Automatically Generating Labeled Examples for Web Wrapper Maintenance

Juan Raposo, Alberto Pan*, Manuel Álvarez
University of A Coruña
{jrs,apan,mad}@udc.es

Abstract

In order to let software programs gain full benefit from semi-structured web sources, wrapper programs must be built to provide a “machine-readable” view over them. A significant problem of this approach is that, since web sources are autonomous, they may experience changes that invalidate the current wrapper. In this paper, we address this problem by introducing novel heuristics and algorithms for automatically maintaining wrappers. In our approach the system collects some query results during normal wrapper operation and, when the source changes, it uses them as input to generate a set of labeled examples for the source which can then be used to induce a new wrapper. Our experiments show that the proposed techniques show high accuracy for a wide range of real-world web data extraction problems.

1. Introduction

A program able to provide software applications with a structured view of a semi-structured web source is usually called a wrapper. Wrappers accept a query against the web source and return a set of structured results, thus enabling applications to access web data in a similar manner to that of information from databases. Web wrapper generation has been an active research field for some years. See [4] for a survey.

The main problem with wrappers is that they can become invalid when the web sources change their underlying templates. In this paper, we present novel heuristics and algorithms for dealing with this issue.

The rest of the paper is organized as follows. Section 2 provides the context for the rest of the paper by presenting the basic stages in which the problem of wrapper maintenance can be divided in our approach. Section 3 and 4 discuss our methods for implementing these stages. Section 5 summarizes our experiments in using the system for several real-world web data extraction problems. Finally, section 6 discusses related work.

2. The stages of wrapper maintenance in our approach

The approach we use for wrapper generation and maintenance is based on the wrapper induction techniques ([4]). In them, data extraction rules for wrappers are generated from a training set of user-labeled examples.

The basis of our approach for automatic maintenance is making wrappers collect some results from valid queries during their operation, and when the source changes, to use those results to generate a new training set of labeled examples to bootstrap the wrapper induction process again. Thus, the process is comprised of four stages:

1. Collecting results from queries during wrapper operation.
2. Verifying whether the wrapper continues to be valid or not.
3. Generating a new training set. When the wrapper is determined to be no longer valid, the system must generate a new set of labeled examples from the results stored during the first stage.
4. Re-generating Wrappers. From the examples generated in the previous stage, the new extraction rules can be re-induced.

In this paper we will focus on the first and third stages. Methods for stage 2 have been addressed in [3,5]. In the stage 4, we use our own induction algorithm [8] although others could also be used.

3. Collecting results from queries

The objective of this stage is to collect the results from a subset of the queries executed by the wrapper during its normal operation so, when the source change, they can be used to generate a new training set of labeled examples. The system works as follows:

* Alberto Pan’s work was partially supported by the “Ramón y Cajal” programme of the Spanish Ministry of Science and Technology.
1. When the wrapper answers a query, it stores up to a certain (customizable) number of results from the query in a local database. For each result collected, the system stores the query that generated it, the extraction rule (or set of rules) used by the current wrapper to parse it and an expiration date computed from the current time.

2. Periodically, the current content in the database is explored for checking whether the non-expired results are sufficient to satisfy certain requirements we will describe below or not. If so, all the expired results are removed. If not, then the system computes a minimum set of expired queries that must remain stored.

The requirements we want to assure that the stored results satisfy are:

1) Storing at least a certain (customizable) number of results. In general terms, the stages 3 and 4 will provide better output results when using a sufficient number of results (but not big enough to cause too much delay). We usually store at least 100 results.

2) Storing results from at least a certain (customizable) number of different queries. Queries may implicitly determine some properties of the response pages. For instance, an electronic shop may assign a given discount (e.g. 15%) for all the books by a given author. If we happened to use only results from the query ‘author=a’, the wrapper induction process could wrongly consider the string “15%” to be an invariant for all the books. We usually store results from at least 10 different queries.

3) Storing at least a certain (customizable) number of results for each current visual representation of a result in the source. Most web sources use slightly different visual representations for the results of a query. This means that a wrapper will need a different set of extraction rules for extracting the results using each visual representation. In order to generate the extraction rules for all the visual representations, the wrapper induction process will benefit of having examples from all of them. We have heuristically found very useful to assure having stored at least several results using each former visual representation (before the source change) in order to increase the probability that the stored results also cover all the new visual representations (after the source change).

4. Generating a new training set

In this section we describe our approach for obtaining new labeled examples from the stored results.

First, let us introduce the data model underlying our system. We model the data elements to extract from a web source as a set whose members are either atomic label/value pairs or other elements. Atomic values are character strings. For instance we can model the schema of data elements of type ALBUM as the set \{TITLE, AUTHOR, DATE, EDITION: \{FORMAT, PRICE\}\}. This way, a data element of a given type can be seen as a tree whose leaves are atomic values and the non-leaf nodes represent non-atomic elements.

In the labeling stage, we have the following inputs:

1. A source modeled according to a type $T = \{field_1, \ldots, field_r\}$.
2. The set of queries $Q = \{q_1, \ldots, q_n\}$ whose results were collected during wrapper operation.
3. For each $q_i \in Q$, we will use as input its stored results. We will consider them as a set of unlabeled examples $E_i = \{e_{i1}, \ldots, e_{im}\}$.
4. For each $q_i \in Q$, the page or set of pages $P_i$ that form the current response to $q_i$ in the source.

The objective of this stage, to be fully explained later, is to label the maximum possible number of examples in the new pages. Labeling an example $e_q$ consists in locating in $P_i$ an occurrence of the example by correctly identifying the values of all its fields.

For instance, let us consider the case of an electronic bookshop whose results are of type $T = \{title, author, format, price\}$. Let us also suppose an example $e_q = \{title, "Beginning XML, Second Edition", (author, "David Hunter et al"), (format, "Paperback"), (price, "27.19")\}$ obtained through a query $q_x$.

In the Figure 1, an Amazon page containing $e_q$ is shown. As can be seen, the values of the data fields of $e_q$ appear in several positions on the page, making it difficult...
to identify the real occurrence of the example. For instance:

- The value of the ‘Format’ field (‘Paperback’) is repeated several times in the page, since many other books have the same value (the same may also apply to the ‘Price’ and ‘Author’ fields).

- Amazon shows information about the book represented by the example in two different places in the page (the list of most popular results and the complete list of results). Nevertheless, we should use the occurrence in the complete list, since we will need to generate extraction rules to extract all the results and not only the most popular ones.

Another complication arises because the stored example might actually not appear in the page even though the page contains individual occurrences of all of its data values. Since there will be some time interval between the moment when the results were collected and the moment when we try to regenerate the wrapper, the data on the source may have changed during the interval. For instance, Figure 2 shows an example where the price of the book has changed but, by chance, another book has the same price. The system needs to detect these situations and discard these “fake” occurrences.

Note that the main assumption of our approach is that we can usually be confident that a good number of the unlabeled examples contained in E are still present in P. In the cases where the values of the data elements to be extracted from the source vary very rapidly (e.g. stock quotes information), this assumption may be partly wrong. The extended version of this paper [8] shows how our approach can deal with those sources.

4.1. Labelling examples

Once we have stated the labeling problem, we proceed to describe our approach to address it. We will focus in the particular case where the data elements to extract are modeled according to a type composed exclusively of atomic fields. These techniques can be easily generalized to types of arbitrary depth (see [8]).

More formally, we can define the problem of correctly labeling an example in this case as follows. Let us assume we have an example e of type \( T = \langle \text{field}_1, \ldots, \text{field}_r \rangle \) which we want to label in the page P. Let us also assume that all the fields of T are atomic. Then examples of type T are of the form \( e = \langle \text{value}_{1}, \ldots, \text{value}_{r} \rangle \) where \( \text{value}_{k} \) is a string denoting the value for the \( k \)-th field of \( T \) in \( e \). Each string \( \text{value}_{k} \) may appear multiple times in \( P \) and we will term each appearance of \( \text{value}_{k} \) as a candidate field occurrence of \( e \) for \( \text{field}_{k} \).

Now, a candidate example occurrence for \( e \) in \( P \) will be of the form \( \text{coe} = \langle \text{value}_{e, pos_{1}}, \ldots, \text{value}_{e, pos_{r}} \rangle \) where \( \text{value}_{e, pos_{k}} \) denotes the \( pos_{k} \)-th occurrence in \( P \) of the string \( \text{value}_{k} \) that will be a correct labeling for \( e \) if the identified occurrences for all the \( \text{value}_{k} \) form a real apparition in \( P \) of the data element represented by \( e \).

As we have already said, we have as inputs a source modeled according to a type \( T \) along with the set of queries \( Q \) whose results were collected during wrapper operation. For each \( q \in Q \), we will have a set of unlabeled examples \( E = \{ e_{1}, \ldots, e_{n} \} \) and the page or set of pages \( P \) that constitute the current answer to \( q \) in the source.

An important remark is that if \( P \) consists of several pages (e.g. when the examples are contained in “detail pages” accessed from a search results listing, or when the result listing is paginated), the algorithms which follow will consider all the input pages as if they were part of a single “aggregated” page. The DOM tree of this “aggregated page” would be built by making the DOM trees of the original pages all be children of a new common root node (i.e. the DOM trees of the original pages will be considered as siblings).

The algorithm to label an example \( e \in E \in P \) is shown in Figure 3. It begins by computing all the candidate example occurrences for \( e \). If their number is higher than a certain limit, then the most promising ones are chosen (see [8]). Then the algorithm ranks the candidate example occurrences according to two heuristics: the “sibling nodes” heuristic and the proximity heuristic.

The sibling nodes heuristic (detailed in the section 4.2.1) ranks a candidate example occurrence according to whether it is an instance of a repetitive sequence containing several other examples with similar layout or not. The proximity heuristic (detailed in the section 4.2.2) ranks candidate example occurrences according to the proximity in the page of the data values that form them.

The algorithm uses the sibling nodes heuristic as the main criteria for choosing the best candidate occurrence. The proximity heuristic is used only when the previous heuristic returns the same value for several candidates. This is because, as we will see, the sibling nodes heuristic has into account the context provided from all the other examples making it very effective.
Algorithm: Obtain best atomic candidate occurrences for an example.
  - \( R_e = \text{obtainBestAtomicCandidateOccurrences}(P, T, e, \text{maxCandidates}) \)

Input:
- \( P \), a document or a set of documents considered as if they were part of a single page where their DOM trees were siblings.
- A type \( T=\{\text{field}_1, \ldots, \text{field}_r\} \), whose fields are all atomic.
- \( E = \{e_1, \ldots, e_m\} \), where \( e_j \) is an example data element contained in \( P \) for data element \( D \).
- \( e_{j+1, k} \) is of the form \( (\text{field}_j, \text{value}_j), \ldots, (\text{field}_r, \text{value}_r) \) where \( \text{value}_k \) denotes the value for the \( k \)th field of \( T \) in \( e_j \).
- \( e \) is an example contained in \( E \).

Output:
- \( R_e \), a set with the best candidate example occurrences for \( e \).

1. Compute the number of occurrences of each \( \text{value}_{e_{k+1}} \) in \( P \). Let \( O_k \) be the set of occurrences of \( \text{value}_{e_{k}} \) in \( P \).
2. Compute the number of all candidate example occurrences of \( e \) in \( P \) as \( \text{numCandidateExampleOccurrences}(P, T, E, e) \).
3. A candidate example occurrence for \( e \) is of the form \( e_{o_i} = (\text{value}_{o_i}, \text{pos}_{o_i}), (\text{value}_{o_2}, \text{pos}_{o_2}), \ldots, (\text{value}_{o_r}, \text{pos}_{o_r}) \) where \( \text{value}_{o_k} \) denotes the \( \text{pos}_{o_k} \)-th occurrence in the page of the string \( e_{o_k} \) and, thus, implicitly identifies a node in the DOM tree of \( P \).
4. If \( \text{numCandidateExampleOccurrences}(P, T, E, e) > \text{maxCandidates} \):
   a. Choose the best candidate example occurrences guaranteeing \( \text{numCandidateExampleOccurrences}(P, T, E, e, \text{maxCandidates}) \).
5. Sort the candidate example occurrences by using the “sibling nodes heuristic”.
6. If the “sibling nodes heuristic” is applicable (it returns a value greater than zero for some candidate occurrence of some example): and all the candidate example occurrences have value zero for this heuristic, then discard the example.
7. If the best candidate occurrences of any two examples overlap in \( P \), discard the example which ranks lower in the proximity heuristic.
8. Sort the occurrences with the same value for the sibling nodes heuristic by using the “proximity heuristic”.
9. Set \( R_e \), a set with all the occurrences with the best punctuation.

Figure 3. Base algorithm

The algorithm also needs to detect “fake” occurrences of the kind shown in Figure 2 (steps 6 and 7 of the algorithm). The techniques employed for this task are detailed in section 4.1.3.

4.1.1. Sibling nodes heuristic

This heuristic builds on the observation that the data elements to extract are usually instances of a repetitive sequence whose elements share a similar layout and are placed in “symmetrical” positions inside the same page or in different pages.

Therefore, if a given candidate example occurrence \( e_{o_i} \) is a real occurrence of an example \( e_j \), then there should be occurrences of other examples \( e_{k} \in E \) in the “sibling subtrees” of the sub-tree containing \( e_{o_i} \) in the DOM tree of \( P \) (see Figure 4).

The sibling nodes heuristic take as input the DOM tree of a page \( P \) and a candidate occurrence \( e_{o_i} \) for an example \( e_{o_i} = (\text{field}_j, \text{value}_{o_i}), (\text{field}_2, \text{value}_{o_j}), \ldots, (\text{field}_r, \text{value}_{o_r}) \). \( e_{o_j} \) is of the form \( e_{o_j} = (\text{value}, \text{pos}), (\text{value}_2, \text{pos}_2), \ldots, (\text{value}_r, \text{pos}_r) \), where \( (\text{value}_{o_k}, \text{pos}_{o_k}) \) denotes the \( \text{pos}_{o_k} \)-th occurrence in the page of the string \( e_{o_k} \) and, thus, implicitly identifies the node in the DOM tree of \( P \) where such occurrence begins.

Now, given a page \( P \), a set of examples \( \{e_1, \ldots, e_n\} \) for the same type \( T=\{\text{field}_1, \ldots, \text{field}_r\} \) and a set of candidate example occurrences \( \{e_{o_1}, \ldots, e_{o_n}\} \) for a given \( e_j \in \{1, \ldots, n\} \) the heuristic rank the example occurrences for \( e_j \) using the following process:
1. For each \( e_{o_i} \in \{e_{o_1}, \ldots, e_{o_n}\} \):
   a. \( \text{numSiblingOccurrences}(e_{o_i}) = 0 \)
   b. Obtain \( A_i \), the deepest common ancestor of all the nodes identified by \( (\text{value}_1, \text{pos}_1), (\text{value}_2, \text{pos}_2), \ldots, (\text{value}_r, \text{pos}_r) \).
   c. Obtain \( \{S_1, \ldots, S_l\} \) the set of sibling nodes of \( A_i \) in the DOM tree of \( P \) (including \( A_i \)).
   d. For each \( S_j \in \{1, \ldots, l\} \):
      i. For each \( e_{o_i} \in \{e_{o_1}, \ldots, e_{o_n}\} \), search for candidate occurrences of \( e_{o_i} \) in the sub-tree with root \( S_j \).
      ii. If \( e_{o_i} \) is found for exactly one \( e_{o_k} \), then
         1. If any of the candidate occurrences found has “similar layout” to that of \( e_{o_k} \), then \( \text{numSiblingOccurrences}(e_{o_i}) = +1 \).
2. Sort candidate example occurrences in \( \{e_{o_1}, \ldots, e_{o_n}\} \) by \( \text{numSiblingOccurrences} \), in descending order.

A crucial consideration about the algorithm is the method used in step 1.d.ii for searching candidate occurrences of other examples having “similar layout” to \( e_{o_i} \). The basic idea is to retain only those occurrences having their values positioned in the tree in a similar way as \( e_{o_i} \). So, we compute the XPath route for every field occurrence of each candidate example occurrence, and we compare them with the route to the corresponding field occurrence in \( e_{o_i} \), checking whether they are equal (except for certain tags considered irrelevant for these purposes, such as \( <B> \), \( <I> \), \( <FONT> \), etc.). In addition, we also check that the ordering of the field occurrences is the same as in \( e_{o_i} \).

It should also be noted that the step 1.d.ii of the algorithm increments \( \text{numSiblingOccurrences} \), if and only if the found candidate example occurrences from a node belong to exactly one alternative example. If candidate occurrences for more than one example, then it probably means that we are searching from a node which is too “high” in the DOM tree. This usually identifies a situation like the one shown in Figure 2, where our candidate example occurrence is actually composed of
field occurrences from several different data elements. Thus, in these cases, the heuristic ranks the candidate occurrence with value 0.

Finally, we need to consider the applicability of the sibling nodes heuristic. In some rare cases the DOM tree of a page is constructed in such a way that any common ancestor node to all the data fields of an example is, at the same time, ancestor of the data fields of the other examples (see Figure 5). In these situations, the “deepest common ancestor” will always be considered as too high by the sibling nodes heuristic and it will always return zero for every occurrence of every example even if some occurrences are actually valid. So, in these cases the sibling nodes heuristic is not applicable and the proximity heuristic will be used.

4.1.2. Proximity heuristic

This heuristic builds on the observation that all the data values belonging to the same example tend to be grouped near each other in the page. We can measure the “sparseness” of a candidate example occurrence by computing the sum of the distances between every two consecutive values belonging to that occurrence. The distance between two values is computed in the DOM tree of P as the number of nodes between the last node of the first value and the first node of the next value (depth-first traversal order). Again certain tags such as <B>, <I> and <FONT> are considered irrelevant.

Another possibility for implementing this heuristic is actually computing the “visual distance” in the page rendered by the browser between the values of the candidate example occurrence. The visual distance between two DOM nodes may be computed by using the functionalities provided by current browser’s APIs. In our tests, the DOM distance measure performs almost as good as the visual distance “measure”. Thus, for convenience and efficiency reasons, our current implementation uses the DOM distance measure.

4.1.3. Detecting “fake” examples

As it has already been said, it may occur that a chosen example is not really present in the page although there are candidate field occurrences for all of their values (see Figure 2). This is dealt with in the steps 6 and 7 of the algorithm. Every candidate example occurrence of these “fake” examples will rank zero in the “sibling nodes heuristic” since its “deepest common ancestor” will be “too high” in the DOM tree (recall section 4.1.1).

Nevertheless, this rule must only be used when the “sibling nodes” heuristic is applicable (recall section 4.1.1 for a description of the cases in which it is not) because, in other case, all the examples would be wrongly discarded. We consider the sibling nodes heuristic as not applicable if all the candidate occurrences rank zero in the sibling nodes heuristic for all the examples.

Another technique we apply for detecting “fake” examples is analyzing overlaps in the page among the best candidate occurrences of several examples. More precisely, if the best candidate occurrences for two examples overlap, we reject the example whose best candidate occurrence ranks lower according to the proximity heuristic. The intuition here is that the sources do not show the fields of a data element mixed with the fields of other data elements.

5. Experiments

To evaluate the effectiveness of our approach we monitored a set of representative Web sites for several months. For every site we generated a wrapper and used it on a daily basis to execute different queries. The system automatically stored some of the queries and their results according to the techniques explained in section 3. When a change in the format of the pages of a web site was detected the system generated a new set of labeled examples and induced the new wrapper. Then we tested it with a new set of pages from the source obtained through different queries.

To quantify the results of the stage of generating a new training set we defined the following metrics:

- LN: number of input examples (from the stored queries).
- LPE: number of input examples that is still present in the current pages of the source after the change.
- LTE: number of total examples labeled.
- LCE: number of examples correctly labeled.
- Labeling Recall (LR) = LCE/LTE.
- Labeling Precision (LP) = LCE/LTE.

Once the wrappers have been regenerated, we define the following metrics to measure the obtained results:
- N: number of data items which should be extracted from the test pages.
- TE: number of total data items extracted by the regenerated wrapper from the test pages.
- CE: number of correct data items extracted by the regenerated wrapper from the test pages.
- Recall (R) = CE / N.
- Precision (P) = CE / TE.

### Table 1. List of sources and data fields.

<table>
<thead>
<tr>
<th>Source</th>
<th>Data fields</th>
<th>Sibling Nodes</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllBooks4Less (Book)</td>
<td>title, author, format, price, quantity</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Amazon (Book)</td>
<td>title, author, format, price, availability</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Amazon (Magazine)</td>
<td>title, price, unit price, issues, months</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Barnes&amp;Noble (Book)</td>
<td>title, author, format, price, availability</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>BigBook (Person)</td>
<td>name, street address, city, state, phone</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Cordis (Project)</td>
<td>title, date, relevance, status, program acronym, project acronym, reference, start date, end date, abstract</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Delphion (Patent)</td>
<td>title, publication, date, score</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Discoweb (Album)</td>
<td>title, artist, date, edition: {format, price}</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Espacenet (Patent)</td>
<td>title, publication, date, inventor, applicant, IPC</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Nasdaq (Flash Quote)</td>
<td>symbol, last sale, share volume, market</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>NewYorkTimes (News Story)</td>
<td>title, author, summary</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>YahooPeople (Person)</td>
<td>name, street address, city, state, phone</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>YahooQuotes (Quote)</td>
<td>symbol, time, trade, change, % change, volume</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

### 5.1. Analysis of results

Table 1 lists the subset of the monitored sources where any changes occurred during the period of study. In each of these sources it took place just one format change except for Barnes & Noble, Amazon Magazine, Espacenet and Yahoo People where two changes occurred (in these cases we computed the averages of the obtained results). The last column indicates if the sibling nodes heuristic was applicable after the change.

Some of the changes were relatively small. For instance, in Yahoo People a new column was added to the data that renders the people information. On the other hand, some sources completely changed their underlying templates (for instance, Barnes & Noble).

Table 2 shows the values of the metrics computed for the labeling process. A first conclusion is that the ratio of preserved examples is high for almost all the monitored sources. This supports the idea of using results from previous queries as a good basis onto which generate a new set of examples for re-induction. The LR and LP metrics give a 100% value for all the sources except for AllBooks4Less where the system failed to label an example that really was in the page and, in addition, it failed to discard two “fake” examples; and for New York Times where the system wrongly labeled one example.

The errors in AllBooks4Less were caused because the sibling nodes heuristic was not applicable and there were some rare cases where the proximity heuristic failed. In New York Times the sibling nodes heuristic is applicable but only the 19% of the stored results were present. The errors in recall and precision were caused because the system wrongly labeled one example which was present in the source.

Table 3 shows the metrics calculated after the wrappers were regenerated completely. The second and third columns show the values computed for the Recall and Precision metrics. The fourth column indicates the number of examples used to test the wrapper. The fifth column shows the number of input examples to the re-induction algorithm (i.e. the examples labeled in the previous stage). The sixth column shows the number of extraction rules created by our induction algorithm for each wrapper. As we have already seen in section 3, an extraction rule can be roughly matched to a different “visual representation” of the data items in the source.

The basic factor influencing the results in this stage is that the induction process benefits from having at least a
certain number of examples for each possible visual representation. So, the recall of the regenerated wrapper depends on the suitability of the set of labeled examples to cover the whole range of those different visual representations. Consequently with this, the worst results were obtained in Amazon Book, which is, by far, the source with more variety of visual representations, and in New York Times, which is the source with more data variability (only 19% of the stored examples are preserved).

Table 3. Wrapper regeneration metrics.

<table>
<thead>
<tr>
<th>Source</th>
<th>R (%)</th>
<th>P (%)</th>
<th>N</th>
<th>LTE</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllBooks4Less (Book)</td>
<td>99.7</td>
<td>97.9</td>
<td>100</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>Amazon (Book)</td>
<td>86.8</td>
<td>93.4</td>
<td>100</td>
<td>147</td>
<td>27</td>
</tr>
<tr>
<td>Amazon (Magazine)</td>
<td>99.8</td>
<td>100</td>
<td>100</td>
<td>50</td>
<td>7</td>
</tr>
<tr>
<td>Barnes&amp;Noble (Book)</td>
<td>98.2</td>
<td>100</td>
<td>100</td>
<td>65</td>
<td>8</td>
</tr>
<tr>
<td>BigBook (Person)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>Cordis (Project)</td>
<td>97.8</td>
<td>100</td>
<td>100</td>
<td>99</td>
<td>5</td>
</tr>
<tr>
<td>Delphi (Patent)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>Discoweb (Album)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>87</td>
<td>2 / 1</td>
</tr>
<tr>
<td>Espacenet (Patent)</td>
<td>99.5</td>
<td>100</td>
<td>100</td>
<td>196</td>
<td>12</td>
</tr>
<tr>
<td>Nasdaq (Flash Quotes)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>NewYorkTimes (News Story)</td>
<td>90</td>
<td>100</td>
<td>200</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>YahooPeople (Person)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>YahooQuotes (Quote)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>2</td>
</tr>
</tbody>
</table>

(1) Album type / Edition subtype

6. Related work

Wrapper generation has been an active research field for years (see [4] for a survey). These works are only concerned with wrapper generation and do not deal with the wrapper maintenance issue. [3] addresses the wrapper verification problem and, thus, it is complimentary to our work.

The approaches most similar to ours are those presented in [5,6,7]. [5] also uses examples collected during previous wrapper operation to perform an automatic process for labeling a set of new examples. Nevertheless, they do not deal with the maintenance of wrappers for pages containing “lists” of data elements.

[6] identifies candidate field occurrences by assuming some structural features to remain invariant when sources change. These features are annotations (text strings that identify a certain data field, such as “Title” or “Price”), hyperlink features and syntactic features (it assumes that the values will conform to a given regular expression). Then, they use the candidate field occurrences and the schema of the target data elements, to identify a set of blocks in the page which conforms to the structure and content suggested by the data schema. In sources where the assumption of invariance of annotations and hyperlinks does not hold the system will fail to identify correct examples.

[7] proposes a wrapper re-induction approach which iteratively uses the extracted data at a given time \( t \) to regenerate the wrapper at a later moment \( t + s \), where \( s \) is small. Nevertheless their approach can only support very simple wrappers (usually called LR-wrappers) that are not expressive enough to deal with most modern web sites.

Another difference with respect to [5,6,7] is that the techniques proposed here have into account the context provided by other examples in order to choose the best candidate occurrences for an example. This improves the accuracy of the labeling process and makes the approach not vulnerable when the pages have several portions that conform to the underlying schema.

The works presented in [1] and [2] deal with automatic wrapper generation. They take as input a set of example pages of the same class, and automatically induce the underlying template they conform to. These approaches have the advantage of not requiring input examples for wrapper generation (at the cost of having less information on which to base the building of accurate wrappers), but they do not annotate the extracted results and thus, they do not constitute complete solutions to the maintenance problem.

7. References


