsilon.
2. If P contains a string t_1 \cdot w \cdot t_2 such that t_1 and t_2 are tags and the w contains no tag, then P_i has the text node n such that it is a leaf P and V(n) = w.
3. If P contains a string of the form

   \langle tag a_1 = v_1, \ldots, a_i = v_i \rangle\langle/\text{tag}\rangle

   then the tree (n_1, \ldots, n_k) is the sub tree of P on n, where

   N(n) = tag, HAS(n) = \{(a_i, v_i) | i = 1, \ldots, \ell\},

   and n_1, \ldots, n_k are the trees t_1, \ldots, t_\ell obtained recursively from the w by the 1, 2 and 3.

Next we define the functions to get the node names, node values, and HTML attributes from given nodes and HTML trees defined above. These functions are useful to explain the algorithms in the next section. These functions return the values indicated below and return null if such values do not exist.

- Parent(n): The parent of the node n \in N.
- ChildNodes(n): The sequence of all children of n.
- Name(n): The node name N(n) of n.
- Value(n): The concatenation V(n_1) \cdots V(n_k) of all values of the leaves n_1, \ldots, n_k of the subtree on n in the left-to-right order.
- Pos(n): The number of big brothers n_i of n such that N(n_i) = N(n).

The following functions are to get HTML attributes.

- HTMLAttSet(n): The HTML attribute set HAS(n) of the node n \in N.
- HTMLAttName(n,i): The HTML attribute name ai of (ai, vi) in HAS(n).
- HTMLAttValue(n,i): The HTML attribute value vi of (ai, vi) in HAS(n).

Finally, we define the notion of common HTML attribute set of HTML attribute sets and define the function which gets the common HTML attribute set.

Definition 2 Let S = \{HAS(n_1) | i = 1, \ldots, k\} and HAS(n_i) = \{(ai, vi) \} (i = 1, \ldots, k) the common HTML attribute set of S, denoted by CHAS(S), is the set (\bigcap S' \in S' \subseteq (\bigcup S') \cap S'), where S' is the set of (a, \epsilon) such that each HAS(n_i) contains an HTML attribute (a, v) for the a and (a, v) \in HAS(n_i), (a, v) \in HAS(n_j) and n_i \neq n_j, where the \epsilon is a special symbol not belonging to \Sigma.

- CommonAttSet(HAS(n_1), \ldots, HAS(n_k)): The common HTML attribute set of all HTML attribute set HAS(n_i) for i = 1, \ldots, k.

What the HTML Wrapper of this paper extracts is the text values of text nodes. These text nodes are called text attributes. A sequence of text attributes is called tuple. We assume that the contents of a page P is a set of tuple t_i = (\langle a_1, v_1 \rangle, \ldots, \langle a_k, v_k \rangle), where the K is a constant for all pages P. It means that all text attributes
in any page is categorized into at most $K$ types. Let us consider an example of HTML document of address list. This list contains three types of attributes, name, address, and phone number. Thus, a tuple can be of the form $(name, address, phone)$. However, this tuple cannot handle the case that some elements contain more than two values such as some one has two phone numbers. Thus, we expand the notion of tuple to a sequence of a set of text attributes, that is $t = (t_1, \ldots, t_K)$ and $t_i \subseteq \mathcal{N}$ for all $1 \leq i \leq K$. The set of tuples of a page $P$ is called the label of $P$.

Figure 1: The tree of the text attributes, name, address, and phone.

The Fig.1 denotes the tree containing the text attributes name, address, and phone. The first tuple is $t_1 = \{(3), (4), \{6, 7\}\}$ and the second tuple is $t_2 = \{(8), \{9\}\}$. The third attribute of $t_1$ contains two values and the second attribute of $t_2$ contains no values.

The Learning Algorithm for the Tree-Wrappers

In this section, we give the two algorithm. The first algorithm execT($P_i, W$) extracts the text value of the text attributes from the page $P_i$ using given the Tree-Wrapper $W$. The second algorithm learnT($E$) finds the Tree-Wrapper $W$ for the sequence $E = (P_1, \ldots, P_N)$, where each $P_i$ is an extraction label.

The Tree-Wrapper

Definition 3 The extraction node label is a triple $ENL = (N, Pos, HAS)$, where $N$ is a node name, $Pos \in \mathcal{N} \cup \{*\}$, $HAS$ is an HTML attribute set. The extraction path is a sequence $EP = (ENL_1, \ldots, ENL_I)$. An $ENL = (N, Pos, HAS)$ of a node $n$ is considered as the generalization of which contains the node name, node value, and the value of the function $Pos$. The first task of execT is to find a path in $P_i$ which matches with the given $EP$ and to extract the text value of the last node of the path. The following function and definition gives the semantics of the matching.

```
boolean isMatchENL(n, ENL)
/*input: node n, ENL = (N, Pos, HAS)*/
/*output: true or false*/
if(N==name(n) && (Pos==Pos(n) || Pos==*) &&
isMatchHAS(n,HAS)) == true;
else return false;
```

```
boolean isMatchHAS(n, HAS)
/*input: node n, HAS=(HA1,...,HAm)/
/*output: true or false*/
for( m=1; m <= M; m++)
  if(HTMLAttValue(n,m) == vm && vm != *)
    return false;
return true;
```

Definition 4 Let $ENL$ be an extraction node label and $n$ be a node of a page $P_i$. The $ENL$ matches with the $n$ if the function isMatchENL($n$, $ENL$) returns true. Moreover, let $EP = (ENL_1, \ldots, ENL_I)$ be an extraction path and $p = (n_1, n_2)$ be a path of a page $P_i$. The $EP$ matches with the $p$ if the $ENL_i$ matches with $n_i$ for all $i = 1, \ldots, I$.

Definition 5 The Tree-Wrapper is the sequence $W = (EP_1, \ldots, EP_K)$ of extraction paths $EP_i = (ENL_1, \ldots, ENL_I)$, where each $ENL_j$ is an extraction label.

The algorithm execT is given as follows.

```
Algorithm execT($P_i, W$)
/* input: W = (EP_1, \ldots, EP_K) and $P_i$ */
/* output: The label $L_i = \{t_1, \ldots, t_m\}$ */
1. For each $EP_i$ ($i = 1, \ldots, K$), find all path $p = (n_1, \ldots, n_k)$ of $P_i$ such that $EP_i$ matches with $p$ and add the pair $(i, n_k)$ into the set $Att$. */ The $n_k$ is a candidate for the $i$-th text attribute.*/
2. Sort all elements $(i, n_k) \in Att$ in the increasing order of $i$. Let $LIST$ be the list and $j = 1$.
3. If the length of $LIST$ is 0 or $j > m$, then halt. If not, find the longest prefix list of $LIST$ such that all element is in non-decreasing order of $i$ of $(i, n)$ and for all $i = 1, \ldots, K$, compute the set $t_i = \{n | (i, n) \in list\}$. If the list is empty, then let $t_i = \emptyset$.
4. Let $t_j = (t_{a1}, \ldots, t_{aK})$, $j = j + 1$, remove the list from $LIST$ and go to 3.
```

The learning algorithm

Given a pair $(P_i, L_n)$, the learning algorithm learnT calls the function learnExPath which finds the extraction path $EP^n_i$ for the $i$-th text attributes and $i = 1, \ldots, K$ and it computes the composite $EP_i \cdot EP^n_i$, where $EP_i$ is the extraction path for the $i$-th text attribute found so far. The definition of the composite $EP_i \cdot EP^n_i$ is given as follows and Fig.2 is an example for a composite of two extraction path.

Definition 4 Let $ENL$ be an extraction node label and $n$ be a node of a page $P_i$. The $ENL$ matches with the $n$ if the function isMatchENL($n$, $ENL$) returns true. Moreover, let $EP = (ENL_1, \ldots, ENL_I)$ be an extraction path and $p = (n_1, n_2)$ be a path of a page $P_i$. The $EP$ matches with the $p$ if the $ENL_i$ matches with $n_i$ for all $i = 1, \ldots, I$.

```
Algorithm learnT($E$)
/* input: $E = (EP_1, \ldots, EP_K)$ and $P_i$ */
/* output: The label $L_i = \{t_1, \ldots, t_m\}$ */
1. For each $EP_i$ ($i = 1, \ldots, K$), find all path $p = (n_1, \ldots, n_k)$ of $P_i$ such that $EP_i$ matches with $p$ and add the pair $(i, n_k)$ into the set $Att$. */ The $n_k$ is a candidate for the $i$-th text attribute. */
2. Sort all elements $(i, n_k) \in Att$ in the increasing order of $i$. Let $LIST$ be the list and $j = 1$.
3. If the length of $LIST$ is 0 or $j > m$, then halt. If not, find the longest prefix list of $LIST$ such that all element is in non-decreasing order of $i$ of $(i, n)$ and for all $i = 1, \ldots, K$, compute the set $t_i = \{n | (i, n) \in list\}$. If the list is empty, then let $t_i = \emptyset$.
4. Let $t_j = (t_{a1}, \ldots, t_{aK})$, $j = j + 1$, remove the list from $LIST$ and go to 3.
Definition 6 Let \( ENL_1 \) and \( ENL_2 \) be any extraction node labels. The composite \( ENL_1 \bar{\cdot} ENL_2 \) is the extraction node label \( ENL = (N, Pos, HAS) \) such that

1. \( N = N_1 \) if \( N_1 \rightarrow N_2 \) and \( ENL \) is undefined otherwise,
2. \( Pos = Pos_1 \) if \( Pos_1 = Pos_2 \), and \( Pos = * \) otherwise,
3. \( HAS = CommonAttSet(HAS_1, HAS_2) \).

Definition 7 Let \( EP_1 = (ENL_1, \ldots, ENL_l) \) and \( EP_2 = (ENL_1, \ldots, ENL_l) \) be extraction paths. The \( EP = EP_1 \bar{\cdot} EP_2 \) is the longest sequence \((ENL_1, \ldots, ENL_l)\) such that all \( ENL_i \) are defined for \( i = 1, \ldots, l \), where \( \leq \min(n, m) \).

Figure 2: The composite of extraction paths.

The learnExPath calls the function getPath(n) which finds the path p from the root to the node n and for each node \( n_i \) of \( p = (n_1, \ldots, n_l) \) \( (n_1 = n) \), it computes the \( ENL_i \) and returns the \( EP = (EP_1, \ldots, EP_l) \). Finally, the complete description of the learning algorithm learnT is given as follows.

**Experimental Results**

We equip the learning algorithm by Java language and experiment with this prototype for HTML documents. For parsing HTML documents, we use the OpenXML 1.2 (http://www.openxml.org) which is a validating XML parser written in Java. It also parse HTML and supports the HTML parts of the DOM (http://www.w3.org/DOM).

The experimental data of HTML pages is collected by the citeseers which is a scientific literature digital library (http://citeseers.nj.nec.com). The data consists of 1,300 HTML pages. We choose \( art_1 = \"the title\", art_2 = \"the name of authors\", and art_3 = \"the abstract\"\) as the text attributes and practice the learnT.

All pages are indexed to be \( P_1, \ldots, P_{1300} \) in the order of the file size. The training example is \( E = \{(P_i, L_i) \mid i = 1, \ldots, 10\} \), where the \( L_i \) is the label made from the \( P_i \) in advance. The result is shown in Fig. 3 which is the Tree-Wrapper \( W \) found by learnT(E).
The future work of this study is to expand the Tree-Wrapper so that it can extract the HTML attributes rather than the text attributes. We also expand the prototype to extract substrings of the text values for overcome the above difficulty.

References


Crave
c


