Sample-based XPath Ranking for Web Information Extraction

Oliver Jundt[1] **Maurice van Keulen**[2]

[1] University of Twente, P.O. Box 217, 7500AE Enschede, The Netherlands. Email `o.jundt@student.utwente.nl`
[2] University of Twente, P.O. Box 217, 7500AE Enschede, The Netherlands. Email `m.vankeulen@utwente.nl`

Abstract

Web information extraction typically relies on a wrapper, i.e., program code or a configuration that specifies how to extract some information from web pages at a specific website. Manually creating and maintaining wrappers is a cumbersome and error-prone task. It may even be prohibitive as some applications require information extraction from previously unseen websites. This paper targets *automatic on-the-fly* wrapper creation for websites that provide attribute data for objects in a 'search – search result page – detail page' setup. It is a wrapper induction approach which uses a small and easily obtainable set of sample data for ranking XPaths on their suitability for extracting the wanted attribute data. Experiments show that the automatically generated top-ranked XPaths indeed extract the wanted data. Moreover, it appears that 20 to 25 input samples suffice for finding a suitable XPath for an attribute.

Keywords: Web information extraction, wrappers, XPath ranking

1. Introduction

Information on web pages is mainly targeted at humans, whereas in many cases it is desirable that this information is also available for processing by a computer. Only in rare cases websites provide machine interfaces for this purpose. Web information extraction (sometimes called web scraping or web harvesting) is an important technology to overcome this lack of service.

Web information extraction typically relies on a wrapper, i.e., program code or a configuration that specifies how to extract some information from web pages at a specific website. Wrappers work particularly well when web pages are generated using a fixed template, which is almost always the case nowadays. The template puts the same type of information, e.g., a product title, always at the same place surrounded by the same tag names and attributes. Wrappers can effectively exploit this constant structure.

Manually creating and maintaining wrappers is a cumbersome and error-prone task. If information is needed from multiple websites, hence from different templates, then each website needs its own wrapper. And, if the template changes, the wrapper is likely to malfunction. A wrapper programming approach may even be prohibitive as some applications require information extraction from previously unseen websites.

In this paper, we use information extraction from online bookstores as the running example. Bookstore websites usually provide a search field or form producing a so-called *search result page* which contains a list of results each carrying a link to the so-called *detail page*, a page with detailed information about one specific book. This 'search — search result page — detail page' setup is very commonly used in a wide variety of websites, not just online bookstores. We call the book details, e.g. 'Title' and 'ISBN', *attributes*, which have possible values like 'Moon Palace' and '0140115854'. The attribute values are the information we want to extract. The book itself we call an *object*. A source of detail pages for a specific type of objects is called a *provider*. For example, Amazon.com is a provider for book detail pages.

1.1. Problem Statement

This paper approaches the problem of web information extraction from a specific angle that enables automatic on-the-fly wrapper creation. Therefore, we define our problem statement as follows

> How to *automatically* find a wrapper that, for every detail page of a given provider, extracts the attribute data of the objects described on the detail pages. (Figure 1)

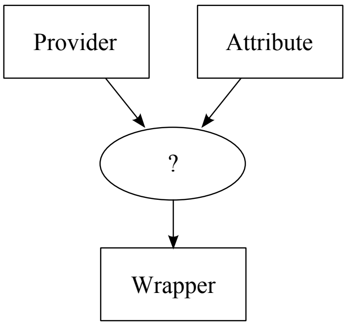


Figure 1: Abstract problem of automatic information extraction

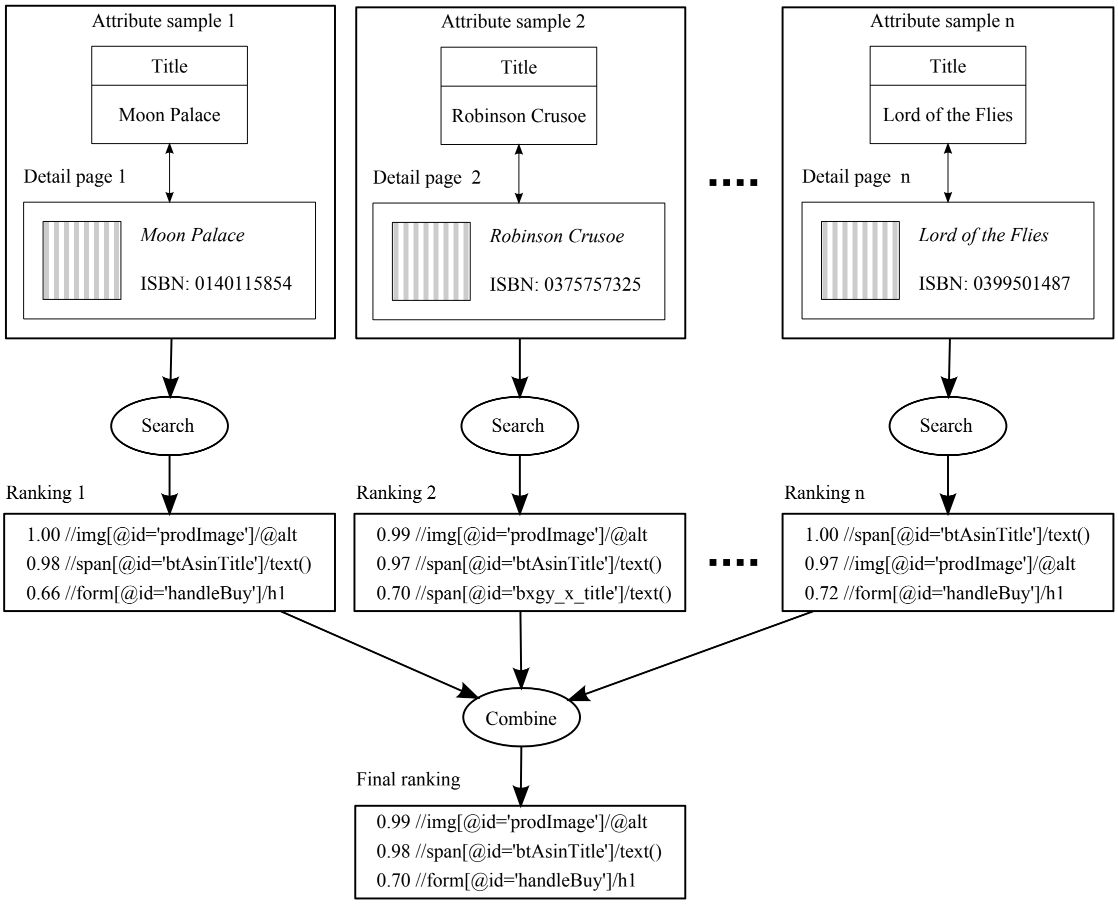

Attribute sample 1
Title
Moon Palace
Detail page 1
Moon Palace
ISBN: 0140115854
Attribute sample 2
Title
Robinson Crusoe
Detail page 2
Robinson Crusoe
ISBN: 0375757325
Attribute sample n
Title
Lord of the Flies
Detail page n
Lord of the Flies
ISBN: 0399501487
Search
Search
Search
Ranking 1
1.00 //img[@id='prodImage']/@alt
0.98 //span[@id='btAsinTitle']/text()
0.66 //form[@id='handleBuy']/h1
Ranking 2
0.99 //img[@id='prodImage']/@alt
0.97 //span[@id='btAsinTitle']/text()
0.70 //span[@id='bxgy_x_title']/text()
Ranking n
1.00 //span[@id='btAsinTitle']/text()
0.97 //img[@id='prodImage']/@alt
0.72 //form[@id='handleBuy']/h1
Combine
Final ranking
0.99 //img[@id='prodImage']/@alt
0.98 //span[@id='btAsinTitle']/text()
0.70 //form[@id='handleBuy']/h1

Figure 2: Approach idea

1.2. Approach

Our approach is illustrated with an example in Figure 2. The basic idea is to use XPath [1] as the extraction language and a small set of easily obtainable sample data to rank automatically generated XPaths on their suitability for extracting the wanted attribute values. The ranking score for a XPath describes how well the content of the node addressed by the XPath matches the searched attribute sample. The better the match, the higher the ranking score. Furthermore, since we expect that just one detail page is rarely completely representative for all detail pages of a provider, the search has to be repeated with more detail pages to get a more robust ranking result. The final ranking of XPath wrappers is a combined result derived from the individual sample rankings. A more detailed description of the approach is given in Section 2.

1.3. Related research

In their survey, Laender et al. [2] propose a taxonomy for web information extraction tools. Our approach can be categorized as *wrapper induction*: generation of extraction rules derived from a given set of training samples. This is fundamentally different from approaches where wrappers have to be defined by the user, even if this user is assisted by elaborate tools such as Lixto [3] or XWrap [4], because a wrapper may fail if the website changes its layout. Wrapper induction enables automatic adaptation to a new layout or even extracting from a previously unseen layout without human intervention.

Other approaches for wrapper induction distinguish themselves mainly in the rule language they use and the features derived from the training samples. Stalker [5] automatically learns extraction rules in their own Simple Landmark Grammar. SoftMealy [6] learns finite state transducers. DEByE [7] uses two kinds of patterns: (1) object extraction patterns determining the objects and (2) attribute value patterns determining the attribute values of these objects. In our approach, we use XPath as extraction rule language similar to Anton [8].

Regarding features, in our approach we use a combined score for how well the extracted attribute values from a number of detail pages match the training samples. Roadrunner [9] analogously relies on structure similarity between several pages. The probabilistic tree-edit model by Dalvi et al. [10], however, relies on different versions of the exact same page to estimate a *robustness* score against layout evolution. The availability of historic versions reduces its applicability in practice though. The visual placement of information is also a useful feature. Visual nesting is used by VisQI [11]. Placement also plays a role in Trieschnigg et al. [12]. Their objective is not to derive an XPath wrapper for detail pages, but for search result pages. The rich set of features including visual placement makes that only a single page suffices for their purpose, hence allows *unsupervised* on-the-fly automatic wrapper generation.

1.4. Outlook

Section 2 describes our approach in more detail. It discusses obtaining the input sample data, the handling of discrepancy problems between sample data and detail page data, the generation of flexible and intuitive XPaths, and the calculation and combination of the ranking scores. Section 3 presents our experiments with real-world data. The data set covers 7 attributes and 6 providers for book detail pages. Besides a general validation of our approach, the experiments also investigate how many input samples on average suffice for determining a suitable XPath wrapper.

2. Algorithms

This section describes in more detail the algorithms involved in our approach. Note that for simplicity the algorithms are formulated for a given target attribute and a given target provider. In practice the approach may be repeated for other attribute and provider combinations; intermediate results like generated XPaths for detail pages may be reused.

2.1. Input Data

As input for the approach we require an easily obtainable small data set of samples. More specifically, each input sample consists of an attribute sample and its associated object detail page.

Definition 1 *Let S be a set of samples (v_i, d_i) where v_i is an attribute sample for the target attribute a and d_i is the associated detail page at the target provider p. Let $D = \{d_i \mid (v_i, d_i) \in S\}$ be the set of sample detail pages from p.*

We believe that this sample data can be easily obtained for the following two reasons. First, there already exist many data sets containing attribute samples for a variety of attributes, e.g., lists of famous books. If the data set does not exist, we still believe that it is easier to create a small data set than to manually create and maintain the wrappers. Second, if combined with an automated approach for obtaining search results from a provider such as the one by Trieschnigg et al. [12], attribute samples can be used to automatically obtain the associated detail pages. In the use case of online bookstores, searching for the ISBN of a book is a good strategy to find associated detail pages. In general, unique identifiers like ISBN are good candidates for linking attribute samples to detail pages.

Since we use XPath as our extraction rule language, we also require the detail pages to be well-formed XML. Unfortunately many real world web pages are not completely valid XML documents, even when they claim to be XHTML. Luckily, many tools exist that take (X)HTML web pages as input, repair them and output a well-formed XML document. It is therefore assumed that the detail pages are available in well-formed XML.

2.2. Matching Score Calculation

In the example given in Figure 2, we are searching for the attribute 'Title' and one of the input attribute samples is 'Lord of the Flies'. Our approach simply traverses the whole XML tree of the associated detail page, including all HTML tag elements, attribute nodes and text nodes. It compares the text content of each node with the data sample 'Lord of the Flies'. With text content, we mean the XML *string value* of a node. For text nodes, the text content is the same as the node's content. For all other nodes the text content is created by concatenating the text contents of all child nodes or, if they don't have child nodes, the empty string.

One complication in this approach is the comparison: there may exist a data discrepancy between the attribute sample and the data on the detail page. For example, the letter case can be different and typing errors can exist:

Attribute sample *The Lord of the Rings. The Fellowship of the Ring Part 1*

String on page *The Fellowship Of The Ring: The Lord Of The Rings Part 1*

A simple string comparison would fail at detecting a match between both book titles. To overcome this problem, the algorithm is case insensitive and uses a string similarity measure to determine how well a node's text content matches the searched attribute sample. Two popular string similarity measures for this purpose are Jaro-Winkler and Levenshtein. We have chosen Jaro-Winkler, because earlier research has shown that it seems to be the preferred choice when it comes to record linkage [13]. The Jaro-Winkler similarity is transformed into a matching score between 1 (perfect match) and 0 (no match). Note that, within a detail page, an attribute sample may be found multiple times with varying scores.

Definition 2 *Let $d_j = \frac{1}{3}(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m})$ be the* Jaro distance *between strings s_1 and s_2 (or 0 if $m = 0$), where m is the number of* matching *characters and t the number of* transpositions *(matching but in a different order). Characters are only matching iff*

Algorithm 1 XPath generation for node n

```
procedure XPATH($n$)
    if $n$ is root then
        return ""
    end if

    if $n$ is element with tag $e$
      and has attribute "id"
      and its value is not in the ignore set then
        return "//e[@id='$id$']"
    end if

    $p \leftarrow$ XPath(Parent($n$))

    if $n$ is element with tag $e$
      and has attribute "class"
      and is first sibling with that class then
        return $p$ + "/e[@class='$class$']"
    end if

    if $n$ is element with tag $e$ then
        $p \leftarrow p$ + "/$e$"
    else if $n$ is attribute with name $a$ then
        $p \leftarrow p$ + "/@a"
    else if $n$ is text node then
        $p \leftarrow p$ + "/text()"
    end if

    if $n$ is $i$th sibling of that node type
      and $i > 1$ then
        $p \leftarrow p$ + "[$i$]"
    end if

    return $p$
end procedure
```

they are not father apart than $\lfloor \frac{1}{2} max(|s_1|, |s_2|) \rfloor - 1$*. Let* $d_w = d_j + lp(1 - d_j)$ *be the* Jaro-Winkler distance*, where* l *is the length of the common prefix and* p *a scaling factor (default* 0.1*).*

Definition 3 *The* individual score *for text content* t*,* $score(v_i, t) \in [0, 1]$*, is equal to the Jaro-Winkler string distance between* v_i *and* t*. For convenience,* $score(v_i, n) = score(v_i, t_n)$ *where* t_n *is the text content of node* n*.*

2.3. XPath generation

We have chosen the well known XPath standard as the rule language for addressing parts of the detail pages and extracting the wanted information. For each node in a detail page, we can automatically generate an XPath, although one has to be careful to make the XPath neither too precise nor too flexible. For example, we could construct XPaths like */html/body/p/h1[.='Lord of the Flies']*. While this XPath wrapper might work for one specific detail page, it is very inflexible, i.e., it does not generalize well to other detail pages. The other extreme is an XPath like *//a[3]* which selects the third anchor of a detail page. This XPath likely has a match on every detail page, but it is unlikely that it will select the same part of the template in every detail page. A compromise between those extremes is needed.

Our XPath generation algorithm (Algorithm 1) works as follows. Starting at a given XML node, the corresponding XPath is generated bottom up by visiting all ancestors one-by-one unless a stopping condition is met along the way. One stopping condition is if a visited node has an attribute node called *id*. In this case, an XPath like *//a[@id='booktitle']* is generated. Note that *id* attributes are not always suitable. In some cases, we encountered ids that were not unique although according to the HTML standard, there may be only one node with a certain id. Also in some cases, ids were generated strings that do not generalize well such as *contributorNameTriggerB000APSP20*. Such *id* attributes are ignored.

If a visited node does not have a unique id but has a class attribute and the node is the first sibling with this class, it is identified by its tag name and the class (e.g., */div[@class='bookInfo']*). Other tag nodes, text nodes and attribute nodes are handled with a position index. Only if the position index equals 1, we omit it from the XPath for brevity.

Note that we evaluate the XPath wrappers under a slightly different semantics, namely if an XPath evaluates to more than one node, we only take the first one. In other words, we assume a '[1]' predicate at the end of each XPath. The described XPath generation approach has been designed with this semantics in mind.

We believe that this approach for generating XPaths is a good compromise and matches the way how a human would intuitively write the XPaths. The following list shows some XPaths generated with this algorithm. They are short and flexible but, assuming a constant template, still able to unambiguously select the same intended part in all detail pages.

- *//input[@id='ASIN']/@value*
- *//span[@id='isbn13']*
- *//div[@id='bookmetadata']/strong[2]*
- *//li[@id='edition-details']/div[@class='about']/p/text()[3]*

2.4. Ranking

From the previous steps, we get two intermediate results for each input sample: (1) a list of nodes in the detail page and their matching scores and (2) XPaths that address the nodes. We arrive at a combined ranking of XPaths from these results in two steps.

First, each individual XPath gets as ranking score the already calculated Jaro-Winkler matching score of the first node it addresses. This is probably the most obvious and simple score candidate.

Definition 4 *The individual score score*(v_i, x_i) *for XPath* x_i *is equal to score*(v_i, n) *where* n *is the first matching node of* x_i *evaluated on* d_i.

Analogously, probably the most obvious and simple candidate for score combination is taking the average of the individual scores.

Definition 5 *The* final score *score*(a, x) *for an XPath* x *as a wrapper for attribute* a *is the average of the individual scores:* $score(a, x) = \frac{1}{|S|} \sum_{1 \leq i \leq |S|} score(v_i, x_i)$ *where* $x_i = x$.

Note that the combined ranking naturally deals with many ambiguous situations such as where an author "John Smith" also appears as editor: this will only produce a high score for the editor-XPath in one or two of the rankings in Figure 2, hence the author-XPath will surely outrank the editor-XPath in the final ranking.

A drawback of using the string similarity score as the ranking measure is that many XPaths are semantically equivalent and get the same score. For example, the following two XPaths likely result in the same text content.

- *//a[@id='booktitle']*
- *//a[@id='booktitle']/text()*

Therefore we decided to prune the ranking by discarding all XPaths from the final ranking that are just more specific versions of another XPath with the same score. In the mentioned example, only *//a[@id='booktitle']* would remain. This optimization is based on the intuition that less specific XPaths give shorter and more flexible wrappers.

3. Experiments

This section empirically assesses the performance and limits of the approach with real world data from the use case example of online bookstores.

Attributes	Providers
Title	Google Books
First author	Bookdepository
Publisher	Alibris
Publication year	Abebooks
Number of pages	Biblio
ISBN10	Amazon (also source of attribute samples)
ISBN13	

Table 1: Data set: 1120 input samples for each combination of provider and attribute

3.1. Data set

Our data set consists of attribute samples for 7 attributes of 1120 books and, for each book, 6 associated detail pages from different providers (See Table 1). This data set has been constructed as follows.

First a list of popular books was obtained from librarything.com [14]. Then, to obtain the attribute samples, each ISBN was automatically searched on amazon.com to find the associated detail page. The attribute samples were extracted with a manually created wrapper. Note that Amazon is the source for the attribute samples and also used as a provider. This is an interesting case where the attribute samples perfectly match with what is on the page. More detail pages were obtained from five other book websites, again automatically by searching for the ISBN number.

If one of the ISBNs from librarything.com had no corresponding detail page at one or more providers, then the sample was discarded. Only book samples which are known at all providers were retained to ensure that the book is indeed popular.

Ultimately the data set consisted of 6720 detail pages (1120 detail pages for each provider) and each detail page was associated to 7 attribute samples. The downloaded (X)HTML files were then converted to well-formed XML with the Java library HtmlCleaner [15]. Additionally we discarded all comments, text nodes containing only whitespace, script and style elements and namespace notations to simplify and speed up the experiments. No relevant attribute information was lost with this simplification.

The whole prototype was implemented in Java and used well known and tested libraries where possible. For example the Apache Lucene Spellchecker library [16] provided the important correct implementation of the Jaro-Winkler algorithm.

We also applied an additional optimization to the prototype to further speed up the experiment. For XPath generation, we apply a light pre-filter to the nodes: only those nodes are considered, where the length of the text content is less than or equal to three times the length of the attribute sample. Nodes that are excluded with this condition would have a low Jaro-Winkler score anyway, hence this optimization does not significantly affect the ranking.

3.2. Varying sample size

To get an indication of how large the input sample size needs to be to produce robust XPaths, we vary the sample size from 1 to 300 and randomly select subsets of those sizes from the aforementioned data set. For each such 'training' sample, we run our algorithm and determine the top-3 XPaths for each attribute. To assess the quality of these XPaths, we have randomly selected a 'test' sample with size 500.

The performance of the XPaths was determined by calculating the mean Jaro-Winkler string similarity between the text content of the extracted nodes and the attribute data associated with the test detail pages. To minimize the influence of the random selection, this procedure was repeated 50 times with different training and test samples.

3.3. Results

Figure 3(a) to Figure 3(e) show five of the 42 experimental results (one figure for each provider-attribute combination). The X-axis shows the input sample size. The Y-axis shows the mean similarity test score. Three lines are visible, one for each of the three best ranked XPath wrappers that resulted from the training. The lines also show the standard deviation of the mean test scores of the 50 experiment iterations. Note, that only sample sizes up to 30 are included because the mean performance and deviation did not significantly change with more input samples.

Figure 3(a) to Figure 3(c) show three results that are representative for most of the results. In Figure 3(a) there is clearly one XPath that achieves significantly better test scores than any other generated XPath. In Figure 3(b) there are apparently multiple XPaths that all work quite well. However, in both cases the best XPath does not reach the perfect test score of 1.0. This is mainly due to small differences between the input attribute data and detail page data. Figure 3(c) shows the biased case of Amazon where the input attribute data perfectly matches with the attribute data on the detail pages. As expected the best XPath actually reaches the test score of 1.0 in this case.

Figure 3(d) shows a representative case where the detail pages of a provider contained no data for a certain attribute. This is reflected in the test scores as the best XPath not exceeding 0.45 and the standard deviation remaining quite high even with high sample sizes. It still reaches a score of 0.45, though, because the best XPath extracts some 'size' attribute of an HTML element with integers in the same range as the target attribute 'number of pages'.

Figure 3(e) shows the typical case where the granularity of nodes in the detail page was not fine enough. This happens for example when the algorithm searches for the attribute sample '345' (number of pages) but the finest text content it can find is the string '345 pages'. One can argue that it still extracts the correct information. The maximum test score, however, is almost as low as for the case where no attribute information is found on the detail pages. Nevertheless, the standard deviation does become low with higher sample sizes.

What can be observed in all cases is that with low sample sizes there is a high standard deviation which usually decreases with more samples. This standard deviation results from the fact that with few samples the algorithm cannot be certain that the best XPath found generalizes to all detail pages. The algorithm can be lucky and actually find an XPath that works for all detail pages or it can be unlucky and find an XPath that applies only to the training samples. Furthermore, it can be observed, that in all cases 20 to 25 samples suffice to find the most suitable XPath.

4. Conclusions

This paper presents an approach for web information extraction that provides *automatic on-the-fly* wrapper creation. Our approach uses a small set of data samples for ranking XPaths on their suitability for extracting attribute information from the object detail pages of a website. We have shown with a real world use case of online bookstores that 20 to 25 samples suffice for finding the most suitable XPath. The set of candidate XPaths we generate is chosen to be precise enough to select a specific node but also flexible enough to generalize well to all detail pages. It appears that identifiers typically embedded in the generated HTML of the detail pages are particularly useful for keeping the set of candidate XPaths small. We used Jaro-Winkler similarity to address two problems: (1) small differences between attribute samples and detail page data, and (2) evaluating the quality of the chosen XPaths.

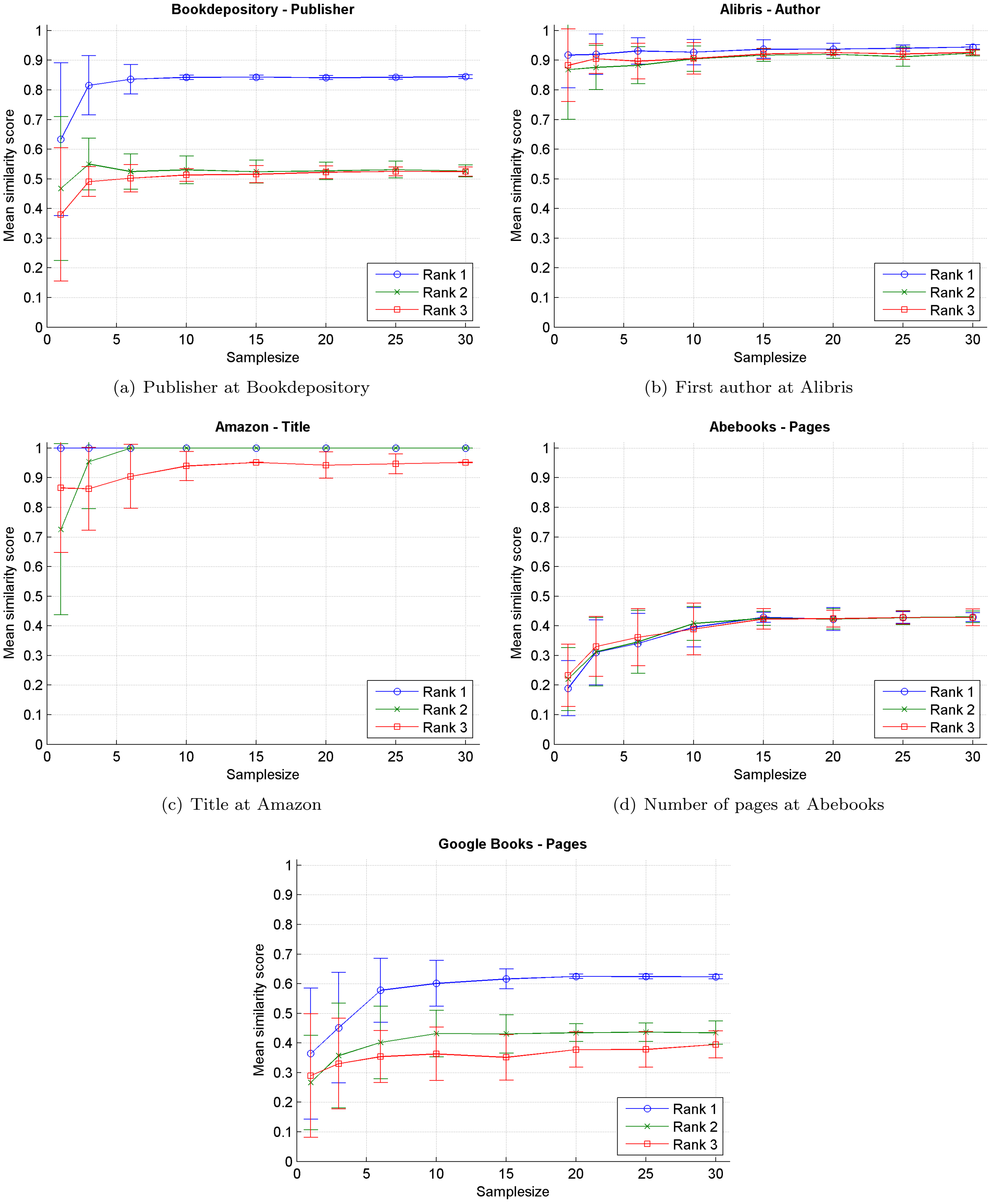
A remaining problem is making the distinction between the case that the attribute does not exist on the detail pages of a provider and the case that the granularity of the (X)HTML structure is not fine enough. A possible solution to the granularity problem may be the artificial insertion of nodes into the detail page.

Furthermore, we intend to investigate other XPath generation approaches. Our XPath ranking approach can easily consider more candidate XPaths. Generating more XPaths may improve the ability to find the most suitable XPath. Also, the use of a probabilistic database approach may be able to more robustly address ambiguous situations [17, 18].

Finally, we plan to investigate other string similarity measures and their exact influence on the ranking. A preliminary experiment showed that using Levenshtein resulted in a ranking of XPath wrappers that was similar to the ranking obtained with Jaro-Winkler. However, the ranking scores were generally lower, and it seems that Jaro-Winkler has a bias to finding more true positive matches while allowing more false positives than Levenshtein.

References

[1] XML Path Language (XPath) 2.0 (Second Edition). http://www.w3.org/TR/xpath20, December



(a) Publisher at Bookdepository

(b) First author at Alibris

(c) Title at Amazon

(d) Number of pages at Abebooks

(e) Number of pages at Google Books

Figure 3: Experimental results

2010.

[2] A.H.F. Laender, B.A. Ribeiro-Neto, A.S. Da Silva, and J.S. Teixeira. A brief survey of web data extraction tools. *ACM SIGMOD Record*, 31(2):84–93, June 2002.

[3] Robert Baumgartner, Georg Gottlob, and Marcus Herzog. Scalable web data extraction for online market intelligence. *Proceedings of the VLDB Endowment*, 2(2):1512–1523, August 2009. ISSN 2150-8097.

[4] L. Liu, C. Pu, and W. Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, pages 611–621. IEEE, 2000.

[5] Ion Muslea, Steve Minton, and Craig Knoblock. Stalker: Learning extraction rules for semistructured, web-based information sources. In *Proceedings of AAAI-98 Workshop on AI and Information Integration*, pages 74–81, 1998.

[6] Chun-Nan Hsu and Ming-Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Information Systems*, 23(8):521–538, 1998. ISSN 0306-4379. DOI `10.1016/S0306-4379(98)00027-1`.

[7] A.H.F. Laender, B. Ribeiro-Neto, and A.S. Da Silva. DEByE: data extraction by example. *Data & Knowledge Engineering*, 40(2):121–154, 2002.

[8] Tobias Anton. XPath-wrapper induction by generalizing tree traversal patterns. In *Lernen, Wissensentdeckung und Adaptivität (LWA) 2005, GI Workshops, Saarbrücken*, pages 126–133, 2005.

[9] V. Crescenzi, G. Mecca, P. Merialdo, et al. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 109–118, 2001.

[10] Nilesh Dalvi, Philip Bohannon, and Fei Sha. Robust web extraction: an approach based on a probabilistic tree-edit model. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 335–348, 2009. ISBN `978-1-60558-551-2`. DOI `10.1145/1559845.1559882`.

[11] Thomas Kabisch, Eduard C. Dragut, Clement Yu, and Ulf Leser. Deep web integration with VisQI. *Proceedings of the VLDB Endowment*, 3(1-2):1613–1616, September 2010. ISSN 2150-8097.

[12] R.B. Trieschnigg, K.T.T.E. Tjin-Kam-Jet, and D. Hiemstra. Ranking XPaths for extracting search result records. Technical Report TR-CTIT-12-08, Centre for Telematics and Information Technology, University of Twente, March 2012. ISSN 1381-3625.

[13] S.J. Grannis, J.M. Overhage, and C. McDonald. Real world performance of approximate string comparators for use in patient matching. *Medinfo*, 11(Pt 1):43–7, 2004.

[14] Book awards: 1001 books you must read before you die. `http://www.librarything.com/bookaward/1001+Books+You+Must+Read+Before+You+Die`, June 2012.

[15] Htmlcleaner. `http://htmlcleaner.sourceforge.net`, June 2012.

[16] Apache. Lucene core. `https://lucene.apache.org/core/`, May 2012.

[17] M. van Keulen. Managing uncertainty: The road towards better data interoperability. *IT - Information Technology*, 54(3):138–146, May 2012. ISSN 1611-2776. DOI `10.1524/itit.2012.0674`.

[18] M. van Keulen and A. de Keijzer. Qualitative effects of knowledge rules and user feedback in probabilistic data integration. *The VLDB Journal*, 18(5):1191–1217, October 2009. ISSN 1066-8888. DOI `10.1007/s00778-009-0156-z`.