

Site-Wide Wrapper Induction for Life Science Deep Web Databases

Saqib Mir^{1,2}, Steffen Staab², Isabel Rojas¹

¹EML Research

Schloss-Wolfsbrunnenweg 33
D-69118 Heidelberg, Germany

²Institute for Computer Science,
University of Koblenz-Landau,
D-56016 Koblenz, Germany

{saqib.mir, isabel.rojas}@eml-r.villa-bosch.de, staab@uni-koblenz.de

Abstract. We present a novel approach to automatic information extraction from Deep Web Life Science databases using wrapper induction. Traditional wrapper induction techniques focus on learning wrappers based on examples from one class of Web pages, i.e. from Web pages that are all similar in structure and content. Thereby, traditional wrapper induction targets the understanding of Web pages generated from a database using the same generation template as observed in the example set. However, Life Science Web sites typically contain structurally diverse web pages from multiple classes making the problem more challenging. Furthermore, we observed that such Life Science Web sites do not just provide mere data, but they also tend to provide schema information in terms of data labels – giving further cues for solving the Web site wrapping task. Our solution to this novel challenge of Site-Wide wrapper induction consists of a sequence of steps: 1. classification of similar Web pages into classes, 2. discovery of these classes and 3. wrapper induction for each class. Our approach thus allows us to perform unsupervised information retrieval from across an entire Web site. We test our algorithm against three real-world biochemical deep Web sources and report our preliminary results, which are very promising.

Keywords: Deep Web, Wrapper Generation, Information Extraction, Database.

1 Introduction

Hundreds of freely-accessible databases are available on the Web in the Life Sciences domain, covering areas such as Genomics, Proteomics, Systems Biology and Micro Array Gene Expression, to name a few. These databases often provide complementary data, pertaining to narrow specialized sub-domains. Life Science researchers thus need to search, collect and aggregate data from multiple online resources. This Web site hopping is time consuming and error-prone, whereby a user must learn search interfaces of various Web sites, perform multiple copy-paste actions, create temporary text-files and manually link records.

“Deep Web” research aims to virtually integrate such Web-accessible databases, provide a unified query interface and, typically, aggregate query results. Deep Web data integration consists of a number of distinct sub-tasks (See [5] for a survey) such as *Source Searching and Clustering* [2, 16, 23], *Interface Extraction* [21, 34], *Interface Matching* [31, 24, 32, 14], *Interface Merging & Query Translation* [15, 19], *Wrapper Generation/Data Extraction* (See section 5 for related work)

We focus on unsupervised wrapper induction and data extraction in this paper. Automatic wrapper induction has received considerable attention in recent years. However, most techniques learn wrappers for one class of Web pages. They assume structurally and content-wise similar pages are manually provided as an input for their wrapper induction methods.

As we shall explain in section 2, in our target domain, data are spread across multiple pages of a Web site, which often differ considerably in their structure and layout (template) as well as content. We therefore need an approach to automatically group similar pages in a Web site for our wrapper induction process. Additionally, we need to automatically discover the Web-site structure, so that we may predict which wrapper to use for a Web page encountered in that Web site.

These requirements go beyond traditional wrapper induction methods. We term this compound problem of Web-page classification, site-structure discovery and wrapper induction as *Site-Wide Wrapper Induction*. Though this task is extremely hard in general, in our target domain, i.e. Web-accessible Life Science databases, we may benefit from additional cues in terms of labeling of data. Consequently, we restrict our attention to Web pages from Web sites with labeled data.

In order to solve the challenge of Site-Wide Wrapper Induction in our target domain, we provide the following original contributions in this paper:

1. A novel approach for unsupervised wrapper induction to extract labeled data
2. An original approach for Web-page classification and site structure discovery
3. An automatic mechanism for detecting and correcting errors in our wrapper learning process

The rest of this paper is organized as follows. Section 2 makes some observations about deep Web scientific sources. Based on these observations, we formulate our problem and present our approach for site-wide wrapper induction in section 3. Section 4 presents our results, followed by a review of the related work and comparison to our contributions in section 5. We conclude the paper in section 6.

2 Online Life Science Databases: Observations and Implications

We observe the following about result pages of Life Science Web sources:

I) Structured Data – The results are highly structured. This owes to the fact that the backend relational schemas are very complex, and entities in scientific domains generally have complex relations and associations.

II) Highly Dynamic Page Structures – Data fields that are NULL are often omitted from the results displayed, resulting in pages with widely varying structure. A wrapper induction method which ties its learning process to the page structure only would require numerous training pages, covering all possibilities of data arrangement.

One drawback of such an approach would be the need for a large number of input seed queries to probe the deep Web source. A related observation is that Web pages undergo frequent updates [18]. An approach which circumvents the need to learn the page structure would hence be desirable.

III) Labeled Data – Scientific data require precision and clear annotation. A natural consequence of this observation is that scientific data are labeled and, often, annotated to controlled vocabularies. We carried out a survey¹ of 20 Biochemical Web sites and found that more than 97% of the data fields on these Web sites were labeled. This differs from other domains such as E-commerce, where many data fields are often unlabeled because they have become self-explaining in the public domain (e.g. price, title). This labeling can be exploited to not only help determine data regions, but can also serve as anchors for these data regions, allowing us to disregard the portion of the page which does not contain data. We further observe that labels for the same real-world entity can be different across Web pages of the same source. Finally, labels of real-world entities, such as biological concepts, rarely change, which is beneficial for wrapper maintenance.

IV) Rich Site Structure – Data is scattered across multiple Web pages. This gives such Web sites a comprehensive structure. Therefore, the wrapper must be able to navigate through the result pages to extract data. We also observe that some data fields, together with their labels, reappear on multiple pages. This reoccurrence can be used for mutual reinforcement, to detect and correct errors in the induction process.

V) Web Services – Our final observation is that of Web service API access. While some Life Science databases do provide such APIs, our survey² of 100 online databases showed that only 11 sources provide programmatic access, and even among these, the coverage of the database in some cases is not complete. Therefore, Web pages still remain the primary form of data dissemination. Furthermore, these services have varying granularity and do not provide any semantics in their descriptions, making unsupervised data extraction extremely difficult.

These observations serve to clarify the two broad characteristics of our work: Firstly, at the Web site level, the challenge is to extract data from a number of pages, generated from many templates. This requires determining homogenous clusters of pages having similar templates so that we can induce wrappers for these clusters. Another implicit requirement is that of learning the structure of a Web site through navigational steps. This is essential because our system needs to know which wrapper to apply during data extraction while traversing through the Web site. Secondly, at the page level, the presence of labeled data gives us the opportunity to segment data records and fields based on these labels, and to accommodate the dynamic structure of pages by using these labels to extract data, rather than analyzing and learning the entire Web page structure in a regular expression-like syntax. However, these labels must themselves be identified. Although the vocabulary for labels converges across different sources in a domain [7], it is not trivial to manually provide a set of possible labels (which can number in their hundreds) to aid in identification of data regions. Therefore, a desirable approach would be to automatically identify the labels.

¹ Complete survey available at <http://sabiork.villa-bosch.de/labelsurvey.html>

² Databases indexed by the Nucleic Acids Research Journal (<http://www3.oup.co.uk/nar/database/c/>). Complete survey available at <http://sabiork.villa-bosch.de/servicesurvey.html>

3 Wrapper Induction

We present our algorithm for page-level wrapper induction in section 3.1. Subsequently, in section 3.2, we describe how this algorithm is used in our site-wide wrapper-induction method. In section 3.3, we discuss a technique to automatically detect and correct certain erroneous results of our induction algorithm in section 3.1.

3.1 Page-Level Wrapper Induction

Our wrapper induction algorithm relies on multiple sample *instance pages* from a *class of pages*. We borrow this terminology from [9], which describes a *class of pages* in a site as a collection of pages which are generated by the same server-side script or program. Different inputs to this script result in different *instance pages*. We clarify this further using Figure 1, our running example, which shows two instance pages of a class of pages³. The wrapper induction algorithm is shown in Figure 2.

We note that, upon querying, the initial response pages generated by a deep Web source belong to the same class⁴. Therefore, we can probe a source with different inputs, and use the resulting initial pages to learn a wrapper, without having to cluster similarly structured pages. In fact, for a given Web site, the site-wide wrapper induction process is bootstrapped by using these initial result pages and learning a wrapper for this class.

Entry	C00221	Compound	Entry	C00185	Compound
Name	beta-D-Glucose		Name	Cellullobiose; 1-beta-D-Glucopyranosyl-4-D-glucopyranose	
Formula	C6H12O6		Formula	C12H22O11	
Mass	180.0634		Mass	342.1162	
Reaction	R00026 R01520 R01521 R01522 R01600 R01601 R01602 R02187 R02887 R03256 R04783 R06077 R06092 R06110 R06144		Reaction	R00026 R00306 R00952 R01441 R01442 R01443 R01444 R01445 R02365 R02886	
Enzyme	1.1.1.47 1.1.3.4 1.1.3.5 2.7.1.1 2.7.1.2 2.7.1.63 3.1.6.3 3.2.1.21 3.2.1.23 3.2.1.85 5.1.3.3		Enzyme	1.1.99.18 2.4.1.20 2.7.1.85 3.2.1.4 3.2.1.21 3.2.1.74 3.2.1.91 5.1.3.11	

Fig. 1. Results obtained by probing KEGG Compound with C00221 and C00185 respectively

Briefly, the algorithm compares text entries on the sample pages and identifies some (possibly not all) data entries among them. These data entries are subsequently used to identify bigger data regions, so that more data entries can be discovered. A label is then selected for each data entry from text entries outside the data regions, based on vicinity. Our approach is based on the DOM⁵ representation of Web pages, and uses XPath⁶ for performing the above operations on the DOM tree. The output of the wrapper is a collection of XPath expressions, each pointing to a label and associated data region.

We now explain each step of the algorithm in detail using our running example. Each step is annotated to its corresponding location in the algorithm in Figure 2.

³ From KEGG [15]. Portions of these pages have been removed to simplify the discussion and to save space, while remaining true to the challenges encountered

⁴ In certain cases, probing a source with an imprecise keyword leads to a disambiguation step. This is a separate research issue and we don't address it in this paper. We assume exact keywords are used to perform the search, as explained in Figure 2.

⁵ W3C. Document Object Model. <http://www.w3.org/DOM/>

⁶ W3C. XML Path Language (XPath 2.0) Recommendation. <http://www.w3.org/TR/xpath20/>

```

Input: n Web pages P
Output: R: L => Xgr //L is a set of labels. Xgr is a set of XPathS to data entries. R is a map
from each label l in L to each data XPath dχ in Xgr.

Start
For each sample page pi in P{
1 For each text entry t in pi
2 If t is unique to pi, Add t to Di;
Else Add t to Oi; }
3 For each pi in P{
Xdi = get_XPath(Di);
Xoi = get_XPath(Oi);}
4 Dio, Oio, Xio, Xid = reclassify(Xio, Xid); // grow the data regions, and reclassify data
For each XPath dχ in Xid{
5 Find closest XPath lχ in Xio; // search for XPath of most suitable label in O
If the corresponding text (label) in Oio is not in R
X = {dχ}; // X is a set containing all data XPathS associated with one label
Else
X = X U {dχ};
R = lχ => X; // XPath of label is mapped to set of corresponding data XPathS
For each lχ in R{
6 Generalize corresponding X to Xg; // Create a single XPath from all paths in set X
7 Find relative path Xgr from lχ to Xg; // relative path from label to data
Replace X with Xgr;
Replace lχ with l; // replace Xpath with the corresponding label
}
}
End

```

Fig. 2. Wrapper Induction Algorithm

1. The two HTML pages are converted to well-formed XHTML using an HTML-parsing library, i.e. TagSoup⁷, so that standard XML tools can be applied to them. Finally, each page is *screen-scraped* to obtain a set T of values contained in all text nodes. Both sets (T₁ and T₂ in our example) thus contain a union of presentation text, labels and data entries.

2. We compare both sets to initially classify some data entries. Mutually exclusive entries in T₁ and T₂ are classified as data entries (D₁, D₂), and the remaining as non-data entries, or “*Other*” (O₁, O₂). For example:

```

D1 = {C00221, beta-D-Glucose, ..., R01520, 1.1.1.47, ...}
D2 = {C00185, Cellobiose, ..., R00306, 1.1.99.18, ... }
O1 = {Entry, Name, ..., Reaction, R00026, Enzyme, ..., 3.2.1.21}
O2 = {Entry, Name, ..., Reaction, R00026, Enzyme, ..., 3.2.1.21}

```

Notice that since the data entry R00026 occurs in both instance pages, it is erroneously classified as *Other* at this point.

3. We compute XPath expressions for each entry in the above sets. The expression determines the unique path along the DOM tree for the XHTML file, from the root node to the node containing the entry. For example, the XPath for C00221 is:

```
html/body/.../code[1]/table[1]/tr[1]/td[1]/code[1]/text ()
```

4. We use the XPath expressions to reclassify some data entries which might have been wrongly classified in the previous step (such as R00026, 3.2.1.21). This can be considered as *growing* of a data region, whereby data entries are used to reclassify other entries in their vicinity as data. This reclassification step compares an XPath of a data entry with that of an entry not classified as data using the following two rules:

⁷ A SAX-compliant HTML Parser. <http://home.ccil.org/~cowan/XML/tagsoup/>

Rule 1 (Last Element Node Rule): If two XPath expressions are identical and differ only at the ordering of the last element node, and this last element node in the data XPath precedes the last element node in the non-data XPath, the non-data entry is re-classified as data.

This rule can be explained from an example in Figure 3(a). As shown in the figure, this rule uses data elements to automatically grow data regions towards the right in a table-row. While Figure 3 only shows the example of a table, it is important to emphasize that since this rule is independent of tag names, it works on tags other than those associated to HTML tables, for example, downwards in a list or in a succession of anchor tags. For instance, for the latter case, the XPath expressions for successive anchor tags (even if they are separated by line breaks) could be:

`html/body/a[1], html/body/a[2]`

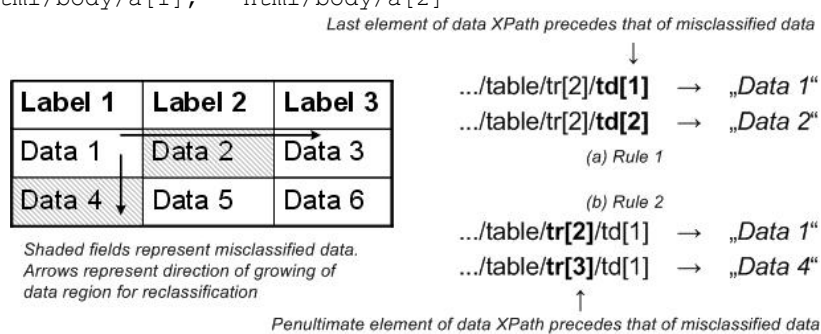


Fig. 3. Growing of data region

It is easy to see from the above example that this rule grows data regions in many types of HTML structures.

Rule 2 (Penultimate Element Node Rule) If the two XPath expressions are identical and differ only at the ordering of the penultimate element node, and this penultimate element node in the data XPath precedes the penultimate element node in the non-data XPath, the non-data entry is re-classified as a data entry

This rule is similar to Rule 1, except it grows data regions down a table-column as shown in Figure 3(b)

We note that our rules for growing data-regions operate in only two directions. This is based on the observation that labels generally occur above or towards the left of data [33]. Therefore, we restrict our re-classification in these two directions.

After this re-classification step, we have the modified sets:

$D_1' = \{C00221, \text{beta-D-Glucose}, \dots, R01520, 1.1.1.47, \dots, 3.2.1.21\}$

$D_2' = \{C00185, \text{Cellobiose}, \dots, R00306, 1.1.99.18, \dots, 3.2.1.21\}$

$O_1' = \{\text{Entry, Name}, \dots, \text{Reaction, R00026, Enzyme}, \dots\}$

$O_2' = \{\text{Entry, Name}, \dots, \text{Reaction, R00026, Enzyme}, \dots\}$

Note that R00026 is not re-classified (incorrectly) because there are no data entries which can grow the data region in its direction. The sets of non-data entries, O_1' and O_2' , now contain both presentation text, as well as labels for our data entries.

5. For each data entry in a data set, we now select the closest non-data entry as its label. This can be achieved by comparing XPath of the data entry against the XPath expressions of the non-data entries. The closer a non-data entry is to a data entry, the more element nodes in their corresponding XPath expressions will be matched before a

mismatch. The closest element will have the longest common leading path, which is classified as the label. For example, the XPath for data entry 1.1.1.47 is given by:

```
html/.../table[1]/tr[8]/td[1]/.../code[1]/a[1]
```

Some XPath expressions for the set of non-data entries include, for example:

```
html/.../table[1]/tr[6]/th[1]/.../code[1]/ ("Reaction")
```

```
html/.../table[1]/tr[8]/th[1]/.../code[1]/ ("Enzyme")
```

The latter XPath has the longest sequence of matching nodes with the XPath of our data element (indicated by the bold-face font above). Therefore, the label (“Enzyme”) and corresponding XPath are associated with data entry 1.1.1.47 and its XPath (This association is represented by the “=>” symbol in Figure 5).

Note that multiple data elements can be associated with a single label in such a manner, as shown in Table 1. The last row of the table shows that the data entry R00026, which has been misclassified previously, has been selected as a label for data entries R01520, R01521 etc, as it found to be the closest non-data entry.

Table 1. Two labels inferred, with corresponding data entries and their XPath expressions.

Label – With XPath	Data Entry	XPath of Data Entry
Enzyme html/.../th[1]/.../code[1]/	- 1.1.1.47	html/.../td[1]/.../code[1]/a[1]
	1.1.3.4	html/.../td[1]/.../code[1]/a[2]

R00026 html/.../th[1]/.../code[1]/a[1]	- R01520	html/.../td[1]/.../code[1]/a[2]
	R01521	html/.../td[1]/.../code[1]/a[3]

6. The XPaths of data entries classified to the same label are then generalized to form a single XPath expression. The XPaths for entries in Table 1 can be generalized as:

```
html/.../table[1]/tr[8]/td[1]/.../code[1]/a[position() ≥ 1]/text ()
```

```
html/.../table[1]/tr[6]/td[1]/.../code[1]/a[position() ≥ 2]/text ()
```

The last XPath expression above, for example, selects all text entries pointed to by a collection of anchor tags, starting from the second anchor tag. This is required as sample pages may only contain a small number of multiple data entries associated to a label. What is required is that we recognize and generalize that multiple number of data entries are associated for that label, rather than the number of data entries *seen* by the wrapper induction algorithm.

7. A relative path from the label to its corresponding generalized data path is computed. For the “Enzyme” label in Table 1, the relative path to its data is:

```
../.../.../td[1]/.../code[1]/a[position() ≥ 2]/text () (1)
```

Finally, the XPath for the label is replaced with an *anchored* XPath expression, i.e., an XPath which directly access the text node, and does not utilize ancestor nodes. For the “Enzyme” label:

```
//*[text()='Enzyme'] (2)
```

Concatenating (1) with (2) gives us, for label “Enzyme”:

```
//*[text()='Enzyme']../.../td[1]/.../code[1]/a[position() ≥ 2]/text ()
```

The wrapper, thus, comprises of a collection of labels associated with a generalized anchored XPath expression to extract corresponding data.

Discussion It is worth noting here that the wrapper learnt by our algorithm is not tied to the structure of a class of pages. The wrapper anchors, or pivots, to a particular label, and finds a relative path from the label to associated data entries. As noted in section 2, data pages returned by Web databases can be very dynamic. Our approach is beneficial in this case, as well as in the case that a Web site undergoes a template redesign, for instance. As long as the relative path from a label to its corresponding data remains the same, there would be no need to re-learn the wrapper. The only other limitation is that of labels remaining constant, and as we mentioned in section 2, changes in names of real-world entities, such as biological concepts, is extremely rare. Finally, recall that the wrapper-induction process for our running example results in the misclassification of “R00026” as a label. We discuss a technique to automatically detect and possibly correct such errors in section 3.3.

3.2 Site-Wide Wrapper Induction

As we noted in section 1, data-intensive sites, such as those in the Life Sciences domain, have their data scattered across multiple pages. Therefore, we need a wrapper-induction strategy that extracts data from multiple pages, which might belong to different page classes. This implies that we need to not only discover which pages returned by the server belong to the same class, but also to distinguish between classes and navigational steps between them. We make the following observations:

1. Not all pages of a Web site contain data, for example, pages pointed to by navigational menus, help pages, contact information etc. Therefore, we do not wish to discover all page classes. Rather, we wish to perform *targeted crawling* to only seek out *data-pages* and discover their classes.
2. We observe the concept of *link-collection* [9], which refers to anchor links in a page (class) that share the same path in the DOM tree, from the root element to their parent or grandparent element. As a result, these hyperlinks appear grouped together in the rendered page. In our example of Figure 1, the links on the reaction names form a link-collection, as well as those on the enzyme names. A link collection may be a singleton as well, comprising only a single hyperlink. We also note that hyperlinks in such a collection might not point to the same class of pages. For example, links in a navigation bar or those in categorization menus. However, link-collections that have been *classified* as being over data regions point to the same class of pages. For example, all hyperlinks on enzyme names point to “enzyme details” pages.
3. Pages belonging to the same class contain similar set of labels. However, due to the highly dynamic nature of pages, some labels may be omitted (e.g. NULL values in databases), but their ordering typically remains the same, as they are generated by scripts. If the order of labels on two pages is different, then their page-structures will most likely be different as well. Furthermore, it is highly unlikely that a site will have two different templates to display the same set of labels in the same order.

We base our approach for site-wide wrapper induction on these assumptions:

1. Given the initial result output of a deep Web source, all data-intensive pages can be reached by iteratively following link-collections that occur on data regions. This assumption allows us to do targeted crawling for data-intensive pages, and eliminate navigation bars and menus etc.

2. A pre-classified link-collection points to the same class of pages.
3. Classes of pages can be distinguished from each other based on the labels they contain, and their order.

We now model our site-wide wrapper-induction problem as follows:

A page class C_i is defined by $C_i = \text{SEQ}_i$, where $\text{SEQ}_i = (\ell_{i1}, \ell_{i2}, \dots)$, a sequence of labels $\ell_{i1}, \ell_{i2}, \dots$ described in page class C_i in this order of arrangement. These labels may have link-collections associated with them. Two classes C_i and C_j are considered not equal if $\text{SEQ}_i \neq \text{SEQ}_j$

Our site model is given by a collection of navigation steps:

$$R_{ijn} = C_i \rightarrow \ell_{im} \rightarrow C_j \quad (i \neq j, i, j, n, m \geq 0)$$

Where ℓ_{im} is the m -th label in page class C_i , the associated link-collection of which points to pages of class C_j .

The goal of the site-wide wrapper induction algorithm is to find the following:

1. $C_i \neq C_j \quad (i \neq j, i, j \geq 0)$
2. $R_{ijn} = C_i \rightarrow \ell_{im} \rightarrow C_j \quad (i \neq j, i, j, n, m \geq 0)$

<pre> 1 Input: S={C₀}, C₀=(ℓ₀₁, ℓ₀₂,...); // Set of all page classes discovered. C₀ corresponds to the initial results page of a deep Web source. W = {}; // Set of navigation steps between page classes. Output: S={C_i} (i≥0) W={R_{ij}} (i,j≥0, i≠j) Start S' = S; Do{ 2 For each C in S' { // for each class in our set For each ℓ in C { // for each link-collection associated with a label Follow ℓ; // follow the link-collection 3 induceWrapper C_{new}; // induce wrapper 4 if (C_{new} ∉ S) { // if this is a new class Add C_{new} in S and S'; // add it to our set R = (C → ℓ → C_{new}); // form the navigation step Add R in W; // add the step } } Remove C from S'; 5 } While(S' ≠ NULL) // all classes' link-collections have been explored End </pre>
--

Fig 5: Site-Wide Wrapper Generation Algorithm

That is, all possible data-intensive page classes, and the navigation steps between them. The algorithm for site-wide wrapper induction is presented in Figure 5. We explain the algorithm using our example of Figure 1. Each step explained below is annotated to its corresponding location in the algorithm in Figure 5.

1. The algorithm starts with an input of the initial page class C_0 , which corresponds to the initial response pages of the Web source. In our case, this is the page class that is generated by the sample pages of our running example, shown in Figure 1.
2. For each label in this class, corresponding link-collections are followed. Assuming we follow the link-collection of “Enzyme”, a sample result is shown in Figure 6.
3. According to our assumption 2, these pages belong to the same class. We learn a wrapper for this page class using our algorithm in Figure 2, with these sample pages as input. We note that not all links in a link-collection need to be followed. Our initial experiments have shown that ~9 sample pages yield a very good result (section 4).

4. If this wrapper learning process results in a new class, according to our assumption 3, we add this new class to our sets, and define its corresponding navigational steps. In our example, a new class is created, $C_1 = (\text{Entry, Name, Class, ...})$, as well as a navigation step $R = (C_0 \rightarrow \text{"Enzyme"} \rightarrow C_1)$.
5. The above steps are repeated for each new page class that is learnt in step 3.

Entry	EC 2.7.1.2	Enzyme
Name	glucokinase; glucokinase (phosphorylating)	
Class	Transferases; Transferring phosphorus-containing groups; Phosphotransferases with an alcohol group as acceptor	

Fig 6: Small excerpts of pages obtained from following "Enzyme" link-collection

3.3 Error-Detection by Mutual Reinforcement

The natural residual output of our site-wide wrapper-induction approach is labeled data. These labels and data can be used to automatically detect and possibly correct errors in our wrapper-induction method for a page-class. We observe that some data entries reappear on different page classes. For example, the enzyme classification numbers in Figures 1 and 6. If the reappearing entries have been correctly classified as data across different page-class wrapper induction runs, then this enforces our confidence that the classification is correct. On the other hand, if, for example, some entries are classified as labels or presentation text by some wrappers and data by others, then this points to a misclassification. This indicates that not enough sample pages were available to distinguish between data, labels and presentation text. We can address this by providing more samples for these page classes. We call such a mismatch as *label-data mismatch*. For example, recall from Section 3.1 that the data entry R00026 was misclassified as a label in our running example (for page class C_0). When we follow the "Reaction" link-collection, we come across the page shown in Figure 7. While learning the wrapper for this class of pages (C_1), R00026 will be (correctly) classified as data. Based on this mismatch, we introduce more learning pages for both C_0 and C_1 . In our example, any page for class C_0 which does not contain "R00026" as an entry will force our algorithm to classify this entry as data, thereby correcting the label to "Reaction" as well. The other type of mismatch is *label-label mismatch*, where the same data entry is assigned different labels across page classes. Recall our observation from section 2 that the same data entries can be labeled differently across different page classes. This can be observed from Figures 1 and 7, where R00026 is labeled as "Reaction" and "Entry", respectively. Therefore it is impossible to detect whether a label-label mismatch was an error or a correct classification. We slightly modify our site-wide wrapper induction algorithm to incorporate automatic error detection and correction for label-data mismatches. We introduce this mutual reinforcement step each time a new page class is created. The entries classified as data in this new class are compared with labels of previously formed page classes. If a mismatch is found, more sample pages for this new class and conflicting page class are introduced until the mismatch is resolved.

Entry	R00026	Reaction
Name	beta-D-Glucoside glucohydrolase	
Definition	Cellobiose + H2O <=> 2 beta-D-Glucose	
Equation	c00185 + c00001 <=> 2 c00221	

Fig 7: Top-most portion of "Reaction" page of R00026 From KEGG

4. Results and Evaluation

We have developed a prototype in Java which implements our algorithms. We use it to perform some preliminary experiments on three real-world biochemical sources, namely KEGG[17], ChEBI[12] and MSDChem[13]. All these sources provide basic qualitative data, and are often used for reference or annotation in more specialized domains. We use a simple random sample of input values for search forms of these sites in order to probe and induce their initial results page⁸. A Web-crawler based on httpUnit⁹ was manually configured to submit the search forms with these values.

4.1 Page-Level Wrapper Induction Results

We verify the accuracy of our wrappers by applying them to sets of 20 test pages. We manually count the total number of data entries and note corresponding labels across these test pages a priori, and determine the precision and recall of our algorithm in retrieving these data entries and classifying them with the right label.

Before discussing the results, we briefly explain some conditions under which false-negatives and false-positives occur. False-positives generally occur when there is unlabeled data present in the pages. These data usually occur at the top of a page, such as a heading or a large caption, and are often *redundant* data entries, as they reappear as labeled data later in the same page (e.g. compound identification numbers etc). False-positives also occur when data entries are misclassified as labels (as “R00026” in section 3.1). This *may* also result in a corresponding false-negative for the *missed* label (“Enzyme” by misclassification of “R00026”). False-negatives also occur when sample pages do not contain the labels. This is a limitation for all wrapper induction approaches – you can only learn what you see. Our results are summarized in Table 2. The wrapper for the page class belonging to KEGG Reaction has a perfect precision and recall, with a relatively small training set. This owes to the fact that there are frequent pages in KEGG Reaction which contain all labels. Our algorithm is thus able to correctly induce a wrapper for this class. The wrappers for KEGG Compound and ChEBI are unable to retrieve data entries corresponding to the labels which were not learnt (“Sequence” and “IN Number” respectively). Manual inspection of the training set revealed that none of our 15 training pages contained those labels, which leads us to believe that they occur rarely. This can only be removed with more sample pages. The wrapper for MSDChem has false-positives as a result of false classification of redundant data entries to some presentation text in the learning phase. The results for MSDChem are quite interesting, as they demonstrate the usefulness of our *data-region growing* approach. Unlike other sources, pages in MSDChem have a very static structure – no labels are omitted from the pages when corresponding data entry is NULL, as shown in Figure 8. This means that the frequency of data fields being NULL (or “Not Assigned” in this example) is very high. Such fields are not classified as data in our algorithm, as they are constant across many pages. However, we note that the label-data pairs are arranged in a

⁸ The values can be collected from downloadable flat files or Web services provided by each source.

⁹ A Java library for automated testing of Web sites. <http://httpunit.sourceforge.net/>

(invisible) table, as shown in Figure 8. Therefore, through rule 2 in Section 3.1, an entry classified as data at the top of the data column in Figure 8 reclassifies all entries below it in the column as data. This accounts for perfect recall, with 3 sample pages.

Table 2. Results from applying wrappers to 20 test pages each. (L=Labels, T=data entries in 20 test pages, S=samples, R=Retrieved but not classified correctly, IR=Incorrect Retrieval)

SOURCE	# L	#T	#S	TP	FN	FP		P	R
						#R	#IR		
KEGG Compound http://www.genome.jp/kegg/compound/	10	762	3	411	351	46	0	89.9%	53.9%
			6	638	124	39	0	94.2%	83.7%
			9	759	3	0	0	100%	99.6%
			12	759	3	0	0	100%	99.6%
			15	759	3	0	0	100%	99.6%
KEGG Reaction http://www.genome.jp/kegg/reaction/	10	205	3	173	32	0	0	100%	84.4%
			6	205	0	0	0	100%	100%
			9	205	0	0	0	100%	100%
			12	205	0	0	0	100%	100%
			15	205	0	0	0	100%	100%
ChEBI http://www.ebi.ac.uk/chebi	22	831	3	595	236	41	0	93.5%	71.6%
			6	713	118	0	0	100%	85.8%
			9	809	22	0	0	100%	97.3%
			12	829	2	0	0	100%	99.7%
			15	829	2	0	0	100%	99.7%
MSDChem http://www.ebi.ac.uk/msd-srv/msdchem/cgi-bin/cgi.pl	30	600	3	600	0	0	20	96.7%	100%
			6	600	0	0	20	96.7%	100%
			9	600	0	0	20	96.7%	100%
			12	600	0	0	20	96.7%	100%
			15	600	0	0	20	96.7%	100%
Average (based on final wrappers for each source)								99.1%	99.8%

Overall, we observe that we can get very high precision and recall (~99%, ~98% respectively) from ~9 samples. The precision can be improved with more samples, especially if they contain rarely occurring labels.

```

Formula          C10 H16 N5 O13 P3
Defined at       1999-07-08
Last modified at 2007-08-16
EBI name         ADENOSINE-5'-TRIPHOSPHATE
EBI Id          not assigned
Additional name  ADENOSINE-5'-TRIPHOSPHATE
Classification   NUCLEOTIDES
Cas reg number  not assigned
Therapeutic category not assigned
Merck Id        not assigned
Polymer topology not assigned
Polymer code    not assigned
Polymer sub type not assigned
Hetgroup type  NON-POLYMER
Obsoleted      not assigned
Parent         not assigned
Topological variant not assigned

```

Fig 8: Portion of MSDChem page for “ATP”, showing unassigned values

4.2 Site-Wide Wrapper Induction

In this section, we present our results for site-structure discovery, which together with the wrapper induction algorithm constitutes our site-wide wrapper induction approach. We manually model all three sources, which involves manually determining classes for data pages for a source, and the navigation steps for generating these classes. MSDChem and ChEBI have relatively simple models. KEGG on the other hand has a very complex model. It actually consist of a number of back-end database schemas, each having its unique Web interface, with more than 30 page classes. For this paper, we restrict our manual model to a specific sub-site (KEGG Compound, Drug, Reaction, RPair, Enzyme and Orthology).

Next we use our site-wide wrapper induction algorithm to learn the corresponding wrappers. We restrict our algorithm from exploring the KEGG portal outside our manually defined boundary, allowing it to discover navigation steps within the aforementioned sub-site. Finally, we apply our site-wide wrappers to the three sources to extract data. We limit the execution so that our system extracts data from only 20 test instances of each class of the Web site. We manually count the total number of data entries and note corresponding labels across all test pages a priori, and determine the accuracy of the algorithm, with the results shown in Table 3.

Table 3: Site-wide wrapper evaluation. (#C = Total number of classes, #C' = Number of classes discovered, T = data entries in 20 test pages)

SOURCE	#C	#C'	#T	TP	FN	FP	P	R
MSDChem	1	1	N/A	N/A	N/A	N/A	N/A	N/A
ChEBI	3	1	1711	1195	516	0	100%	69.8%
KEGG	10	7	6223	5044	1179	188	97%	81.1%
Average							98.5%	75.5%

We observe that for MSDChem, even though the navigation steps constituting the site model are correct, the site-wide wrapper induction fails. Upon inspection, we notice that the navigation step from a page instance actually results in the same instance. This implies that the MSDChem Web site consists of a large number of leaf nodes only, having no hyperlinks connecting them to each other. For ChEBI, we have a perfect precision, but a low recall. This indicates that the two classes our algorithm failed to retrieve had rich data regions. Our algorithm also fails to retrieve 3 classes in the KEGG sub-site, though a relatively higher recall suggests these missing classes did not contain as big a data region as in the case of ChEBI. Furthermore, we have some misclassifications in some page wrappers for KEGG which slightly lower the precision for the corresponding site-wide wrapper for KEGG. Overall, a relatively high precision suggests that our assumption that all link-collections associated with data regions point to classes of pages containing data is indeed correct. However, the relatively low recall seems to suggest that we need to relax the restriction that only link-collections associated with data regions should be followed. Instead, links close to data regions should also be followed.

5. Related Work

The earliest approaches to wrapper induction, including [20] and [26] required manually labeled training sets. Due to the large size of the Web and its dynamic nature, supervised techniques do not scale well. Recent attempts have focused on fully automatic wrapper induction techniques. The reader is instructed to read [22] for a survey on wrapper induction techniques. RoadRunner [8, 10] is an automatic wrapper induction algorithm that is closest to our approach, as it uses multiple sample pages of a page-class. However, unlike our approach, it compares the structures of the sample pages to learn a regular expression, which takes into consideration the mismatches between text and HTML tags across the samples. This regular expression based wrapper is thus tied directly to the page structure. As we noted in section 2, pages from deep Web sources are often very dynamic, where concepts that are NULL are often omitted. RoadRunner would thus require a large number of sample pages, covering all possible types of such omissions, so that its regular expression can accommodate for this dynamic behavior. Lixto [3] and W4F [28] use XPath-like languages “Elog” and “HEL” respectively, and both offer visual tools for creating wrappers in an unsupervised manner. The user selects data of interest in a Web page, and a path from the root of the page to the target node is generated in the respective languages. Therefore, manual identification of data elements is required for each page, which can be laborious for pages containing numerous data entries. ANDES [25] is based on XPath and requires the user to manually provide XPath expressions to extract data. [1] builds on ANDES to induce the XPath expressions using tree traversal patterns but requires annotated examples. IEPAD [6], DeLA [33], ViNTs [27], DEPTA [35] and ViPER [29] are unsupervised wrapper induction techniques that are all based on one common assumption: Data regions in Web pages are constituted by at least two spatially consecutive records that are structurally and visibly similar. This assumption partially holds for result pages of search engines, online listings and E-commerce Web sites, but not for scientific repositories on the Web, as is apparent from our example in Figure 1. Even in the case of E-commerce sites and listings, the initial response pages of a search do exhibit a repetitive structure comprising of records, but the *details* pages describing each result do not exhibit this repetitiveness. All approaches cited above perform wrapper induction on a single class of pages, whereas our approach attempts to automatically classify pages in a Web site into appropriate classes, learn wrappers for each class and discover rules for applying these wrappers on Web pages encountered on the Web site. IDE [36] extracts structured data from different classes of Web pages. It starts with one labeled training page, indicating the information to be extracted. It proceeds to extract corresponding data from test pages based on the similarity between a consecutive sequence of tags before and after the labeled data and the data in the test pages. Whenever extraction fails for a page, it is manually labeled. This requires foreknowledge about which information must be extracted, and assumes that the same information is present and to be extracted from all classes.

We are only aware of one approach to automatic site-structure discovery [9], which also constitutes the main motivation for our approach. The focus of their work is slightly different from ours. It tries to efficiently discover the entire site-structure, whereas we focus on discovering only data-rich regions. Their approach to clustering

of Web pages into classes is based on the assumption that pages belonging to the same class contain link-collections that are in a structurally similar arrangement. Based on structural similarity of these link collections, they group Web pages into classes. This is a good assumption for sites that do not have *leaf* pages which do not have any links, such as help pages, FAQs, contacts, legal disclaimers etc. In the absence of hyperlinks, all these pages would be classified into a single class (because their link-collections have the same structure), even though these pages may exhibit considerable structural variations. Our approach is also based on an assumption over link-collections, but contrary to [9], we assume that link-collections classified to the same label point to pages belonging to the same class. A closely related research field is that of focused crawling, which can either be content-based or structure-based. Content based-crawlers [4] fetch Web pages relevant to a given topic, which is specified by example Web pages. Structure-driven crawling relies on the structural similarity between given sample pages and pages of a Web site to find similar pages [30] or between the pages encountered in a Web site to cluster similar pages [11].

6. Conclusions

We have described a novel wrapper induction technique to extract labeled data from data-intensive Web pages of deep Web sources. The approach takes advantage of the peculiarities typically associated with Life Science Web sites, most notably that they contain labeled data. Our approach is unique in that it automatically classifies structurally similar pages into classes which can then be used for learning wrappers. Navigation steps that are retrieved during the site-wide wrapper induction phase are used to associate wrappers to classes of pages, allowing us to automatically select and apply a wrapper for a page in the Web site. The approach is fully automatic, the samples required for page-level wrapper induction are collected automatically and do not require any manual labeling. The approach does not need fine-tuning of any heuristics or parameters, but does require the presence of labels.

References

1. Anton, T.: XPath-Wrapper Induction by generalizing tree traversal patterns. In Workshop on Web Mining, in ECML/PKDD (2006)
2. Barbosa, L., Freire, J.: Searching for Hidden-Web Databases. In WebDB, p 1-6 (2005)
3. Baumgartner, R., Flesca, S., Gottlob, G.: Visual Web Information Extraction with Lixto. Proc. 27th Internatl. Conference on Very Large Data Bases:119 – 128 (2001)
4. Chakrabarti, S. et al. Mining the Web's link structure, Computer 32 (8) 60–67 (1999)
5. Chang, K. C.-C., Cho, J.: Accessing the Web: From Search to Integration. In Proceedings of the 2006 ACM SIGMOD Conference (2006)
6. Chang, C.-H., Hsu, C.-N., Lui, S.-C.: Automatic information extraction from semi-structured web pages by pattern discovery. SCI expanded, 35(1):129–147, Special Issue on Web Retrieval and Mining (2003)
7. Chang, K. C.-C., He, B., Zhang, Z.: Mining Semantics for Large Scale Integration on the Web: Evidences, Insights and Challenges. SIGKDD Explorations, 6(2):67-76 (2004)

8. Crescenzi, V., Mecca, G., Merialdo, P.: RoadRunner: Towards Automatic Data Extraction from Large Web Sites. In VLDB p. 109-118 (2001)
9. Crescenzi, V., Merialdo, P., Missier, P.: Clustering Web pages based on their structure. *Data & Knowledge Engineering* 54:279–299 (2005)
10. Crescenzi, V., Mecca, G., Merialdo, P.: Improving the expressiveness of ROADRUNNER. *SEBD* 62-69 (2004)
11. de Castro Reis, D. et al. Automatic web news extraction using tree edit distance. In WWW13, pp. 502–511 (2004)
12. Degtyarenko, K. et. al.: ChEBI: a database and ontology for chemical entities of biological interest. *Nucleic Acids Res.* 36, D344–D350 (2008)
13. Golovin, A. et. al.: E-MSD: an integrated data. *Nucleic Acids Research*, 32 (Database issue), D211-D216 (2004)
14. He, B., Chang, K. C.-C.: Statistical Schema Matching across Web Query Interfaces. In SIGMOD Conference, pages 217-228 (2003)
15. He, H., Meng, W., Yu, C. T., Wu, Z.: WISE-Integrator: An Automatic Integrator of Web Search Interfaces for E-Commerce. In VLDB, pages 357-368 (2003)
16. He, B., Tao, T., Chang, K. C.-C.: Organizing structured web sources by query schemas: a clustering approach. In CIKM, pages 22-31 (2004)
17. Kanehisa, M.: The KEGG database. *Novartis Found Symp*, 247:91-101; discussion 101-3, 119-28, 244-52 (2002)
18. Knoblock, C., Kambhampati, C.: Information Integration on the Web. In AAAI (2002)
19. Kabra, G., Li, C., Chang, K. C. C.: Query Routing: Finding Ways in the Maze of the DeepWeb. *WIRI 2005*: 64-73 (2005)
20. Kushmerick, N.: Wrapper Induction for information extraction. In ICAI (1998)
21. Kushmerick, N.: Learning to Invoke Web Forms. *CoopIS/DOA/ODBASE*, 997-1013 (2003)
22. Laender, A. H. F., Ribeiro-Neto, B., Silva, A. S. D., Teixeira, J. S.: A brief survey of web data extraction tools. *ACM SIGMOD Record*, 31(2):84–93 (2002)
23. Lu, Y., et al.: Clustering e-commerce search engines based on search interface pages using WISE-Cluster. *Data Knowl. Eng.* 59(2):231-246 (2006)
24. Madhavan, J., et al.: Corpus-based Schema Matching. In ICDE, pages 57-68 (2005)
25. Myllymaki, J., Jackson, J.: Robust Web Data Extraction with XML Path Expressions. IBM Research Report (2002)
26. Muslea, I., Minton, S., Knoblock, C.: Stalker: Learning extraction rules for semistructured, web-based information sources. AAAI-98:AI and Information Integration Workshop (1998)
27. Meng, W., Raghavan, V., Yu, C.: Fully automatic wrapper generation for search engines. In WWW14 (2005)
28. Sahuguet, A., Azavant, F.: Building intelligent Web applications using lightweight wrappers. *Data Knowl. Eng.* 36(3): 283-316 (2001)
29. Simon, K., Lausen, G.: ViPER: augmenting automatic information extraction with visual perceptions. *CIKM 2005*: 381-388 (2005)
30. Vidal, A. et al. Structure-driven crawler generation by example. *SIGIR'06*:292-299 (2006)
31. Wang, J., Wen, J.-R., Lochovsky, F. H., Ma, W.-Y.: Instance-based Schema Matching for Web Databases by Domain-specific Query Probing. In VLDB, p 408-419 (2004)
32. Wu, W., Doan, A., Yu, C. T.: WebIQ: Learning from the Web to Match Deep-Web Query Interfaces. In ICDE, page 44 (2006)
33. Wang, J., Lochovsky, F. H.: Data extraction and label assignment for web databases. In WWW12, p 187–196 (2003)
34. Zhang, Z., He, B., Chang, K. C.-C.: Understanding Web Query Interfaces: Best-Effort Parsing with Hidden Syntax. In SIGMOD Conference, pages 107-118 (2004)
35. Zhai, Y., Liu, B.: Automatic Wrapper Generation Using Tree Matching and Partial Tree Alignment. In AAAI 2006, Boston, USA, July 16 - 20 (2006)
36. Zhai, Y., Liu, B.: Extracting Web Data Using Instance-Based Learning. *WWW16* (2007)