Robust and Noise Resistant Wrapper Induction*

Tim Furche[1] and Jinsong Guo[1] and Sebastian Maneth[2] and Christian Schallhart[1]

[1]University of Oxford
firstname.lastname@cs.ox.ac.uk

[2]University of Edinburgh
smaneth@inf.ed.ac.uk

ABSTRACT

Wrapper induction is the problem of automatically inferring a query from annotated web pages of the same template. This query should not only select the annotated content accurately but also other content following the same template. Beyond *accurately* matching the template, we consider two additional requirements: **(1)** wrappers should be *robust* against a large class of changes to the web pages, and **(2)** the induction process should be *noise resistant*, i.e., tolerate slightly erroneous (e.g., machine generated) samples. Key to our approach is a query language that is powerful enough to permit accurate selection, but limited enough to force noisy samples to be generalized into wrappers that select the likely intended items. We introduce such a language as subset of XPATH and show that even for such a restricted language, inducing optimal queries according to a suitable scoring is infeasible. Nevertheless, our wrapper induction framework infers highly robust and noise resistant queries. We evaluate the queries on snapshots from web pages that change over time as provided by the Internet Archive, and show that the induced queries are as robust as the human-made queries. The queries often survive hundreds sometimes thousands of days, with many changes to the relative position of the selected nodes (including changes on template level). This is due to the few and discriminative anchor (intermediately selected) nodes of the generated queries. The queries are highly resistant against positive noise (up to 50%) and negative noise (up to 20%).

1. INTRODUCTION

Web wrappers are programs that extract information from web pages. For instance a wrapper may extract the names of movie directors from movie listings such as IMDB. Wrappers have always played a significant role in accessing the web as data—whether to extract competitors' prices for competitive intelligence (see [10] for a recent survey) or as bridge between traditional web pages and linked data [19]. Wrapper induction is then the process of inferring a wrapper from a given sample of annotated data which follows a certain template. A good wrapper is expected to select the same intended information also from other pages of the same template. For instance, given a sample with "Martin Scorsese" (more specifically, its enclosing `span`-node) annotated, a wrapper that searches for this particular string would be far too specific in most cases: it works only for movies of this single director. A wrapper is **(1)** *accurate* if it infers the full set of intended information from the given examples. This is a notion common to most wrapper induction approaches. Our wrappers should additionally be **(2)** *robust* against structural changes of the web page over time, and—different from previous approaches—**(3)** the induction process should be *resistant to noise* in the annotations. E.g. if all but one of several movie directors are annotated on a page, then the induced wrapper should select all directors, i.e., treating the missing director as noise.

Previous wrapper induction approaches [18, 1, 16] have focused on the *supervised* setting where a human annotator provides a perfect set of annotations. We consider a very different setting for wrapper induction, motivated by a recent rise in automated data extraction [15, 5, 13] and wrapper maintenance approaches [22]. Samples are no longer provided by humans but automatically by entity recognizers [5, 13] or by finding known instances in new [15] or modified [22] web pages. For both cases, only as much annotations may be provided for any single page or site as there is overlap between the page's items and the recognized entities or known instances. Different from the supervised setting, automatically generated samples can be *noisy*—some annotations might be missing and others might be incorrectly placed.

Thus, our wrapper induction method considers samples that may be both noisy and minimal, allowing wrappers to be induced from annotations for a single page and from as little as a single sample. It achieves this primarily through the careful choice of an *expressive subset* of XPATH, that is sufficient to express most wrappers, but limited enough to force noisy annotations to be generalized into wrappers that select the likely intended items. The proposed method achieves remarkable robustness. It does so in a very different fashion than previous approaches—by optimizing for *short, selective wrappers* that use HTML's *semantic markup* for selection where possible: our approach favors **(1)** *short* expressions with a small number of steps over multi-step expressions. It also prefers **(2)** *selective* expressions that drill down quickly into a small portion of the HTML page over expressions that are unselective (though possibly shorter). Selective expressions are less likely to be affected by changes in unrelated parts of the document. It is also heavily biased towards exploiting hints about the **(3)** *se-*

*Corresponding author: Jinsong Guo. The research leading to these results has received funding from the European Research Council under the ERC grant DIADEM, no. 246858 and ExtraLytics, no. 641222, as well as from the EPSRC programme grant VADA, no. EP/M025268/1. The current affiliation of Christian Schallhart is Google London (email address: christian.schallhart@gmail.com).

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2882903.2915214

mantic role of elements in the template. In HTML, these are typically expressed in `id` or `class` attributes and often used for styling and scripting. HTML5's Microdata adds further semantic attributes such as `itemprop`. Many templates also provide static labels, either visible or in form of `title` tooltips. These criteria are designed to mimic human-created robust XPATH expressions.

As an example, consider the following wrapper, extracting (`span`-elements of) directors from IMDB movie pages:

```
descendant::div[starts-with(.,"Director:")][1]/
    descendant::span
```

This XPATH query selects the first `div`-element with text-value of the form "Director: ...". Starting from this `div`-element, the query selects all descendant `span`-elements. There is only one such span-element, and this element contains precisely the director names of the movie. Thus, the above is an accurate wrapper for extracting director names. However, this query is *not* robust against changes to the web page: **(1)** Imagine more `span` elements (containing non-director information) are inserted under the correct div-element. The wrapper would wrongly select all of them. **(2)** Imagine that more `div`-elements (without director information) are inserted *before* the `div`-element containing the director information. The wrapper would select the wrong `div`-element and return its contained `span` elements, if any. Our approach does *not* attempt to build an accurate model of changes done to a specific web site or class of web sites. Instead, we aim to model heuristics for building robust wrappers on any type of web site.

What then, is a robust wrapper for this example?

```
descendant::div[starts-with(.,"Director:")]/
    descendant::span[@itemprop="name"]
```

This wrapper q_{director} is the most robust one in our setting. It follows the heuristic outlined above for short, selective wrappers that prefer semantic features in their predicates. It increases the selectiveness of the `span` step by using a predicate on a semantic attribute (`itemprop`) and shortens the `div` step by removing the positional predicate that is no longer needed (because of the `itemprop`-attribute). We often refer to the intermediate nodes selected by an XPATH expression as "anchors". Figure 1 shows a fragment of the tree structure of an IMDB movie page with "Martin Scorsese" as director; the red shaded nodes are anchors of q_{director}.

Contributions. We present a novel approach for inducing wrappers under the combined requirements of robustness and noise resistance.

(1) We define *directed XPath with sideways checks* (for short, dsXPath). dsXPath is powerful enough to capture a large class of wrappers and is weak enough to support the generalization to correct queries from noisy samples.

(2) We define a natural framework for computing the *robustness* of a query. The essential feature of this framework is its composability: the robustness score of a composed query can be computed from the scores of its constituents.

(3) We show that computing the top-k most robust queries is intractable, hence there is no hope for an exact solution.

(4) We devise a recursive bottom-up scheme for estimating the top-k most robust dsXPath queries for a given set of samples.

(5) We experimentally evaluate our approach using old versions of popular pages from the Internet Achieve.

Noise Resistance. To deal with noisy samples, we restrict to a particularly small XPATH fragment. This fragment is *not* selection complete, i.e., there are annotated documents for which no query in the fragment selects the annotated positions accurately. For instance, our fragment cannot select all but some particular elements of a list. Most notably, the fragment misses important features such as negation and union. Interestingly, while trying to keep the XPATH fragment as restricted and small as possible, we found that certain axes are crucially required in order to achieve full accuracy for all our samples. Besides the four base axis of XPATH (**`child`**, **`parent`**, **`descendant`**, and **`ancestor`**), we found that the two "sideways" axes **`following-sibling`** and **`preceding-sibling`** are essential for our wrappers. For instance, often we want to select all elements of a list appearing *after* a specific determining element.

Experiments. Our evaluation validates three claims: **(1)** The approach generates robust wrappers, that remain effective as long as the relevant data is present, often for hundreds of days. **(2)** The generated wrappers have comparable or higher robustness than human-created ones. **(3)** The approach is able to generate such wrappers even in presence of significant amount of noise in the annotations. We selected over 100 popular web pages from more than 50 different sites. We track the evolution of the pages over *six years* in the Internet Archive in 20 day intervals. For each of these over $> 10k$ "page versions", we evaluate the human crafted wrapper, the wrapper produced by our system, and a simple baseline wrapper.

Related Work

This paper introduces a novel wrapper induction approach tailored to the emerging setting of automatic, rather than human provided annotations. Wrapper induction is the process of inducing an extraction program, often one or more XPATH or similar expressions over the HTML tag tree, from a set of examples.

Supervised Induction. Initially, wrapper induction approaches [18, 1] have been **supervised** where annotations are provided by humans (see [12]). The supervised setting has seen a flurry of industrial tools such as Lixto [1] or Mozenda, http://mozenda.com/. In fact, a simple incarnation of these inducers has become an essential tool for web developers, to inspect web sites and find relevant nodes. Firebug, as well as the developer tools provided with Chrome, Firefox, and Safari, allow to induce XPATH expressions from single samples. For neither of these approaches has robustness of the expressions been a primary concern. For the latter, e.g., the induced expressions are often one-off means for debugging and thus optimized for simplicity. In fact, most of these tools provide expressions close to the canonical path (see Section 2), at most exploiting unique `id` attributes if present. These expressions are typically not *robust* per se, as noted in [8, 9]. Furthermore, all of these tools assume a perfect input and *only* return expressions that cover all and only the given examples with no *noise*. Some of the more advanced systems, such as [1], can infer a wrapper for a list of template items from only a few examples of that list and thus deal with a limited form of negative noise, but even in these cases positive noise is not considered.

That lack of robustness of earlier supervised approaches has been noted in [8, 9, 6] and let to two methods that partially address that issue: **(1)** The first [8, 9] aims to find a more robust wrapper the selects the exact same data items as the given one. However, the approach is limited by the choice of wrapper language, a fairly narrow subset of XPATH. Most importantly, it does not address attributes and attribute values at all, yet the choice of attributes and values is among the most crucial choices for robust wrappers (see Section 6). **(2)** The second is focused on learning probabilistic models of the evolution of web pages over time to rank XPATH expressions by their robustness [6]. This allows to tailor robustness to the specific change behavior of a web site. However, the used XPATH fragment is strictly weaker than the one used here (only **`child`** and

`descendant` axis, at most one predicate per step, only equality predicates). Furthermore, our results demonstrate that expensive, site-specific training of change models is not required to achieve long term robustness. This work is extended to the notion of an optimal wrapper [21] with respect to a probabilistic and a adversarial (worst-case) change model.

Automatic Induction. Supervised wrapper induction requires significant supervision and human effort. Thus lately a number of automated approaches to wrapper induction have been suggested. Generally, these approaches are focused on inducing wrappers for a list of search results or single-entity (e.g., product) pages following the same template structure. They fall into three categories depending on the primary means to identify the elements of such a list: **(1)** *Pure template discovery:* The earliest approaches attempt to discover the template structure purely from the visual and structural similarity of the items of the template. As for wrapper induction, there is a notion of robustness for template discovery (see, e.g., [20]). However, robustness here measures how well a certain method is able to deal with a wide variety of sites and application domains. **(2)** *Entity extraction-based:* These approaches [23, 5, 7, 13] use domain-specific entity extractors to annotate the relevant items on the page. They employ fairly standard XPATH wrapper induction approaches that can not deal with positive noise and only with limited negative noise if any. Thus, their main contribution and difference lies in how they are reconciling the annotations in a pre-processing step before the XPATH induction: ODE [23] uses annotations only for post-processing to filter and label the item lists identified through template discovery, [5] considers multiple subsets of the annotations, [13] combines the annotation with, among others, a novel template discovery component that filters the given annotations to maximize regularity of the detected template items. In both cases, annotation noise is eliminated in elaborate and costly post-processing steps, which may be not be necessary at all or could be simplified if combined with the induction method suggested here. **(3)** *Redundancy-based:* Finally, methods such as [4, 3, 15, 2] exploit overlapping sources in the same domain. They consider many candidate item lists and select ones that maximize cross-domain instance-level redundancy. In many ways, the techniques for inducing a robust wrapper discussed in this paper are orthogonal to these approaches, as also pointed out in [5]. Most of these approaches [5, 7, 13] use XPATH induction once the relevant list items are identified, a few use regular expression-like patterns [23], or queries against some form of visual representation of a page [20]. For none of the automatic induction approaches robustness of the induced wrapper is a primary concern, though [5] employs, among others, the induction method from [6]. In most cases, our induction method could be integrated with these systems, thus yielding more robust wrappers, as shown in Section 6.

2. PRELIMINARIES

We assume the reader to be familiar with the basics of XPATH 1.0 and use HTML and XML syntax freely. An (HTML or XML) *document* D is a text containing markup and attribute definitions. The latter two give rise to a tree structure, containing element nodes, attribute nodes, and text nodes. An example of a document's tree structure is shown in Figure 1. Element nodes are depicted as ellipses. The attributes of an element give rise to attribute nodes. These attribute nodes are depicted as boxes, which are connected by dotted lines to their element nodes. The dotted line is labeled by the name of the attribute (prepended by the `@`-symbol), while the box contains the attribute's value. For instance, the document fragment `<h4 @class="inline">Director:</h4>` gives rise to the circled `h4`-node in the lower left of the figure, and to the `inline`-labeled attribute box connected to it.

Wrappers. Let D be a document. XPATH queries are evaluated relative to a given node u of D. We denote by $q_D(u)$ the set of target nodes of q when evaluated relative to u in D, and simply write $q(u)$ if D is clear from the context. A *wrapper* is an XPATH expression q. These expression will be evaluated relative to the root of D, thus, $q_D(r)$ is the result set of the wrapper expression on D, where r is the root node of D. We denote $q_D(r)$ also by $q(D)$. *Wrapper induction* is an algorithm I that given a document D and a set of nodes V of D, returns a wrapper $q = I(D,V)$. We say a wrapper q "matches" a target node n from a context u if $n \in q_D(u)$, omitting u if $u = r$.

Canonical Path. Let D be a document and let u be an element or text node of D. The *canonical path of* u, denoted $\mathsf{canon}(u)$, is an XPATH query defined recursively as follows. If u is the root node then $\mathsf{canon}(u) = /$. Otherwise, let $p = \mathsf{canon}(v)$ where v is the (element) parent node of u, $k \geq 1$ such that u is the k-th child of v, and t is a node test for u (i.e., `text()` if u is a text node, or *name* if u is a *name*-labeled element node). Then, $\mathsf{canon}(u) = p/t[k]$.

For Figure 1, the canonical path of the director node on an IMDB director page (the red `span` element) is:

```
/html[1]/body[1]/ ... /div[4]/a[1]/span[1]
```

The dashed line in Figure 1 and the dots in the expression above refer to a part of the path that is not shown (consisting of eleven XPATH steps, seven of which are `div`s).

Let $\mathcal{D} = \langle D_1, \ldots, D_n \rangle$ be a sequence of documents all containing a given node v (typically a sequence of versions of a single page, e.g., snapshots of an IMDB director page). Then, we say that there are k *c-changes* in $\mathcal{D}$ w.r.t. v, if $k = |\{D_{i+1} : \{v\} \neq \mathsf{canon}(v)(D_i)\}|$. The number of c-changes is a measure for changes in $\mathcal{D}$ that affect the path from v to the document root. Such changes are far more likely to affect any query selecting v than changes in regions of the document far from v.

Robustness. Let q be a wrapper and let D and D' be documents. The query q is *robust* (*for D and D'*), if there exists a bijection π between $q(D)$ and $q(D')$ so that for all nodes v in $q(D)$: $D/v = D'/\pi(v)$. Here, D/v denotes the (abstract, nodeId-free) subtree of D rooted at node v. Note that since $q(D)$ is a set (and not a sequence), our robustness definition is order independent.

Noise Resistance. Let D be a document and V, V' be sets of nodes of D such that V' is obtained from V by adding and deleting nodes. A wrapper induction is *noise resistant* (*for V, V' and D*), if given D and V' it returns the same query q as for input D and V. A wrapper induction I is *k%-robust*, if for k% of all pairs (D, D') and sets of nodes V from D, the wrapper $I(D,V)$ is robust. A wrapper induction I is *k%-noise resistant*, if for k% of all documents D and sets of nodes V, V' of D, the wrapper $I(D,V)$ is noise resistant.

Precision, Recall, and F-Score. Imagine a set B approximating another set A. The elements in $B \cap A$ are called *true positives* and t^+ denotes their number $|B \cap A|$. The elements in $B - A$ are called *false positives* and we define $f^+ = |B - A|$. The elements in $A - B$ are called *false negatives* and we define $f^- = |A - B|$. The *precision* of B with respect to A is calculated as $\mathsf{prec}(A,B) = t^+/(t^+ + f^+)$. The *recall* of B with respect to A is $\mathsf{rec}(A,B) = t^+/(t^+ + f^-)$. The *F-score* $\mathsf{F}_\beta(A,B)$ measures the accuracy biased toward precision ($\beta < 1$) or recall ($\beta > 1$): $\mathsf{F}_\beta(A,B) = \frac{(1+\beta^2)\mathsf{prec}(A,B)\mathsf{rec}(A,B)}{\beta^2\mathsf{prec}(A,B)+\mathsf{rec}(A,B)}$.

3. XPATH FRAGMENT DSXPATH

In this section we define our XPath fragment, called *directed XPath with sideways checks* (for short, dsXPath). The idea of

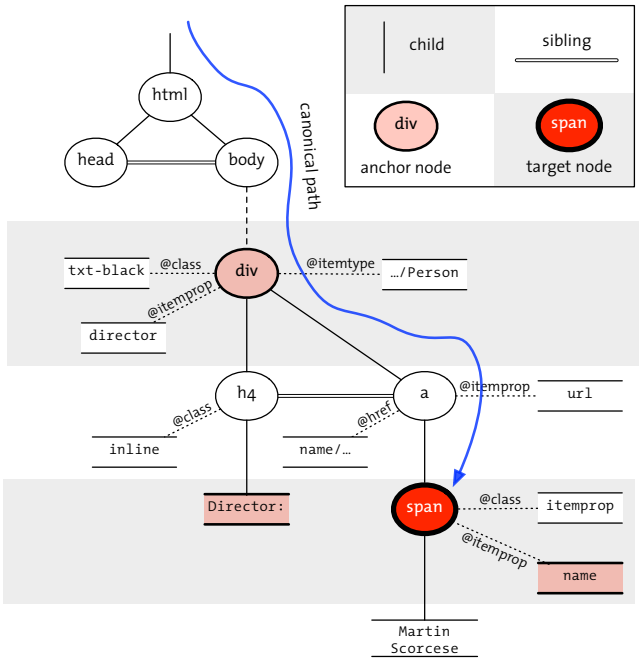

Figure 1: Tree structure of an IMDB movie page

dsXPath is to be **(1)** *general enough* to cover all samples with 100% precision and recall and **(2)** *restricted enough* as to enforce noise resistant queries. In Section 4, we show how the careful choice of the language fragment together with a straightforward, but flexible scoring of expressions yields a noise resistant, yet accurate and robust wrapper induction.

Query Syntax. The syntax of dsXPath queries is presented in Figure 2. A query consists of one or more steps. Each step consists of an axis, a nodetest, and an arbitrary number of predicates. The axis is any of XPATH's navigational axes (including the attribute axis), except the following and preceding axes. The nodetest is any of XPATH's nodetests. The nonterminals ⟨*TagName*⟩ and ⟨*AttrName*⟩ in Figure 2 refer to the strings allowed as element and attribute names. A predicate is either positional (i.e., consisting of a number or of `last()` minus a number), the test for the existence of an attribute, or one of four possible Boolean functions on strings. The second argument to a Boolean string-function is an arbitrary string of characters (indicated by the nonterminal ⟨*String*⟩). The first argument to a Boolean string-function is either an attribute selection (given by the attribute axis followed by a name) or a text access. The only text access that we consider is the fixed expression `normalize-space(.)`. This expression selects and normalizes (i.e., removes extra whitespace) the text-value of the current node. E.g., applied to the `div`-node in Figure 1 we obtain "Director: Martin Scorsese". For brevity, we write only `.` instead of `normalize-space(.)` and abbreviate **`attribute`** with `@` as in XPATH.

⟨*Query*⟩ ::= ⟨*Step*⟩ ('/' ⟨*Step*⟩)*
⟨*Step*⟩ ::= ⟨*Axis*⟩ '::' ⟨*Nodetest*⟩ ('[' ⟨*Predicate*⟩ ']')*
⟨*Axis*⟩ ::= **`child`** | **`attribute`** | **`descendant`** | **`following-sibling`** | **`parent`** | **`ancestor`** | **`preceding-sibling`**
⟨*Nodetest*⟩ ::= '*' | `node()` | `text()` | ⟨*TagName*⟩
⟨*Predicate*⟩ ::= ⟨*Int*⟩ | `last()` '-' ⟨*Int*⟩ | **`attribute`** '::' ⟨*AttrName*⟩ | ⟨*Function*⟩ '(' ⟨*Content*⟩ ',' ⟨*String*⟩ ')'
⟨*Function*⟩ ::= `equals` | `contains` | `starts-with` | `ends-with`
⟨*Content*⟩ ::= **`attribute`** '::' ⟨*AttrName*⟩ | `normalize-space(.)`

Figure 2: Syntax of our XPATH Fragment

We explain the semantics of XPATH using examples, see [14] for details. We implicitly assume a given document D. A query q is evaluated starting at a given node u of D. The result $q(u)$ is a set of nodes of D. Consider the following query q.

```
descendant::div[starts-with(.,"Director:")]/
    descendant::span[(@class="itemprop")]
```

The evaluation of q on the root of a document D goes step-wise through the XPath query: first all div-labeled descendants of the root node are selected, and of those, the ones with text-value starting with "Director:". Finally, we select from these nodes the span-labeled descendants with class-attribute equal to "itemprop" and return them as result.

The above query uses two anchor nodes to characterize the target `span`. An *anchor node of* q is a node of D that is selected during the evaluation of $q(r)$ on D. We do not give a formal definition but note that red-marked nodes in Figure 1 are the anchors of this query.

dsXPath Queries. We now define *directed* XPATH *queries with sideways checks* (dsXPath Queries) as a restriction of the syntactical fragment given in Figure 2. We choose this fragment for two reasons, namely, to obtain a feasible search space and to foster generalization during induction to avoid overfitting of noisy query samples. Such queries are either *one-* or *two-directional*. Intuitively, a one-directional query does not change its up/down direction, i.e., it strictly traverses the tree either top-down or bottom-up (but not mixtures thereof). However, in a one-directional query we do additionally allow "sideways checks". A sideways check is a subquery consisting of the following- and preceding-sibling axes. A two-directional query is the concatenation of two one-directional queries. Given a query $q = a_1 :: t_1P_1/a_2 :: t_2P_2/\ldots/a_n :: t_nP_n$ with axes a_i, node tests t_i, and predicates P_i, we define $\mathsf{axes}(p) = a_1 \ldots a_{n-1}$ if a_n = attribute and $\mathsf{axes}(p) = a_1 \ldots a_n$ otherwise. The query q is *one-directional,* if $\mathsf{axes}(q)$ matches one of the following regular expressions with **`following-sibling`*** | **`preceding-sibling`*** = ⟨*sideways*⟩:
(1) ((**`parent`** | **`ancestor`**) ⟨*sideways*⟩)* or
(2) ((**`child`** | **`descendant`**) ⟨*sideways*⟩)*.

Finally, for a given sequence of documents $\mathcal{D} = (D_1, \ldots, D_n)$, and a given dsXPath query q, we say that q is *plausible*, if all constants appearing in the predicates of q also appear in the considered document. Recall that the text-value of a document is the concatenation of all texts in the document. The query q is plausible, if it is generated by the EBNF in Figure 2 in such a way that **(1)** each string is either a substring of the text-value of a document in $\mathcal{D}$, or is a substring of an attribute value of a document in $\mathcal{D}$, and **(2)** each integer produced by the nonterminal ⟨*Int*⟩ is not larger than the number of nodes of any document in $\mathcal{D}$. From now on, we only consider plausible dsXPath queries.

4. WRAPPER INDUCTION

We first introduce the query model and types of induced queries. Then our ranking is defined and the query induction problem is shown to be infeasible (**NP**-hard).

Query Model. Given a document D, a *query sample* is a pair $\langle u, V\rangle$, where u is a node of D and V is a non-empty set of nodes of D. Our query induction induce takes a sequence of query samples $S = (\langle u_1, V_1\rangle, \ldots, \langle u_n, V_n\rangle)$ over the documents $\mathcal{D} = (D_1, \ldots, D_n)$ (possibly containing duplicates) as input and returns a ranked set of *query instances* $Q = \mathsf{induce}(S)$. A query instance $q = \langle p, t^+, f^+, f^-\rangle$ consists of an XPATH expression $p = \mathsf{query}(q)$ and the corresponding numbers of true positives, false positives, and false negatives, i.e., the numbers $t^+ = \Sigma_{i=1}^n |p(u_i) \cap V_i|$, $f^+ = \Sigma_{i=1}^n |p(u_i) - V_i|$, and $f^- = \Sigma_{i=1}^n |V_i - p(u_i)|$. Given q, we write $t^+(q)$, $f^+(q)$, and $f^-(q)$ for the individual values of q. We also write $q(u)$ in lieu of $p(u)$ with $p = \mathsf{query}(q)$. Ideally, each XPATH expression $p = \mathsf{query}(q)$ for $q \in \mathsf{induce}(S)$ should lead from u_i to V_i, i.e., $p(u_i) = V_i$ (evaluated in D_i) for all $1 \le i \le n$. Note however that a fully accurate expression p is not always possible or – in case of noisy input – not even desired.

Ranking. In order to deal with noisy additional annotations, we choose $\beta = 0.5$ for the F-score. We rank the query instances q by maximizing the F-score $\mathsf{F}_{0.5}(q)$ and minimizing the robustness score $\mathsf{score}(q)$. More precisely, we define the order $<$ with $q < q'$ iff (1) $\mathsf{F}_{0.5}(q) > \mathsf{F}_{0.5}(q')$ or (2) $\mathsf{F}_{0.5}(q) = \mathsf{F}_{0.5}(q')$ and $\mathsf{score}(q) < \mathsf{score}(q')$. The *robustness score* $\mathsf{score}(q)$ is a positive number computed recursively from the structure of the query expression $\mathsf{query}(q)$. The *smaller* the score, the more favorable the query. The robustness score of an XPATH expression is computed as the sum of the scores of its steps, because we favor shorter queries over longer ones; we call this *plus-composability*. The scores of steps are multiplied by a power of a *decay-factor* δ. This allows to favor cheaper steps towards the end (beginning), i.e., closer to (further from) the target node, by choosing $\delta \le 1$ ($\delta \ge 1$). We define $\mathsf{score}(a_1 :: t_1 P_1 / \cdots / a_n :: t_n P_n) = \sum_{i=1}^n \mathsf{score}(a_i :: t_i P_i) \cdot \delta^{i-1}$. where for $i \in \{1, \ldots, n\}$, a_i is an axis, t_i is a node test, and P_i is a sequence of predicates.

For each axis a (such as **descendant** of **child**) we fix a constant score s_a. Similarly, for each node test t we fix a constant s_t. Note, that different tags can thus be scored differently, e.g., `div`'s differently from `script` tags, though in most cases a default value $c_{default}$ is used. The score for a single step $a :: tP$ is now computed as the sum of the scores for axis, node test, and predicates $P = p_1 \ldots p_m$: $\mathsf{score}(a :: t\, p_1 \cdots p_m) = s_a + s_t + \sum_{j=1}^m \mathsf{score}(p_j)$.

The score for a predicate p is computed as follows. If p is a positional predicate, i.e., of the form $p = [n]$ or $p = [\texttt{last()} - n]$ where n is a positive integer, then the score for p is $\mathsf{score}(p) = c_{\text{pos}} \cdot n$, where c_{pos} is a fixed constant.

If p is of the form `[`f`(`**attribute**`::`a`, `w`)]` or `[`**attribute**`::`a`]` (in the latter case we set $\mathsf{length}(w)$ and s_f to zero) where a is an attribute name, f is a function (such as "equals" or "contains"), and w is a string, then the score for p is the sum of a fixed score s_f, a *no-function-penalty* constant y, a fixed score s_a, and the product of the fixed length factor c_f and the length of the string w: $\mathsf{score}(p) = s_f + y + s_a + c_f \cdot \mathsf{length}(w)$, where y has a non-zero value only if $p =$`[`**attribute**`::`a`]`. For a predicate with text access of the form $p =$`[`f`(.,`w`)]` where f is a function (such as "equals" or "contains") and w is a string, we define $\mathsf{score}(p) = s_f + s_{\text{text}} + c_f \cdot \mathsf{length}(w)$, where s_{text} is a fixed score. If q contains no predicate, we add to $\mathsf{score}(q)$ a *no-predicate-penalty*. The two penalty scores *no-function-penalty* and *no-predicate-penalty* bias the scoring towards queries with predicates that are likely more selective.

Query Induction Problem. We define the query induction problem and prove that it is infeasible. The query induction problem is the optimization problem of finding a selection of the best ranked query instances among the plausible directed expressions in dsXPath, where optimality is in terms of the ranking given in the previous section. For a natural number K, a top-K set is a set of ranked query instances with the highest achievable scores; there might be several such top-K sets achieving the same scores.

DEFINITION 1 (QUERY INDUCTION PROBLEM). *Given a sequence $S = (\langle u_1, V_1\rangle, \ldots, \langle u_n, V_n\rangle)$ of query samples over implicit documents $\mathcal{D} = (D_i, \ldots, D_n)$ and a constant K, the* query induction problem *is to find a top-K set of plausible query instances.*

THEOREM 1. *The query induction problem is* **NP***-hard, even with query samples of the form $\langle u, \{v\}\rangle$, requiring an optimal query instance leading from u to v.*

The theorem can be reduced to Minimum Set Cover.

5. DSXPATH INDUCTION ALGORITHM

In this section our query induction algorithms are explained, first for single-target query samples, and then for multiple-target query samples. We always work in the context of a given number K, and compute sequences of "K-best" query instances (ordered according to our ranking). The resulting algorithm has polynomial runtime since it executes a polynomial number of XPATH queries of polynomial size over the analyzed document. We define the *set B of base axes* as $B := \{$**child**, **parent**, **following-sibling**, **preceding-sibling** $\}$, and define **child**.transitive = **descendant** and **parent**.transitive = **ancestor**. Further, we set α.transitive $= \alpha$ for $\alpha \in \{$**following-sibling**, **preceding-sibling**$\}$. For an axis β, we say that node v is β*-reachable* from u if and only if $v \in q(u)$ for the query $q = \beta :: *$.

Single-Target Query Samples. Let D be a document and u a node of D. Here we consider the restricted case of one target node, i.e., $V = \{v\}$ where v is in D. Our algorithms are constructed in such a way that v is guaranteed to be reachable from u via the transitive axis of one of our base axes. The most interesting case of our induction algorithm, is for the case that v is a descendant of u, i.e., for the **child** base axis. The cases for the other base axes follow from this case in a straightforward way. Let us thus discuss now the case that v is a descendant of u. We denote the sequence of nodes on the path from u to v by $\mathsf{spine}(u, v)$, and refer to the nodes in $\mathsf{spine}(u, v)$ as *possible anchors*. To compute the best-K query instances leading from u to v, we incrementally compute the best-K instances $best(n)$ for each node n in $\mathsf{spine}(u, v)$. Initially, we set $best(v)[1] = \langle \varepsilon, 1, 0, 0\rangle$ and $best(v)[k] = \langle \bot, 0, 0, 0\rangle$ for $2 \le k \le K$, The "empty query" ε occurs only as intermediate result; When applied to $best(v)$, it selects only the node v itself. By $\bot$ we denote the "fail query" which always returns $\emptyset$. We now recursively compute for each node $n \in \mathsf{spine}(v, u) - \{v\}$, the superset $cand(n) \supseteq best(n)$ of *candidate instances*, by using the already computed best-K sets $best(t)$ for nodes t preceding n in the sequence $\mathsf{spine}(v, u)$. From this superset, we assign the best-K instances to $best(n)$ and proceed along the spine until we have computed $best(u)$, which is returned as result. The superset $cand(n)$ is computed as $cand(n) = \bigcup_{t \in \mathsf{spine}(v,n) \setminus \{n\}} \mathsf{stepPattern}(n, t, \mathsf{axis}, K) \times best(t)$ as shown in Algorithm 2. Here, axis is the unique base axis in B such that t is axis.transitive reachable from n, i.e., $\mathsf{axis} =$**child** for the case discussed here. The function $\mathsf{stepPattern}(n, t, \mathsf{axis}, K)$ computes the best-K query instances that match t from n along axis (possibly involving "sideways detours" but no intermediate anchors), and $\times$ yields the concatenated instances of the cross-product of the two instance sets. We next explain function $\mathsf{stepPattern}(n, t, \mathsf{axis}, K)$ performing induction along the spine.

Spine Step Induction. The function stepPattern(n,t,axis,K) shown in Algorithm 1 generates the available spine patterns to match spine node t while moving along axis (or its transitive closure) starting at context node u; only the case of the child-axis is shown. Note that in the return specification of the algorithm, axis.reverse refers to the reverse of an axis, i.e., **child**.reverse = **parent** (and vice versa), **descendant**.reverse = **ancestor** (and vice versa), and **following-sibling**.reverse = **preceding-sibling** (and vice versa). This algorithm works as follows. We first compute for axis = child recursively through inducePath (Lines 2–5). Note, child is the *unique* axis for which we ever produce sideways checks in our algorithms! This explains that in the recursive case (for the child axis), we iterate over all siblings s of t (Line 2), compute the node patterns for s (Line 3) and the best-K paths from s to t recursively (Line 4). Note that each such path consists of exactly one XPATH step that uses either the **following-sibling** or **preceding-sibling** axis (depending on the position of s with respect to t). Then we add all possible concatenations between node patterns and paths (Line 5). Thus, each query in P starts with a node test. As all recursive calls to inducePath generate a path along the sibling axes, the subsequent calls to stepPattern will be handled non-recursively (because of Line 1).

Algorithm 1: Step Induction stepPattern(n,t,axis,K)

input : node t is axis.transitive-reachable from node n
returns : path set P such that $\forall p \in P$ and all nodes x if x is axis.reverse.transitive-reachable from t, then $\{t\} \subseteq p(x)$

1 **if** axis = *child* **then**
2 **for** $s \in \text{siblings}(t)$ **do**
3 $P' := \text{nodePattern}(s)$;
4 $P'' := \text{inducePath}(s,\{t\},K)$;
5 $P := P \cup \{p'/p'' \mid p' \in P' \text{ and } p'' \in P''\}$;
6 **else** $P := \text{nodePattern}(t)$;
7 $R := \{\text{axis.transitive} :: p \mid p \in P\}$;
8 **if** $t \in \text{axis}(n)$ **then** $R := R \cup \{\text{axis} :: p \mid p \in P\}$;
9 **return** rescore$(n,\{t\},R)$;

Let us discuss the function nodePattern. Given a node u, this function iteratively computes a set of possible node tests followed by at most two predicates. In a first step, a node test for the node u is generated. This starts with the most general such test `node()` followed by the test of the tag name of node u. Then one predicate with an attribute (or text-value) equality-comparison is produced. The resulting queries, with axis prepended are tested on node n. If u is *not* uniquely matched, then an additional positional predicate is concatenated. Thus, each pattern returned by nodePattern contains at most two predicates: one attribute (or text) comparison (possibly) followed by a positional predicate. The selection of attribute names and text functions follows our definition of scoring for such query constructs (see Section 4). As an example if the node u is labeled `div`, then the first few node patterns that are returned by the call nodePattern(u) are of the form:

```
node()                          div
div[@id='x']                    div[@class='y']
div[contains(.,'z')]
```

Note that x,y,z are constrained to be single strings that appear in the input document as follows: either as single words (space-separated and/or bordered) or as the full text-value of a node.

The node tests P at hand, we complete these patterns by prepending axis.transitive to all patterns in P (Line 7). If t is reachable with a single step along axis, we prepend axis to all patterns in P (Line 8). At last, the algorithm returns the resulting patterns R, scored as expressions matching only t from context node u (Line 9).

As an example, consider the sample $\langle u,\{v\}\rangle$ where u is the body-node and v the em-node in this document:

```
<body>
  <div class="content">
    <div id="main">
      <em class="highlight">The Target</em>
</div></div></body>
```

This means that we have four nodes on the spine from u to v: the body-node, the two div-nodes, and the em-node. We now generate stepPatterns, starting at the lower div-node (with id-attribute), and matching the em-node. E.g. these expressions are generated:

```
descendant::em
child::em
child::node()[class="highlight"]
```

Next, similar patterns are generated for expressions matching the em-node, starting at the upper div-node, and then starting at the body-node. In the next step we generate stepPatterns for the lower div-node; first, starting from the div-node above it, e.g., the expressions **descendant**::div and **descendant**::div[@id="main"]. Next, patterns are generated that match the lower div-node, starting at the body-node, e.g., the expression **descendant**::div[@id="main"]. Note however that the expression **descendant**::div it *not* generated at this step, because it matches both div-nodes, and hence is not accurate (strictly speaking, the expression may be generated, but receives a very low score). To update the best patterns (from u to v), we now also generate combined patterns such as patterns such as

```
descendant::div[@id="main"]/child::em
```

Finally, stepPatterns for the upper div-node are generated, starting from the body-node, and are combined with the previous expressions, so that for instance the expression

```
descendant::div[@class="content"]/child::div[@id="main"]
                                                /child::em
```

is generated. The precise ranking of these expressions depends on the parameter; see Section 6.3 for typical parameter choices.

Algorithm 2: Axis Path Induction inducePath$(u,V,K,\text{axis},best,tar)$

input : node u and node set $V \neq \emptyset$,
$\forall v \in V$: v is axis.transitive-reachable from u,
$K > 0$ best-K bound,
$best$ initial table with best-K paths (wrt. score),
tar table of reachable target nodes
returns : query instances q with $q(u) \approx V$, ranked by score

1 **for** $v \in V$ **do**
2 **for** $t \in \text{spine}(v,u) - \{u\}$ **do**
3 **for** $n \in \text{spine}(u,t) - \{t\}$ **do**
4 $V' := tar(n)$;
5 **for** $1 \leq k \leq K$ **and** $p \in \text{stepPattern}(n,t,\text{axis})$ **do**
6 $p' := p/best(t)[k]$; $M := p'(n)$;
7 $q := \langle p', |M \cap V'|, |M - V'|, |V' - M| \rangle$;
8 **if** $q < best(n)[K]$ **then**
9 **insert** q **into** $best(n)$;
10 **return** $best(u)$;

Multiple-Target Query Samples. We now generalize to multipe-target query samples, i.e., samples $\langle u, V\rangle$ with $|V| > 1$. Thus, inducePath takes u, V, and K as arguments, along with pre-initialized tables *best* and *tar*, respectively for the best-K instances and the relevant targets to be matched (explained below). The *best* table is initialized as before to $\langle \bot, 0, 0, 0\rangle$ for all relevant entries, except for the first entry in every target node $v \in V$ which is set to $\langle \varepsilon, 1, 0, 0\rangle$. The table *tar* contains for each relevant node n the axis-reachable targets in V, where axis is the direction of the one-directional dsXPath reaching from u to V Both tables are handed in as arguments to enable the reuse of this algorithm in more general settings beyond the base case. For dealing with $|V| > 1$, the algorithm iterates over all targets $v \in V$ (Line 1) while maintaining a single best-K table *best* throughout all iterations. In each iteration v, inducePath induces the best-K instances which match v from u, possibly along with other targets from V. However, to produce accurate results, inducePath evaluates the accuracy of the induced instances against V (or subsets of V in for intermediate instances). Hence, $best(u)$ contains upon loop termination the so-far best expressions matching V and is returned as result (Line 10). The sign $\approx$ in the specification of the algorithm means that the query instances select *some* nodes of the sample. Note that $q < best(n)[K]$ in Line 8 of means that the ranking of q is strictly smaller than the ranking of the K-th query instance in the table $best(n)$.

For each v, inducePath computes for all nodes $n \in \mathsf{spine}(u, v) - \{v\}$ the best instances to match v and evaluates them against all reachable targets $tar(n) = V \cap \mathsf{axis.transitive}(n)$, given *tar* is initialized accordingly. To enable a dynamic programming approach, we need to ensure that the entries in *best* are already computed when the algorithm reads these entries. Thus, we first iterate (Line 2) over all possible anchors $t \in \mathsf{spine}(v, u) - \{u\}$ and in an inner loop (Line 3) over all nodes n for which t could serve as anchor, i.e., $n \in \mathsf{spine}(u, t) - \{t\}$. In each inner iteration, we enumerate all instances in $\mathsf{stepPattern}(n, t, \mathsf{axis}, K) \times best(t)$ (Line 5) but only add those to $best(n)$ which beat the K-best know instance (Lines 8–9). This way, $best(t)$ is not altered anymore, once the algorithm reads $best(t)$ to generate candidates for $best(n)$. The remaining lines in Algorithm 2 deal with the accuracy evaluation of the generated instances: we obtain the relevant targets V' and wrap each generated XPATH expression p' into an instance q with the true positives, false positives and negatives evaluated against V'.

Two-Directional Paths and Multiple Samples. Our general induction algorithm is shown in Algorithm 3. It uses inducePath from Algorithm 2 as subprocedure, and generalizes our previous considerations to two-directional paths and multiple samples. Two-directional paths are needed if a match node $v \in V$ is not a descendant (and not an ancestor) of the context node u.

(1) *Samples requiring two-directional paths* (Lines 5–15). A two-directional path from u_i to V_i is required, if not all nodes in V are reachable via the same base axis. Then induce searches for the closest node l_i such that u_i and V_i are both reachable via one-directional paths from l_i. This node l_i is the least common ancestor of either V_i or $V_i \cup \{u_i\}$ (Lines 5–7). Once l_i is fixed, induce first computes the best-K tail instances from l_i to V_i. For computing the final result, the algorithm induces instances leading from u_i to l_i, but with an deviating initialization: First, we take standard initialization for *best* (Line 8) but *initialize* $best(l_i)$ *with the tail instances* from l_i to V_i (Line 11). Then, the overall induced expressions lead from u_i to l_i and continue with one of the expressions in $best(l_i)$ to match the nodes in V_i. Second, we set the *targets for computing the accuracy* of the obtained instances to V_i, i.e., we set $tar(n) = V_i$ for all $n \in \mathsf{spine}(u_i, l_i) - \{l_i\}$, since we want to match V_i from each node between u_i and l_i.

Algorithm 3: Path Induction $Q = \mathsf{induce}(S, K)$

input : samples $S = \{\langle u_1, V_1\rangle, \ldots, \langle u_n, V_n\rangle\}$ such that $V_i \neq \emptyset$ for all $1 \leq i \leq n$, $K > 0$ as best-K bound
returns : query instances Q with $q(u_i) \approx V_i$ for all $q \in Q$ and $1 \leq i \leq n$

1 **for** $\langle u_i, V_i\rangle \in S$ **do**
2 **if** $\exists a \in B\ \forall v \in V_i$: *v is* axis.transitive-*reachable from* u_i **then**
3 $Q_i := \mathsf{inducePath}(u_i, V_i, K, a, \mathsf{init}(u_i, V_i, K))$;
4 **else**
5 $l_i := \mathsf{lca}(V_i)$;
6 **if** $\nexists a \in B$: l_i *is* a.transitive-*reachable from* u_i **then**
7 $l_i := \mathsf{lca}(V_i \cup \{u_i\})$;
8 $\langle best, \_\rangle := \mathsf{init}(u_i, \{l_i\}, K)$;
9 $a :=$ unique axis in B so that $\forall v \in V_i$:
10 v is a.transitive-reachable from l_i
11 $best(l_i) := \mathsf{inducePath}(l_i, V_i, K, a, \mathsf{init}(l_i, V_i, K))$;
12 **for** $n \in \mathsf{spine}(u, l_i) - \{l_i\}$ **do** $tar(n) := V_i$;
13 $a :=$ unique axis in B so that l_i
14 is a.transitive-reachable from u_i
15 $Q_i := \mathsf{inducePath}(u_i, \{l_i\}, K, a, best, tar)$;
16 **return** $\mathsf{aggregate}(\bigcup_{i=1}^{n} Q_i)$;

(2) *Multiple Samples.* We deal with multiple samples by first computing a best-K set Q_i for each sample $\langle u_i, V_i\rangle$ individually, and second choosing from these instances the ones which perform best on all samples (Line 16).

6. EVALUATION

Our experiments show that our wrapper induction produces accurate and robust XPATH expressions even from noisy query samples. We first compare our system with two state-of-the-art XPath wrapper induction system [6, 2]. We first evaluate the robustness of our induction system and look at change frequencies occurring on the involved pages (Section 6.2). Finally, we point out some characteristics of the induced expressions (Section 6.3) and analyze the noise resistance of the wrapper induction (Section 6.4).

Running time of the wrapper induction is generally in the same order of magnitude as page retrieval and loading. For a single node expression, it ranges from a few milliseconds to several seconds, with a median of 1.4 seconds and more than 60% of the cases running faster than page retrieval and loading.

6.1 State-of-the-Art Comparison

For robust wrapper induction, [6] presents one of the seminal approaches and includes experiments on IMDB pages. As the system is not available, we rather replicate their experiments as faithfully as possible: as in [6], we take 15 snapshots of director names on IMDB movie pages taken at 2 months intervals. If within such a 2 month interval, no new snapshot is available, we extend the interval until one becomes available, in order to ensure that we use 15 distinct snapshots. We use their success ratio measure as the percentage of snapshots at time t where the induced wrapper still works on the immediately following snapshot at time $t + 1$. We tried to approximate the likely time period used in their paper by considering the following three snapshot periods: 2004–2006, 2005–2007, and 2006–2008. Our approach achieves a success ratio of 100%, 86%, 86%, respectively. This compares to 86% reported in their paper [6], where the precise time period is not mentioned.

Site	Induced / Human XPATH queries	valid days	c-changes
S1	`descendant::div[@id="console"]/descendant::p`	400	4
	`descendant::div[starts-with(.,"Top")]/descendant::p`	400	4
S2	`descendant::h3[@class="f-quote"]`	382	16
	`descendant::div[@id="channel0"]/child::h3`	382	16
S3	`descendant::img[@class="adv"][1]` (rank=1)	300	2
	`descendant::div[@class="contentSmLeft"]/descendant::img[@class="adv"]` (rank=3)	456	2
	`descendant::div[@class="contentSmLeft"]/descendant::img[contains(@class,"adv")]` (rank=5)	1999	34
	`descendant::img[ancestor::div[1][@class="contentSmLeft"]]`	1999	34

S1 = www.foxnews.com, **S2** = espn.go.com, **S3** = www.wellsfargo.com

Table 1: Matching single nodes, two typical queries and one difficult case

We also compare with a more recent, automatic wrapper induction system [2] (WEIR). WEIR is based on the observation that different sources for a certain type of data often have some overlapping and is able to induce wrappers automatically in many cases. The trade-off is that WEIR requires multiple pages that follow the same template as input for each source, and each induced XPath expression matches at most one node per page. The induced expressions are generally of two types: absolute expressions that are similar to canonical paths, but starting from the closest anchestor of the target node with a unique ID. The second type of expressions are relative to a close-by "template" node, i.e., a node with a text content that is likely fixed (such as "Director"). To tailor to the limitations of WEIR, we select hotel pages from Tripadvisor with the same template. We let WEIR induce expressions from 10 such pages from 2012, and average over 5 such sets with differing pages and/or target nodes. WEIR exploits the availability of 10 pages to guess which nodes contain static content (such as "Country") and which contain variable values. WEIR's induction returns an unranked set of on average 30 expressions. We compare the robustness of these expressions over the time period from 2012 to 2016 against our system. For our system, we provide only a *single* page with the same target node as for WEIR. The top-10 expressions returned by our system survive on average 67% of the time period, compared with 32% on average for the 10 expressions returned by WEIR. If we compare the most robust expression returned by each system, ours survive for 93% of the period compared to 56% for WEIR. If we only consider the top ranked expression for our case, we survive 92%—our ranking very closely models robustness—of the period. Also note, that our top-ranked and best expressions are robust for the entire period in well over 80% of the cases, compared with around 15% for WEIR.

6.2 Robustness

We evaluate the robustness of the generated expressions in Figures 3 and 4 respectively for single and multiple target nodes. For both cases, we assembled a data set of 50 test tasks respectively, each specifying a URL u together with a manually crafted XPATH expression q leading to a single node in the first set or multiple nodes in the second set (between 3 and 59 nodes with an average of 10). The sites in the dataset come from over 20 different verticals, such as "Movies", "News", and "Travel", and include some of the most visited websites on the web. The data sets select nodes from a range of different node types arising in the construction of site wrappers: Such wrappers do not only select a single type of data on a page but navigate to the data via forms and links to extract multiple objects which are dispersed over multiple pages. To build such a wrapper, one needs to induce expressions matching form elements, menu entries, next links, and data attributes. Our data sets are assembled to reflect this variety of use cases.

To check the robustness of induced expressions, we obtain from the Internet Archive a sequence of snapshots taken from u. On the first snapshot we induce an expression q' to match the same nodes as q does. The induction is restricted to expressions which do not refer to textual data contents, i.e., texts that are not part of the template but belong to changing data, such as article titles. Such textual content would be too volatile and would cause the induced expression to break quickly.

Once induced, we evaluate each such expression q' on subsequent snapshots of u taken at 20 day intervals until **(1)** q' does not match the same nodes as q, **(2)** q breaks itself, or **(3)** we reach a given end date. We check whether q is still correctly working by evaluating a pre-specified predicate on the nodes matched of q. For example, such a predicate might check that the matched nodes belong to a certain class or start with certain string. When these predicates did not match anymore, we manually verified that q did break; if q was still valid, we improved the corresponding predicates. If the Internet Archive does not contain a required snapshot, we search for the closest existing snapshot as replacement. Our evaluations start 01-01-2008 and end 12-31-2013.

We compare the robustness of induced wrappers with canonical wrappers that consist of absolute XPATH from root node to target nodes, and human wrappers carefully designed by experienced experts. Note that many human wrappers are *not* in dsXPath using axes such as `following`, see query **S3** in Table 2. When designing human wrappers, we aimed at making them as robust as possible, in some cases even rewriting a wrapper that broke by manually inspecting the snapshot.

Results for matching a single node. Figure 3 shows a density distribution comparing the robustness of induced, canonical, and human wrappers. Intuitively, the higher the curve is towards the right, the more robust wrappers it contains. From the generated 53 expressions, 6 break within 100 days, however, most of these expressions are not robustly wrappable, since in 5 of these cases the corresponding manually crafted expressions fail as well. 27 expressions remain valid between 100 and 400 days, and 20 expressions remain valid for more than 400 days. Thus, discounting the 5 expressions which have no robust wrapper, we only fail in a single case, i.e., our wrapper induction is over 98%-robust.

Table 1 shows the induced expressions for three cases taken from our dataset (following the same format as Table 2). For sites **S1** and **S2**, we show the top-ranked induced expression in the first line and the manually written expression in the second line; for **S3** we show three induced expression and the manual one. In addition to the number of days the expression remained valid, we also show the number of changes in the canonical path leading to the target node during this period, as a very rough indicator for the amount of changes occurring on the page. **S1** achieves maximal robustness, since the top video is replaced by list of videos and thus it

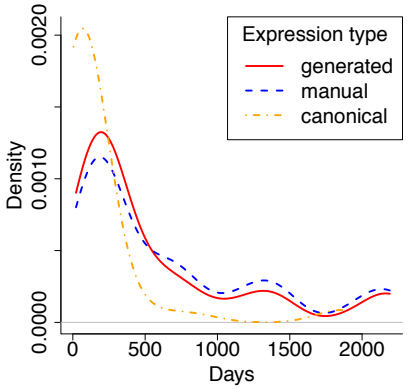

Figure 3: Robustness of expressions for matching a single node

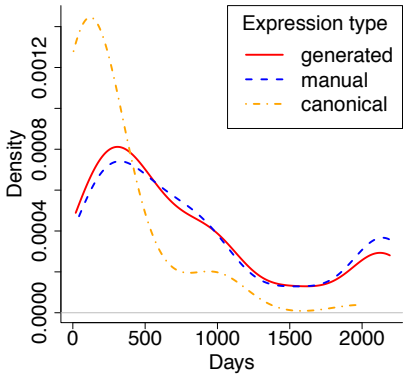

Figure 4: Robustness of expressions for matching multiple nodes

is unclear what the intended selection should be. **S2** also achieves maximal robustness, as the targeted quotation disappears from the site. **S3** is a hard case; our ranking does not work perfectly, since in the top-ranked expression the positional predicate just outruns the expressions involving a second step which is necessary for more robustness. On the other hand, the 5th induced expression would already deliver a very good result – after 1999 days, the involved attributes obtain new names which cannot be anticipated at all.

The **break reasons** for the wrappers fall into five groups:

(a) The first group contains wrapper pairs (induced/human) where both wrappers work over the *full test period* (2008–13), i.e., these are the most robust wrappers. From the 51 wrappers, four fall into this group. For instance, the IMDB search field is selected by this induced wrapper `descendant::input[@name="q"]` and, as the reader may verify, this wrapper still works today.

(b) For 7 wrappers, both human and induced wrapper fail at the *same* time, yet the target information is still present after the break date. This is typically the case when the markup has changed considerably, causing the wrappers to break. In fact, these changes typically coincide with a site-wide visual redesign. Preeminent for these 7 queries are changes in the values of important attributes: e.g., class attributes change from `"hp-content-block"` to `"homepage-content-block"`, or from `"headline20"` to `"headline16"`, or the id-attribute changes from `"searchInputArea"` to `"searchArea"`. We believe that these cases could be dealt with by incorporating more sophisticated text queries. We leave this issue for future work.

(c) For another 7 wrappers the induced wrapper works *longer*. This may seem surprising, but coincidentally our heuristics chose here an expression that proofed more robust. For example, the induced wrapper (on salesforce.com) q = `descendant::input[@type="text"][last()]` works during the entire period, while the human query `descendant::*[@id="search_box_hm"]`/q breaks after 1.5 years.

(d) Conversely, there are 2 cases where the induced query *breaks before* han the human one: first, the induced query `descendant::div[(@id="cnnT1Col")]/descendant::h1` breaks after 241 days, while the human query `descendant::*[@class="cnnT1Txt"]/descendant::h1` breaks only after 681 days. Coincidentally, in this case the `class` attribute proves to be more robust than the `id` attribute. In the last case, the human query `/descendant::img[@id="jobs"]/ancestor::a[1]` lasts for 1201 days, while the induced query `descendant::a[@href="http://www.jobs.nih.gov/"]` breaks after 1170 days, when a second node with the same `href` value is added to the page.

(e) 7 wrappers break because of *erroneous archive snapshots* which are either empty or structurally broken.

(f) 24 wrappers break when the *relevant targets were removed* from their respective pages; thus, no wrapper could survive such a change. Strictly speaking, these wrappers belong to Group **(a)**, as they did not break in the maximally possible (but not full) time range. Here we categorize them as an independent group to illustrate the fact that *no wrappers* could still work.

Results for matching multiple nodes. The results in Figure 4 for the expressions matching multiple nodes show similar but slightly better results. For example, only 2 out of 50 expressions are less than 100 days valid, with an average of 815 days and a median of 568 days. It is interesting to observe that in both datasets, after the initial density peaks at 250 (single node) and 350 days (multiple nodes), the density falls until reaching a minimum at 1500 days. Apparently, the cases before 1500 belong to sites which do change significantly eventually, while the cases beyond belong to sites which are never updated significantly at all.

Table 2 shows four of our multi-target wrappers. For sites **S1–S3** the top-ranked induced expression is shown in the first line, and the manually written expression in a line below; for **S3** we additionally first show the 49th-ranked induced expression. We show the number of target nodes, the number of days the expression remained valid, and the number of c-changes. The latter is a rough indicator of the amount of changes to the document that directly affect the path from the document root to the target nodes (see Section 2).

S1 matches a list of "channels" which are moved to another page after 219 days. Both the human and generated expression achieves perfect robustness. Note that the induced expression does not rely on textual content but on attributes only. **S2** does not break during the full observation period. The induced expression prefers the shorter string but is otherwise identical to the manual one. **S3** shows a good but suboptimal result: the `class`-attribute of our top-ranked expression is removed at day 339; in contrast, the manual expression (relying on textual contents) only breaks at day 622. Interestingly, our rank-49 induced expression (using no attributes or text) holds even for 700 days.

We checked by hand the **break reasons** of the wrappers. This time, no induced wrapper ever worked longer than the human made, so we have only four groups: **(a)** 6 wrapper pairs work over the full period, selecting general lists such as "latest news" items. **(b)** 10 wrapper pairs break at the same time. **(c)** There are no cases where the induced wrapper works

Site	Inferred / Human XPATH queries	#res	valid days	c-changes
S1	`descendant::a[contains(@class,"hpCH2")]/preceding-sibling::a[contains(@class,"hpCH")]`	22	219	1
	`descendant::div[contains(.,"Channels")]/descendant::a[@class="hpCH"]`		219	1
S2	`descendant::tr[contains(.,"News")]/following-sibling::tr`	7	2056	2
	`descendant::tr[contains(.,"News and Latest Reviews")]/following-sibling::tr`		2056	2
S3	`descendant::div[(@class="tvgrid")]/following-sibling::node()/descendant::li` (rank=1)	8	339	12
	`descendant::p/following-sibling::node()/descendant::li` (rank=49)		622	13
	`descendant::p[contains(., "Hit")]/following::ul[1]/descendant::li`		700	13

S1 = www.about.com, **S2** = www.mobiletechreview.com, **S3** = imdb.com

Table 2: Matching multiple nodes, queries with sibling axes (top-ranked / human)

longer than the human one. **(d)** 3 induced wrappers break before the human one. For example, the induced top-ranked wrapper `descendant::div[@class="widePanel"]/descendant::a` breaks after 817 days, while the human wrapper is robust for the entire 5 year period. However, the 30th ranked induced expression is fully robust too: it first selects the text `"offers:"` from a header and then continues with `/following-sibling::node()/descendant::a`. Another example is **S3** from Table 2, where the human selects a node containing the text `"Hit"`, while the induced query uses a `class` attribute. **(e)** 10 wrapper pairs break due to issues with the internet archive. **(f)** 21 wrapper paris break due to diminishing targets. Often, the removal of the list information co-occurs with a complete visual redesign of the respective site.

Both examples of induced wrappers that fail before the human ones suggest that selection through the text-value of nodes gives more robust wrapper than through attributes. This is confirmed by analyzing the 10 cases where induced and human wrappers break simultaneously: often the only possible robust approach is based on selecting keywords within the target list. We believe that our approach can induce robust queries for these cases; however, as we mostly target volatile data entries, our configuration is biased to deal with (relatively) dynamic item lists.

Change Rate. Here our discussions will be for both, single- and multiple-target wrappers. We use a very rough change measure: only if the *canonical path* to the target nodes change, do we count this as one change. Thus, if such a "c-change" occurs, then we increase the change counter, induce a new path, and continue in the same way. Recall that the canonical path consists only of element names and position numbers. In this way, we obtain *very low* change frequencies, the largest being 25 (obviously, many more things change on the pages, even considering attribute values on the canonical path gives completely different numbers). Since the canonical path is our yardstick for a simple wrapper, we find c-changes an appropriate measure. For single-target wrappers, 11 wrappers survive more than 5 c-changes; 16 wrappers survive exactly 1 c-change (that being the largest group). The average is at 4.1 c-changes. For multiple-target wrapper, 13 wrappers survive more than 5 c-changes, but the highest number is only 19 (versus 25 for single). Interestingly, the average is again 4.1 c-changes. We only checked two sites (IMDB and espn) by hand to see what kind of canonical path changes occur. The most frequent here, were positional changes on div nodes. For instance, in a typical path (from IMDB) `html[1]/.../layer[1]/div[1]/div[3]/div[4]/form[1]/input[1]` the `div[4]` becomes a `div[3]` in a new version. Next frequent changes are inserted or removed div nodes on the canonical path (i.e., changing the length of the path).

6.3 Expression Characteristics

Parameter Choices. As explained in Section 4, the typical expression characteristics are chosen as suggested by our experience, as well as previous literature on wrapper induction, e.g., that `descendant` is preferred over `child`. These are further discussed below. The most important other parameter of our approach is the *decay factor*, which favors anchors further away from the targets. Our decay factor is $\delta = 2.5$, as was determined as optimal through a sequence of experience varying delta between 0.5 and 5.

For single- and for multiple-target wrapper induction we use the *same* ranking parameters. We use *no* specific scores for different tags, but merely $c_{\text{node()}} = c_* = 1$ and $c_{default} = 10$. Our positional factor is 20, *no-function-penalty* is 15, and *no-predicate-penalty* is 1000. All other parameters are as follows:

Axes		Attributes		Functions	
descendant	1	id	1	equals	1
attribute	1	type	1	position	1
foll.-sibling	1	title	1	contains	5
child	10	class	5	starts-with	5
parent	10	for	10	norm.-space	5
ancestor	20	name	50	last	20
prec.-sibling	25	*default*	1000	string	100

As an example, for `descendant::img[@class="adv"][1]` we compute as score $c_{\text{descendant}} + c_{default} = 1 + 10 + 1 = 11$ plus the two predicates. The first one gives $s_{\text{class}} = 5$ plus $1 \cdot 3$, and `[1]` gives 21. The final score equals 40.

Analysis. Starting with the 53 expressions generated to match a single node, we see that 34 of these expressions have 1 step, 19 expressions have 2 steps, amounting to 72 total XPATH steps. All of these steps follow the `descendant` axis. Figure 5 gives some details on the characteristics of these expressions: On the left the figures shows that 26 of the 72 steps check for `div` elements, followed by `input`, and `a` elements. Of these 72 nodetests, 62 are refined by at least one predicate, and 10 are refined by two predicates, yielding again coincidentally 72 predicates. As shown in the right of Figure 5, these predicates check in 26 cases for an `id` attribute, in 18 cases for an `class` attribute, and in 17 cases for a position. In contrast, textual contents are only checked in 2 cases. This data suggest the following pattern: If the expression contains two steps, the first step identifies the region containing the targets by typically matching a `div`, `input`, or `a` node, predicated with an `id` or `class` attribute or position. The second step matches the target node, referring to a wide range of different tag names and often checking for its position within the context.

The expressions to match multiple nodes have quite different characteristics than the ones for single nodes. Fist of all, only 9 out of 50 examples use a single step, 34 employ 2 steps, and 7 need

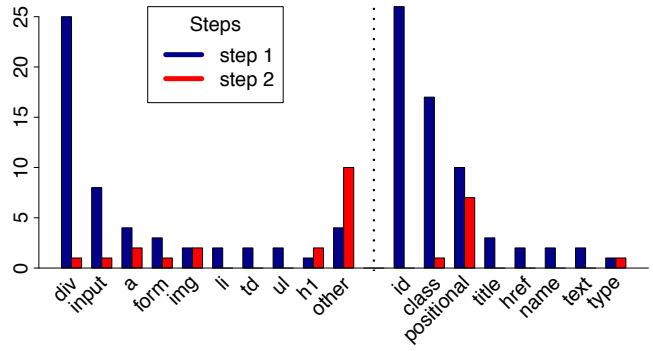

Figure 5: Nodetests/predicates of single-target queries

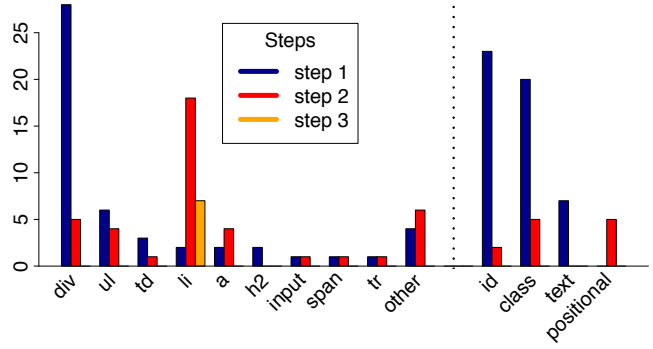

Figure 6: Nodetests/predicates of multiple-target queries

3 steps, yielding a total of 98 steps. These steps employ a bigger variety of axes, although 88 still rely the `descendant` axis; 7 steps use `following-sibling`, 2 step use `preceding-sibling`, and 1 step `child`. The use of the sibling axes is at first surprising, as most useful node lists are grouped below a common ancestor. However, to identify the correct subset of siblings belonging to our target list to avoid, e.g., adverts at the beginning or end, robustly matching lists require sibling anchors. For instance, query **S3** in Table 2 selects the first ul-sibling that follows the h3-node, and similarly, query **S4** (at rank 49) select all list items under following siblings of a p-node. The nodetests, shown on the left of Figure 6, also check in 33 cases for the `div` tag. However, the next nodetests check for unordered lists and table entries with `ul` and `td` as tags containing or neighboring the target nodes. Finally, the 27 steps checking for list items with the `li` occur mostly in the second and third step to match the sequence of target nodes. The right of Figure 6 shows the the distribution of the 62 predicates occurring in the induced expressions. These predicates mostly check for `id` and `class` attributes. However in contrast to the single-target cases, the expressions for multiple targets do not check for any other attribute; aside these, only 7 expressions check in the first step for textual contents, and only 5 expressions check in their second step for positions.

6.4 Accuracy and Noise Resistance

Real-Life Noise. In this experiment, we verify that indeed our approach is able to deal with realistic noise as produced by a real-life entity recognizer. We use the Stanford NER [11] to annotate 10 pages randomly sampled from a set of product listing websites (cf. [13]). We only select pages that contain at least one list of entities supported by the Stanford NER, here one of date, person, location, organisation, money. The size of the lists vary between 8 and 77 elements. For each of these pages, we run the Stanford NER and map its annotations to DOM nodes that serve as target nodes for our approach. The resulting input contains on average 32% negative and 28% positive noise, though both vary widely: $0-67\%$ for negative and $0-145\%$ for positive noise, with each page having at least some noise.

Our approach deals surprisingly well with these noisy samples: in 80% of the cases, our top-ranked expression identifies the correct intended set of nodes, ranging from dates, book authors, artists, and event locations to prices. There are two cases where it fails. **(1)** For "organisations" on newyorknewyork.com, the NER returns 145% positive noise, yet misses 55% of the sought-for target nodes. The returned expression is overgeneralized to a significantly larger set of nodes. **(2)** For "persons" on waterstones.com, the NER returns 50% positive, and 28% negative noise. The target nodes are authors of books returned by a search. Unfortunately, most of the positive noise is structural, annotating a list of author names in a sidebar list used to refine the query. The generated expression selects that list rather than the intended one. Both of these cases could be addressed by limiting the induction to the main content area, obtained by removing boilerplate content [17]. However, this is out of the scope of this paper. It is worth pointing out, that even if we eliminate the two cases where the system fails, noise remains significant (30% negative and 11% positive on average), yet in all of these cases our method compensate flawlessly for that noise.

Synthetic noise. Here we evaluate noise resistance by adding or removing a precentage of the target nodes of a sample. We evaluate different four types of noise: **(N1)** *negative random noise* **(N2)** *negative mid-random noise* where the first and last nodes (in document order) of a target set are not removed, **(N3)** *positive structured noise* where we add random nodes chosen from a node set which is structurally related (via an XPATH expression) to the target nodes. Finally, we consider **(N4)** *positive random noise* where we add random leaf nodes to the target set. We use two datasets: The first one for negative noise has 100 query samples matching between 3 and 59 nodes, with a median of 6 and an average 9.43 nodes. The second one for positive noise features 50 query samples matching between 2 and 100 nodes, with a median of 20 and an average of 34.92.

Figure 7 summarizes our results for all noise types. We show for the noise intensities of 10, 30, 50, and 70% the percentage of cases where the induction with and without noise delivered identical results. Thus, we evaluate the noise resistance in the most aggressive way. Typically, with negative noise, around 10% of the cases deliver with noise a result which is at least within the top 50 results induced without noise. With positive noise, the fraction of such expressions is negligible. Our system deals well with **(N1)** negative random noise at 10 and 30% intensity in achieving *identical* results in respectively 88% and 74% of the cases, thus showing that our approach is 88%- and 74%-noise resistant. However, at 50 and 70% intensity, 38 and 70% of the cases deliver non-identical results. Seemingly unsatisfying, this relatively low noise resistance is caused by removed head and tail nodes – which are critical to determine the start and end of a sequence of list entries. Therefore, we consider **(N2)** negative mid-random noise where we keep the very first and last node (in pre-order) and choose the noisily removed nodes only from the remaining nodes in between. In this case, our system returns identical results in 93, 83, 72, and 66% of the cases for the analyzed intensities. Our system deals exceptionally well with positive noise. For **(N3)** structural positive noise until 50% intensity, our system achieves in at least 90% of the cases an iden-

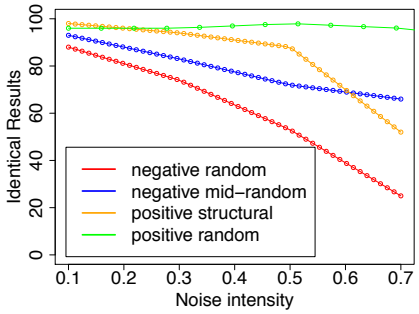

Figure 7: Result degradation with increased presence of noise

tical expression with and without noise. At 70% intensity however, this percentage drops to 54%. For **(N4)** we see almost no effect on the results, e.g., at 70% noise intensity, our system delivers the same result as without noise in 96% of the cases. Even at a noise intensity of 300%, we obtain identical results in 84% of the cases.

7. CONCLUSION

We present a wrapper induction algorithm that combines the ability to deal with noisy samples with a robustness that allows generated wrappers to often survive until the selected data is no longer present in the page. The experiments point to a number of further directions to improve our methods: **(1)** Extending the method to deal with *multi-node wrappers* where not only a single item or list of items, but multiple related items are to be extracted, is a natural step forward. Our method is already designed to allow the induction not only of absolute, but also of relative expressions and thus should be a great fit for a multi-node wrapper as in [13]. **(2)** *Learning an effective scoring* for different types of node types, textual values, and axes from a given corpus of websites. This is particularly relevant to distinguish different text fragments used to identify the same node and to best reflect specifics of a scenario. **(3)** Entity extractors, as used in information extraction or to generate automatic annotations in, e.g., [5], can provide information about the type of entity (e.g., "director") of a piece of text. Using such types in a wrapper to identify the containing node may yield more robust wrappers. **(4)** No matter how sophisticated the wrapper language or scoring, without constraints on the shape of future versions of a page, the robustness of a single wrapper will always be limited. Therefore, we are now *inducing multiple wrappers* that use a variety of independent means for selecting a target node.

Acknowledgements

We thank the anonymous referees for many insightful comments which greatly helped to improve the quality of this paper.

8. REFERENCES

[1] R. Baumgartner, S. Flesca, and G. Gottlob. Visual Web Information Extraction with Lixto. In *VLDB*, pages 119–128, 2001.

[2] M. Bronzi, V. Crescenzi, P. Merialdo, and P. Papotti. Extraction and integration of partially overlapping web sources. *Proc. VLDB Endow.*, 6(10):805–816, Aug. 2013.

[3] S. L. Chuang, K. C. C. Chang, and C. Zhai. Collaborative wrapping: A turbo framework for web data extraction. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1261–1262, April 2007.

[4] S.-L. Chuang, K. C.-C. Chang, and C. Zhai. Context-aware wrapping: Synchronized data extraction. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 699–710. VLDB Endowment, 2007.

[5] N. Dalvi, R. Kumar, and M. Soliman. Automatic wrappers for large scale web extraction. *PVLDB*, 4(4):219–230, 2011.

[6] N. N. Dalvi, P. Bohannon, and F. Sha. Robust web extraction: an approach based on a probabilistic tree-edit model. In *SIGMOD*, pages 335–348, 2009.

[7] N. Derouiche, B. Cautis, and T. Abdessalem. Automatic extraction of structured web data with domain knowledge. In *ICDE*, pages 726–737, 2012.

[8] B. Fazzinga, S. Flesca, and A. Tagarelli. Learning robust web wrappers. In *DEXA*, pages 736–745, 2005.

[9] B. Fazzinga, S. Flesca, and A. Tagarelli. Schema-based web wrapping. *Knowl. Inf. Syst.*, 26(1):127–173, 2011.

[10] E. Ferrara, P. D. Meo, G. Fiumara, and R. Baumgartner. Web data extraction, applications and techniques: A survey. *Knowledge-Based Systems*, 70(0):301 – 323, 2014.

[11] J. R. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *ACL*, pages 363–370, New York, NY, USA, 2005. ACM.

[12] S. Flesca, G. Manco, E. Masciari, E. Rende, and A. Tagarelli. Web wrapper induction: A brief survey. *AI Commun.*, 17(2):57–61, 2004.

[13] T. Furche, G. Gottlob, G. Grasso, X. Guo, G. Orsi, C. Schallhart, and C. Wang. DIADEM: Thousands of websites to a single database. *PVLDB*, 7(14):1845–1856, 2014.

[14] G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The complexity of XPath query evaluation and XML typing. *J. ACM*, 52(2):284–335, 2005.

[15] P. Gulhane, R. Rastogi, S. H. Sengamedu, and A. Tengli. Exploiting content redundancy for web information extraction. In *WWW*, pages 1105–1106, 2010.

[16] W.-S. Han, W. Kwak, H. Yu, J.-H. Lee, and M.-S. Kim. Leveraging spatial join for robust tuple extraction from web pages. *Inf. Sci.*, 261:132–148, 2014.

[17] C. Kohlschütter, P. Fankhauser, and W. Nejdl. Boilerplate detection using shallow text features. In *WSDM*, pages 441–450, New York, NY, USA, 2010. ACM.

[18] N. Kushmerick, D. S. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *IJCAI*, pages 729–737, 1997.

[19] J. Lehmann, T. Furche, G. Grasso, A.-C. N. Ngomo, C. Schallhart, A. Sellers, C. Unger, L. Bühmann, D. Gerber, D. L. Konrad Höffner and, and S. Auer. DEQA: Deep Web Extraction for Question Answering. In *ISWC*, pages 131–147, 2012.

[20] W. Liu, X. Meng, and W. Meng. Vide: A vision-based approach for deep web data extraction. *IEEE Transactions on Knowledge and Data Engineering*, 22(3):447–460, 2010.

[21] A. G. Parameswaran, N. N. Dalvi, H. Garcia-Molina, and R. Rastogi. Optimal schemes for robust web extraction. *PVLDB*, 4(11):980–991, 2011.

[22] J. Raposo, A. Pan, M. Álvarez, and J. Hidalgo. Automatically maintaining wrappers for semi-structured web sources. *Data Knowl. Eng.*, 61(2):331–358, 2007.

[23] W. Su, J. Wang, and F. H. Lochovsky. ODE: Ontology-Assisted Data Extraction. *TODS*, 34(2), 2009.