WebSelF: A Web Scraping Framework

Jakob Thomsen[1], Erik Ernst[1], Claus Brabrand[2], and Michael Schwartzbach[1]

[1] Aarhus University: {gedefar,eernst,mis}@cs.au.dk
[2] IT University of Copenhagen: brabrand@itu.dk

Abstract. We present WebSelF, a framework for web scraping which models the process of web scraping and decomposes it into four conceptually independent, reusable, and composable constituents. We have validated our framework through a full parameterized implementation that is flexible enough to capture previous work on web scraping. We conducted an experiment that evaluated several qualitatively different web scraping constituents (including previous work and combinations hereof) on about 11,000 HTML pages on daily versions of 17 web sites over a period of more than one year. Our framework solves three concrete problems with current web scraping and our experimental results indicate that composition of previous and our new techniques achieve a higher degree of accuracy, precision and specificity than existing techniques alone.

1 Introduction

The World Wide Web is an enormous source of information, (still) mostly represented as HTML which is designed for presenting information to humans, not computers. Therefore, automated information extraction from the web (aka., *web scraping*) is difficult. A program for web scraping, called a *web wrapper*, may be programmed manually [23,25], semi-automatically [14,22,2,11], or automatically [15]. We refer to the survey by Chang et al. [6] for more information.

However, when a web page changes (and similar web pages may have substantially different structure), the extraction often fails or yields incorrect data causing programs that depend on the scraping to malfunction. Web wrappers use *wrapper validation* (aka. wrapper verification) to detect this, typically based on the extracted text [20,10]. Updating the wrapper to recover is known as *reinduction* [10,12,13,17,8,16], and it is often based on older pages and/or user interaction. Validation and reinduction together constitutes *wrapper maintenance*.

Current approaches suffer from three problems. First, wrapper validation based solely on the textual contents and structure of the scraped page may be difficult or inadequate in certain cases. For some pages, it is worth also considering the *context* and *presentation* (i.e., *where* information is physically located on a page after full rendering and applying stylesheets). Second, with client-side scripting (e.g., JavaScript and AJAX) and form elements, it becomes useful to interact with a web page beyond just extracting information. Access to selected elements on a page allows for subsequent manipulation of the original document

(e.g., pressing buttons, filling in text fields, submitting forms). Third, existing wrapper techniques cannot easily be combined, which makes it hard to reuse the vast amount of work done on selection, reinduction, and validation. There is no general and precise signature definition of the components of the web wrapper, how they interact or what level of automation they exhibit.

This paper presents a framework, WebSelF, that addresses these problems for selecting elements on a web page. WebSelF is characterized and parameterized by four scraping constituent functions, that we call framework functions: one for selecting elements, designated as a *selection function*; one for validating the selected element, designated as a *validation function*; and two for maintaining each of them, designated as *reinduction functions*. WebSelF is novel for three reasons. First, it supports composition and reuse of previously defined validation functions, whereby validation can benefit from not only the textual contents of the selected elements, but also combinations of other dimensions (in particular, context and presentation). Secondly, the selection functions in WebSelF are able to not just extract information, but also subsequently manipulate selected elements in the presence of client-side scripting. Finally, WebSelF has a precise model that explicitly divides the labor into a manual and an automatic part, in which the responsibilities of the four functions are explicitly given. Furthermore the signatures of the four functions are defined, which allows WebSelF to easily wrap and use existing selection, validation and reinduction functions.

We have implemented WebSelF in Java, parameterized by the four framework functions, and used the implementation to author, evaluate, and compare validation functions based on different dimensions (including the influential work of Lerman et al. [10]). The evaluation shows that presentational features of the selected elements are beneficial for validation. Further, it shows that combining existing validation functions (such as the previous approach by Lerman et al.) with other orthogonal dimensions may achieve higher *accuracy*, *precision*, and *specificity* (defined later) than the original approach. The validated elements in our experimental evaluation are selected by real world web wrappers, as most of the selection functions have been harvested from the web where they are used for real web scraping purposes. The evaluation results have high credibility, because every selection has been manually verified. The main contributions of this paper is a framework, WebSelF, for web scraping including:

- a model of the process of web scraping (Section 2) explicitly dividing the labor, by decomposing the process into a *selection function*, a *validation function*, and *reinductions* of both, along with a method of composing validation functions (Section 3);
- a full implementation, parameterized by reusable composable framework functions (available at the WebSelF site: `cs.au.dk/~gedefar/webself`); *and*
- an experimental evaluation and comparison of qualitatively different validation functions (including previous work) based on about 11,000 HTML pages taken from 17 web sites over a period of more than one year (Section 4).

```
<body>
<p>ICWE 2012 is to be held
in Berlin, Germany.</p> ...
</body>
```

(a) A simple web page.

```
<body>
<p>ICWE 2012 will be
held on July 23-27.</p> ...
</body>
```

(b) A second version of the web page.

```
<body>
<p>The ICWE 2012 <a..>program</a>
is available.</p> ...
</body>
```

(c) The third version of the page with an added `anchor` tag.

```
<body>
<p>
<img src=".." alt="Group Photo"/>
</p>
<p>The ICWE 2012 conference
was a success.</p> ...
</body>
```

(d) A fourth version of the web page, where an image has been added.

Fig. 1. Evolution of an example web page from a conference news site over time.

2 A Model of the Process of Web Scraping

In this section we introduce the basic structure of our framework by means of a model of the process of web scraping, built from several individual framework functions with a specific interaction. To illustrate the principles, we will use a deliberately simple example and scrape successive evolutions of the "top story" of a conference news site. In WebSelF, a web wrapper consists of a *selection function*, a *validation function* and means for *reinducing* (learning) new versions of both of these functions. As the selection function, the example uses an XPath expression (which is common) with the additional convention that only the *first* element is selected in case the XPath expression matches several elements. For the validation function, the example will use *structural identity* with respect to a DTD[5] in that a selected element is accepted if it conforms to a DTD inferred over the elements selected so far (cf. Bex et al. [4]). The initial DTD is inferred from the first example (which in this example is a `p` tag containing only text).

Figure 1(a) shows an excerpt from the first version of the web page, and the interesting piece of information is the `p` tag and its contained text. The initial XPath expression is `//p` which selects that item in this case.

The next version of the web page is shown in Fig. 1(b). The XPath expression works here by extracting the `p` tag and the validation function accepts it, as the structure has not changed.

The next version of the web page is shown in Fig. 1(c). The XPath expression still works here, but the validation function rejects the element because an `anchor` tag with a link to the program has been added to the `p` tag. The framework now asks the user a simple *yes*/*no* question for whether or not the `p` tag is the correct element. In this case it is, so a new validation function is reinduced (learned) to allow the new `anchor` tag. The new validation function accepts `p` tags with an optional `anchor` tag as its child intermixed with the text (remember that the DTD is inferred over the first three versions).

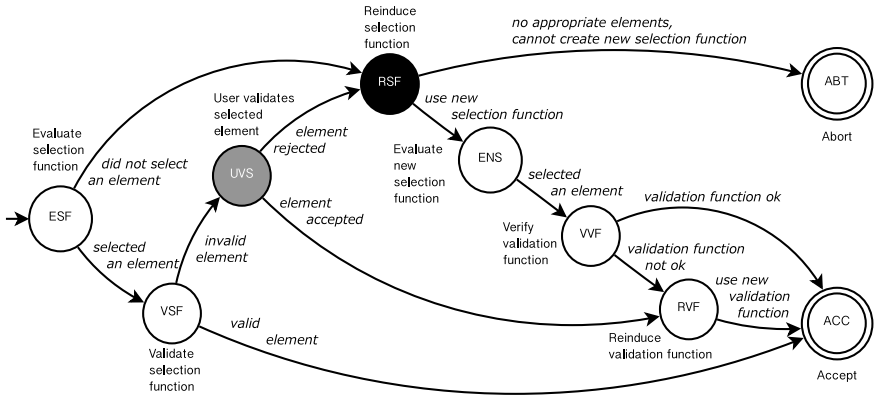


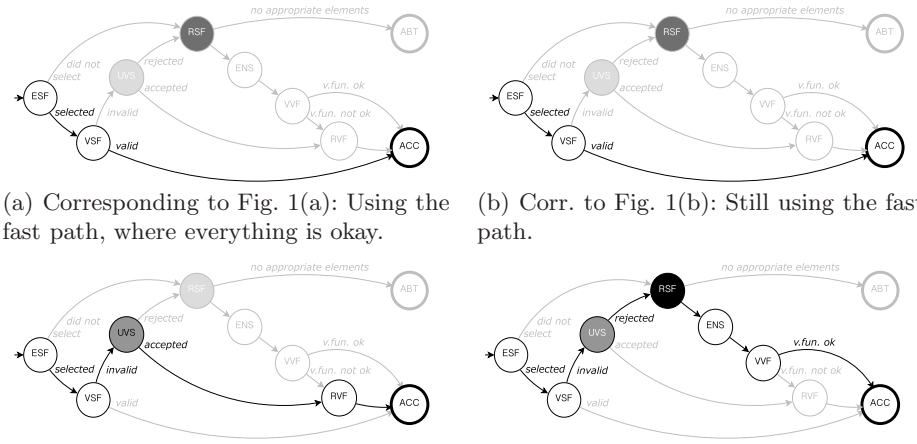
Fig. 2. A model of the process of web scraping.

In the next version, shown in figure 1(d), a new `p` tag with an `img` tag has been added before the interesting `p` tag. In this case the selection function selects the first `p` tag, which does not conform to the previous version of the element. This discrepancy is detected by the validation function, and as it is in fact not the correct element (judged by the user) the XPath expression needs to be *reinduced*, i.e., a new and improved selection function must be constructed. The framework uses the reinduction function for selection functions for this purpose, and this function could consult the user and/or the validation function in order to perform the task. As a result, the XPath expression is updated to select the correct element, e.g., by becoming `//p[not(./img)]`.

The running example illustrates different flows captured by our framework. The general and formal flow between the states of WebSelF is shown in the state diagram in Fig. 2 which shows a model of the web scraping process. Each state has an abbreviated name consisting of three upper-case letters; a nearby explanatory text motivates the choice of letters. Transitions are labeled with a text indicating the decision criterion associated with that transition. One run of the algorithm starts at ESF and ends in a final state—indicated by a double ring—and yields an accepted element from the given page or aborts to indicate that the page does not contain any acceptable elements. At the same time the algorithm maintains the selection function and the validation function, using the provided facilities for reinduction.

The states with a colored background may involve user-interaction and the particular color signifies the level of complexity in the interaction. The state with gray background (UVS) is related to the reinduction of the validation function and is only asking a simple *yes*/*no* question. In contrast, the state with black background (RSF) is related to the reinduction of the selection function, which may involve the user in a much more complex manner.

For a fully formalized description of WebSelF, where the precise signatures of the involved functions and flow of data and control between the states is



(a) Corresponding to Fig. 1(a): Using the fast path, where everything is okay.

(b) Corr. to Fig. 1(b): Still using the fast path.

(c) Corr. to Fig. 1(c): Using a not so fast path, where the selection function is correct, but the validation function is not.

(d) Corr. to Fig. 1(d): Using the slowest path where the validation function is correct, but the selection function is not.

Fig. 3. State model paths corresponding to the example web page evolution.

specified, we refer to the accompanying technical report [18]. It contains proofs for a *preservation* and a *progress* property; it also shows in detail how the work by Lerman et al. [10] may be emulated in WebSelF.

In the following we just describe the information flow in terms of our four versions in the running example. Consider the first version of our running example, shown as Fig. 1(a). As the selection function selects an element, we go from the initial state ESF to VSF. The selected element is accepted by the validation function, so we continue directly to the ACC state, without reinducing a new validation function. This path through the state diagram is illustrated by the graph in Fig. 3(a). This is the fast path where everything works smoothly, and hopefully it is also a very typical path. The flow of the second version (Fig. 1(b)) is completely identical to the first version's flow, where it takes the fast path and is likewise shown in Fig. 3(b).

For the third version, an element is likewise selected and so we go to the VSF state. This time, however, the validation function rejects the selected element, so we proceed to the UVS state where the user is asked. The user decides that the element should be accepted, so we go to the RVF state. Since the validation function made an incorrect decision we obtain an improved validation function through reinduction. This path through the state diagram is illustrated by the graph in Fig. 3(c). As a side note, in WebSelF we assume it's always possible to reinduce a validation function, as there is no requirement that the reinduced validation function is improved compared to the previous validation function. In fact the reinduction function is free to return the old validation function in case it is not capable of producing an improved validation function.

Finally, if we run the fourth version we still go to VSF first as an element is selected, but from VSF we go to UVS and then in turn to RSF, because the wrong element was selected, according to both the validation function and the user. The selection function is reinduced as we transition to the ENS state and on to the VVF state. The new element found by the new selection function is accepted by the validation function, so we do not need to reinduce (learn) a new validation function. This is shown in Fig. 3(d).

We have not covered a case where the selection fails entirely, but this could occur if we receive, e.g., an HTTP error 505 (internal error on server). In this case the path taken would be the uppermost path in the state diagram, from ESF to RSF to ABT. Finally, there is a case where both reinductions occur, which is a slight variant of Fig. 3(d).

To sum up, our framework only involves the user when the discrepancy between already learned examples and the new version is too big. What this means will be explained in Sec. 3.2. Moreover, the user is first involved in a simple manner by being asked whether a given selection should be accepted or not. If the blame falls on the validation function it is updated automatically, and only when the selection function is to blame does the framework proceed to involve the user in the more complex task of reinducing (creating) a new selection function. Other approaches [10,20,17] require a set of prelabeled examples to learn from, and they do not support a similar division of labor between a validation function that allows for automatic maintenance and a selection function which may be tailored to embody an arbitrary amount of domain knowledge. Our approach supports this division and our selection and validation functions are composable to utilize arbitrary domain knowledge. We furthermore believe that the precise description of WebSelF can aid in the engineering of web information extraction software.

3 Framework Instances

There are many kinds of selection function and validation function, as well as reinduction functions for either. This section discusses a few of these at an abstract level. An important choice to make is concerned with the environment provided for evaluation of the selection and validation functions. If a full browser is available for rendering the page and running client side code (e.g., JavaScript and AJAX), as opposed to working straight from the HTML source, the validation function can rely on presentational information beyond the contents and structure of the page, e.g., screen coordinates and colors. For generality, we consider below the situation where the pages are rendered in a full browser (our framework implementation directly supports this).

3.1 Selection Functions

A selection function is responsible for choosing a piece of information (i.e. an HTML element, a list of HTML elements, tuples of text, etc.) from a given web

page and delivering the *selection result* in a suitable format. It can be anything from trivial to near-impossible to specify the "correct" selection for a web page.

However, a few generic possibilities do exist. In particular, XPath expressions were specifically invented in order to be able to designate elements in an XML tree structure. Also, regular expressions and context free parsers are commonly used to locate specific elements by their content and structure. The work by Lerman et al. [10] utilizes a hierarchy of token classes to select textual elements based on sequences of token types. Finally Kushmerick et al. [9] induce selection functions based on delimiters.

An important property of a selection function is its *robustness*, or its ability to "just keep working" when it is used on evolved versions of a web page with similar but updated content and structure. As with the structure itself, page specific approaches and general computation may be needed to deal with such updates. Myllymaki & Jackson [24] discuss the characteristics of robustness for selection functions based on XPath, but they do not discuss any mechanical way to achieve it. Lately, different mechanical techniques to 'robustify' XPath expressions have been proposed [7,8,16]. Basically all the techniques rewrite the XPath expression into a more robust expression relative to the changes seen in a set of training web pages, and they achieve much better robustness than a corresponding fully specified XPath expression.

Reinduction of selection functions is in general just as hard as inventing them in the first place. For generic selection functions it may be possible to derive the selection function from a history of positive examples; e.g., as it is done by Lerman et al. [10]. This process may be seen as an abstraction process whereby the desired element is described by successively more abstract and inclusive specifications, until it matches all the positive examples. Incrementally building a specification works well for semi-static information where only a part of an element is changing, such as an address or a form field, where the static part of the element can be used as an anchor. If the goal is to extract frequently changing information, such as the *top* news story from a news site, then the selection can benefit from using contextual and/or presentational information, such as the structure of the context (e.g., a highly specific `class/id` attribute or the (x, y) screen coordinates of the elements). Some reinduction approaches [12] support manually specifying a fixed context to search for, others automatically infer it from the history [10]. In general, reinduction of selection functions is likely to require supervision.

3.2 Validation Functions

A validation function validates a selection result by either accepting or rejecting the result from a web page. (In the following discussion of validation functions we will assume for simplicity that the selection result is a single HTML element.) Typically, validation functions utilize textual dimensions of both the selection result along with the original web page when validating, but when the element to be selected changes significantly from time to time, other dimensions might be more effective. It may be more informative to investigate for instance the

context (e.g., the tree structure from the grand parent of the selected element). Furthermore, human spectators often rely strongly on the appearance of a web site, and this realization is likewise very useful. For instance, a selection result is likely to be rejected if it appears physically far away from its typical location on the web page.

Validation function *reinduction* is the process whereby an existing validation function is replaced by an updated one that is known to make more appropriate judgments. In WebSelF, as in other approaches [10,20], the validation function is reinduced with respect to a history of selection results. In the theoretic treatment of WebSelF all previous selection results are available, but in a concrete implementation this set can be a too large, so it often suffices to only use the selection results that the validation function wrongfully rejected.

In general, validation functions can be more generic than selection functions because they must primarily flag the occurrence of anomalies. It is our experience that a validation function can often use a generic algorithm customized by a number of parameters, and reinduction then amounts to adjusting those parameters so that the validation function responds more favorably to a given selection history.

One issue to consider in connection with validation function reinduction is whether the new validation function should learn to recognize all examples in the given history. We may wish to suppress the consequences of processing exceptional web pages, also known as *outliers*. The problem is that a validation function may become overly permissive, because a few outliers has taught it to tolerate almost anything. It may be better to reject (or ask for explicit user confirmation in) a few unusual cases, and then retain high selectivity. One example of a validation reinduction function which does not learn to recognize all examples in order to suppress outliers is the one from Lerman et al. [10], as it only includes the examples in the history that are statistically significant.

It is possible to create *composite* validation functions based on existing validation functions. This is particularly interesting as we are able to logically combine qualitatively different (and complementary) validation function strategies; ones that work according to *content*, *structure*, *presentation*, and even *context*. If we, for instance, want to extract the top news story from a news site, we might have to combine looking for a styled heading (structure and presentation) placed close to the top of the page (presentation).

Since a validation function returns a boolean result, we can easily compose validation functions to achieve any *propositional logic* formula, ϕ, over basic validation functions, $Q \in \mathcal{Q}_{\text{BASIC}}$:

$$\phi \quad : \quad \mathit{true} \mid \mathit{false} \mid Q \in \mathcal{Q}_{\text{BASIC}} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$$

Reinduction of a composite validation function can be done in many ways, but often it is done by delegating the reinduction to its failing constituents (according to its constituent validation functions). Negation needs special treatment though. Say for instance that $\neg\phi$ has wrongly rejected an element a, which means that ϕ accepted a. As $\neg\phi$ is reinduced, ϕ should be reinduced to learn to reject a

which it used to accept. Hence as ϕ gradually becomes more permissive when reinduced, $\neg\phi$ will gradually become more restrictive. We will see in Sect. 4.3 that negation can be very useful, despite its somewhat counter-intuitive nature. All of this is supported by our framework.

4 Experimental Validation

Our hypothesis is that the flexibility and composability of the validation part of WebSelF leads to an improvement in accuracy, precision and specificity (defined in Sect. 4.2). We have therefore created a concrete implementation of the framework in Java to test this hypothesis. The implementation includes selection functions using regular expressions [1], XPath expressions, and it includes a full browser in order to let client-side computation take place and provide presentational information about the given web page. Furthermore automatic reinduction of validation functions and composition of validation functions as described in Sect. 3.2 are supported by the implementation. We have used this implementation to perform a substantial experiment which is described in more detail below. For details on the implementation, data set and results we refer to the project homepage at `cs.au.dk/~gedefar/webself`.

4.1 Experimental Setup

In order to evaluate WebSelF in a realistic setting, we collected 30 XPath expressions used as *selection functions*, where most of them were sufficiently successful to be published on the Web. Some of these XPath expressions were complete, concrete paths from the root to the target, while other expressions were more robust paths, such as the expression `//a[starts-with(., 'Next')]`, which selects the next button on the Yahoo Web Shop (by searching for any link starting with "Next"). These more robust expressions used the more advanced operators of XPath, like wildcards, descendant axes, etc. In order to do a proper comparison we created robust versions of the fully concrete paths, and used FireBug[3] to create fully concrete versions of the robust ones. The robust versions were crafted using only knowledge of the first web page version and was guided by the findings of Myllymaki & Jackson [24], meaning that the crafted expressions typically used descendant axis and attribute filters.

For the purposes of our experiment, we normalize all XPath expressions to have the same weight, by letting them return the first selected element if more than one is selected. To evaluate the validation functions directly we fix the selection functions, such that they are not reinduced during the experiment. In total we ended up with 60 XPath expressions, 30 fully specified and 30 robustified.

We have constructed 24 qualitatively different *validation functions* that validates textual, structural, context and presentational properties of the selected element. Eight of these validation functions use a combination of other validation

[3] Available from `http://getfirebug.com`.

functions, such as a conjunction of presentation and content validation functions. In this paper we have included the results from nine of them (see Sec. 4.3). The remaining results can be found on the project web page. This section is devoted to describe these nine validation functions.

validation function	dimension	response	reinduction
Q_{Random}	N/A	random yes/no	N/A
Q_{LMN}	content	text matches pattern	learn token patterns
Q_{DTD}	structure	valid by DTD	infer DTD
Q_{BOX}	presentation	within a rectangle	learn enclosing rectangle
Q_{DTD3}	context	valid by DTD	infer DTD of ancestor
$Q_{\text{BOX}} \wedge Q_{\text{LMN}}$	composite	conjunction	Reinduce failing validation function
$Q_{\text{BOX}} \wedge Q_{\text{LMN}} \wedge Q_{\text{DTD3}}$	composite	conjunction	
$\neg Q_{\text{LMN}}$	composite	negation	
$Q_{\text{BOX}} \wedge \neg Q_{\text{LMN}}$	composite	conjunction, negation	

Fig. 4. The nine described validation functions.

The nine validation functions are summed up in the table of Fig. 4, where the first column states the name of the validation function or its formula if it is a compositional validation function; the second gives the dimension (content, structure, presentation, or composite) of the element, which the validation function relies on; the third gives abstractly what an element is accepted according to; and finally the fourth describes how the reinduction is done, which is of course related to how an element is accepted.

The first five validation functions are basic validation functions that rely on qualitatively different dimensions. Q_{Random} flips a coin to decide whether an element is accepted or not. Q_{LMN} is the validation function introduced by Lerman et al. [10] and it is based upon the textual content of the selected element. Specifically an element is accepted if the text tokens of the selected element is accepted by a pattern learned in the reinduction step. The used pattern is the statistically most significant pattern over the history of examples. For details we refer to Lerman et al. [10]. Q_{DTD} accepts an element if the HTML structure of the element is accepted by an DTD, which is inferred in the reinduction step. For the DTD inference we use the tool by Bex et al. [4]. Q_{BOX} accepts the selected element if the physical position of the selected element is within a rectangle. The rectangle is constructed in the reinduction step, where it infers the smallest enclosing rectangle, that contains all positions in the history. Q_{DTD3} is similar to Q_{DTD}, except the DTD is inferred from the context of the selected element, namely the great grand parent.

The last four validation functions are composite, as described in Sect. 3.2. $Q_{\text{BOX}} \wedge Q_{\text{LMN}} \wedge Q_{\text{DTD3}}$ really showcase the flexibility of our framework as it uses three basic validation functions that are based on three different dimensions of the selected element.

The XPath expressions harvested data from a total of 17 web sites which exhibit considerable diversity, including a TV guide, a blog, an image repository,

price listings, webshops, download sites, search results, and news sites[4]. With the 60 XPath expressions we thus have an average of more than three XPath expressions per site. For each of these sites, we have systematically collected daily versions for a period of one year (from the 24/04 2010 to 1/5 2011), and manually provided a "perfect history" which indicates for each XPath expression whether it selected the correct element. In total 19,664 elements are selected by the selection functions, where 15,843 (81%) are correct selections and 3,821 (19%) are incorrect selections. This data is the starting point of our experiment.

4.2 Evaluation Metrics

When the selection function yields a particular element, there are four outcomes when evaluating validation functions: where the validation function q as well as the human oracle O accept that choice (true positive, TP); where q accepts and O rejects the choice (false positive, FP); where q rejects and O accepts the choice (false negative, FN); and where both reject it (true negative, TN).

There is an inherent asymmetry between FP and FN. Since the user never sees an element accepted by the validation function, FP may be *dangerous* (the scraping program continues with bad data without discovering it) whereas FN is merely *annoying* to the user as it will ask him on an element that is really ok. Thus, it is generally safer for a validation function to answer "too much negative" rather than "too much positive". We will use standard pattern recognition evaluation metrics [20] for evaluating our validation functions, focusing on the ones that involve false positives (**FP**, shown in bold below):

$$\textbf{\textit{accuracy}} = \frac{TP+TN}{TP+TN+FN+\mathbf{FP}}$$

$$\textbf{\textit{precision}} = \frac{TP}{TP+\mathbf{FP}}$$

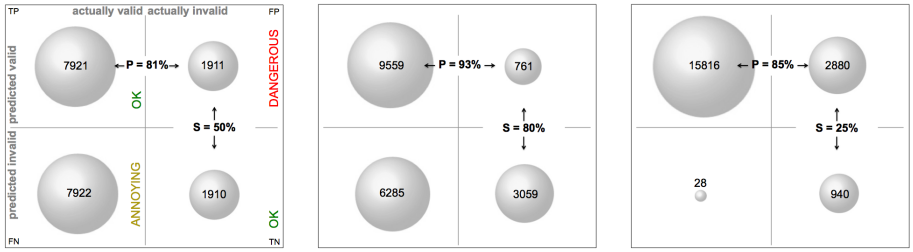
$$\textbf{\textit{specificity}} = \frac{TN}{TN+\mathbf{FP}} \quad \textit{(aka. negative recall)}$$

The *accuracy* measure quantifies "how often q is right"; *precision* is a metric for "how often q is right, when it makes a positive prediction"; and *specificity* quantifies "how often q is right, when the answer is actually negative". (The term *specificity* comes from medical diagnosis; in information extraction, it is often referred to as *negative recall*.)

4.3 Results

Figure 5 shows a graphic depiction of the accumulated results of the nine validation functions applied and reinduced during the year's worth of data. For each validation function, each of the four outcomes (TP, FP, TN, FN) is depicted as a sphere whose three-dimensional volume is proportional to the number of elements in that category. The evaluation metrics are indicated as **P** for precision, **S** for specificity, and **A** for accuracy.

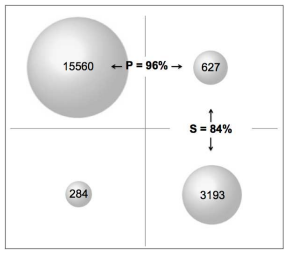
[4] We refer to the project homepage for more information.



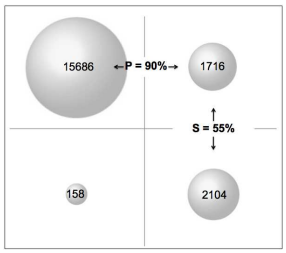
(a) Q_{Random} **A = 50%**

(b) Q_{LMN} **A = 64%**

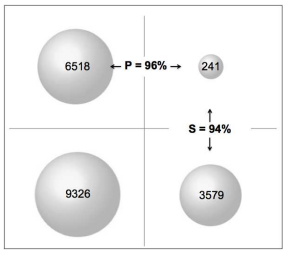
(c) Q_{DTD} **A = 85%**



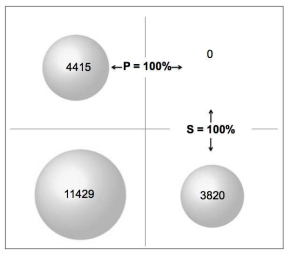
(d) Q_{BOX} **A = 95%**



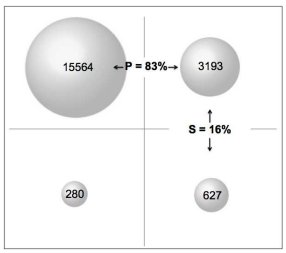
(e) Q_{DTD3} **A = 90%**



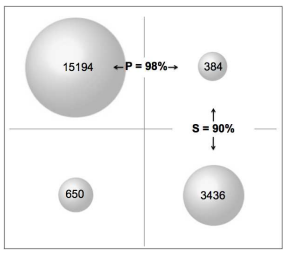
(f) $Q_{\text{BOX}} \wedge Q_{\text{LMN}}$ **A = 51%**



(g) $Q_{\text{BOX}} \wedge Q_{\text{LMN}} \wedge Q_{\text{DTD3}}$ **A = 42%**



(h) $\neg Q_{\text{LMN}}$ **A = 83%**



(i) $Q_{\text{BOX}} \wedge \neg Q_{\text{LMN}}$ **A = 95%**

Fig. 5. Results of using validation functions with different characteristics.

The first figure (Fig. 5(a)) shows the random validation function Q_{Random} and not surprisingly it scores 50% in accuracy. Note that in 81% of the cases where it accepts the element it was correct, just because correctly selected elements are common. This figure serves as a baseline for the performance of the rest of the validation functions.

The next figure (Fig. 5(b)) depicts Q_{LMN}. The accuracy is relatively low because it is too restrictive, i.e., rejects too often. This is seen by the relatively low amount of false positives (761) and high number of false negatives (6,285). The latter is caused by having frequently changing content, such as news articles.

Figure 5(c) shows the result for Q_{DTD}. This validation function is too permissive as the number of false positives is high. There are several reasons for this: Many of the elements that are selected are leaves in the HTML tree, such as an anchor tag. If the XPath expression only selects anchor tags it can be

hard to distinguish a valid element from an invalid one, as there is no internal structure to inspect. A good example is the previous mentioned XPath expression (`//a[starts-with(.,'Next')]`). Most of the time this XPath expression selects the right element, but once in a while an advertisement for `http://nextwarehouse.com/` would show up on the site, and that link would be selected instead. This could not be detected by the validation function, as both anchor tags were anchor tags containing only text.

The following figure (Fig. 5(d)) shows the results of Q_{BOX}. Compared to the previous two validation functions it appears to be in the middle, with regard to false positives and false negatives (presumably since screen coordinates are predictably stable). The number of false positives is mainly caused by one of the XPath expressions, whose purpose is to select a row in a table. The expression uses the row number for the selection, but because the table changes frequently, the correct row jumps up and down in the table, while the selected row is in the same spot in the table throughout the experiment. Hence the presentational features of the selected row is not sufficient to distinguish a correct selection from a wrong selection and therefore the validation function accepts too many selections.

Figure 5(e) depicts the results of Q_{DTD3}. Compared to Q_{DTD} it performs better in all three metrics and this indicates that the contextual dimension is a better guideline for structural validation functions. Still though Q_{DTD3} has a high number of false positives compared to other validation functions.

The next four figures, Fig. 5(f–i) show the different composite validation functions. In Fig. 5(f) we can see that a conjunction of validation functions is, not surprisingly, generally more restrictive than each of its operands(Fig. 5(c,e)), yielding fewer false positives, but at the expense of producing a lot more false negatives. Also, both precision and specificity are higher whereas accuracy suffers from the many false negatives which are also likely to annoy the user. The even more restrictive validation function (Fig. 5(g)) achieves no false positives at all; however, it is at the expense of a very large number of false negatives. In the last two compositional validation functions, Fig. 5(h,i), we have shown the results of using a negation and a negation inside a conjunction. Not surprisingly, using the negation alone on a too restrictive validation function such as Lerman et al., yields a too permissive validation function(Fig. 5(h)). Interestingly, if we take the conjunction of this negated validation function and Q_{BOX}, Fig. 5(i), we get a validation function that performs better than its constituents and in general achieves a high accuracy, a low number of false positives, and a relatively low number of false negatives. Again, like Q_{BOX}, the main source of false positives is the XPath expression using the row number for the selection. If we were to remove that web site from the results, the precision and specificity would become 99.8% resp. 99%. In other words, WebSelF enables significantly improved results in terms of the most important metrics.

We have experimented with disjunction, and performed outlier disqualification (avoiding reinduction on abnormal elements), but none of these validation functions seem to be as promising as either Q_{BOX} nor $Q_{\text{BOX}} \wedge \neg Q_{\text{LMN}}$.

5 Related Work

We have already discussed several pieces of related work, so in this section we focus on a missing perspective, which is the large number of related approaches that would fit very well as the basis for the parameters of WebSelF, namely selection functions, validation functions, or reinduction functions: Kushmerick et al. [9] induce selection functions by finding landmarks in the HTML text. Kistler & Marais [19] uses a markup algebra combining both textual and structural features for selection functions. Cohen & Fan [3] induces selection functions by learning page-independent heuristics and combine them with user interaction. Kushmerick [20] uses textual features of the extracted information for validation. Lerman et al. [10] induce selection functions and validation functions by learning textual patterns and as mentioned in Sect. 2, WebSelF subsumes their approach. The ANDES system [23] uses XPath and XSLT to make selections. SCRAP [8], SG-WRAP [11] and SG-WRAM [12] utilize schemas of the output to guide the induction of selection functions. Liu & Ling [22] extract a conceptual model of the web page upon which the selection is done. Mohapatra et al. [13] induce delimiter based selection functions for a series of web pages with fine grained time resolution. Lingam & Sebastian [21] uses a visual interface to label examples, from which they induces selection and validation functions based on different heuristics. Finally, Dalvi et al. [7] and Parameswaran [16] use a tree edit distance to induce selection functions and validation functions.

6 Conclusion

We have presented WebSelF, a web selection framework that enables the use of existing techniques for selection and validation of selection results, as well as reinducing both of those functions with a carefully minimized amount of assistance from a human being. We have furthermore shown how to compose validation functions based on propositional logic, whereby the validation in WebSelF can benefit from using several dimensions of the selection result. Moreover, we have implemented the framework and performed a substantial experiment involving 11.000 web pages from several diverse web sites over a period of more than one year, based on selection functions successful enough to be published on the web. The experiment shows how validation functions can focus on very different dimensions of the selection result, including contents, structure, and presentation. It also illustrates how the extraction behavior can be tailored according to the needs of the situation. For instance, we may accept an increase in the number of false negatives in order to make sure that we spot almost all false positives, etc. The experiment also shows that by composing several validation functions it is possible to perform better than each of the constituents, and that it is possible to perform better than the previous approach by Lerman et al. [10]. In summary, WebSelF provides a well-understood platform for the exploitation of a large space of possibilities in the choice and combination of selection functions, validation functions, and reinduction.

7 Acknowledgments

We thank Kristina Lerman for quickly responding to our numerous questions regarding their implementation, Mathias Schwarz for constructive comments on an earlier version of the paper and the anonymous reviewers for valuable feedback.

References

1. Brabrand and Thomsen. Typed and unambiguous pattern matching on strings using regular expressions. In *In Proc. of PPDP*, 2010.
2. Cohen. Recognizing structure in web pages using similarity queries. In *AAAI/IAAI*. AAAI, 1999.
3. Cohen and Fan. Learning page-independent heuristics for extracting data from web pages. *CN*, 31(11-16), 1999.
4. Bex et al. Inference of concise DTDs from XML data. In *Proc. of VLDB*, 2006.
5. Bray et al. DTD: Document type definition. World Wide Web Consortium, `http://www.w3.org/TR/xml/#sec-prolog-dtd`, Nov 1996.
6. Chang et al. A survey of web information extraction systems. *TKDE*, 2006.
7. Dalvi et al. Robust web extraction: an approach based on a probabilistic tree-edit model. In *Proc. of SIGMOD*, 2009.
8. Fazzinga et al. Schema-based web wrapping. *KAIS*, 2009.
9. Kushmerick et al. Wrapper induction for information extraction. In *IJCAI*, 1997.
10. Lerman et al. Wrapper maintenance: A machine learning approach. *JAIR*, 2003.
11. Meng et al. Schema-guided data extraction from the web. *JCST*, 17(4), 2002.
12. Meng et al. Schema-guided wrapper maintenance for web-data extraction. In *Proc. of WIDM*, 2003.
13. Mohapatra et al. Efficient wrapper reinduction from dynamic web sources. In *Proc. of WI*. IEEE Computer Society, 2004.
14. Muslea et al. Hierarchical wrapper induction for semistructured information sources. *AAMAS*, 4(1), 2001.
15. Nakatoh et al. Automatic generation of deep web wrappers based on discovery of repetition. In *Proc. of AIRS*, 2004.
16. Parameswaran et al. Optimal schemes for robust web extraction. In *Proc. of VLDB*, 2011.
17. Raposo et al. Automatic wrapper maintenance for semi-structured web sources using results from previous queries. In *Proc of. SAC*, 2005.
18. Thomsen et al. WebSelf: A web selection framework. Tech. report, Computer Science, Aarhus University, 2012.
19. Kistler and Marais. Webl - a programming language for the web. *CN*, 30(1-7), 1998.
20. Kushmerick. Wrapper verification. *WWW*, 2000.
21. Lingam and Elbaum. Supporting end-users in the creation of dependable web clips. In *WWW*, 2007.
22. Liu and Ling. A conceptual model and rule-based query language for HTML. *WWW*, 2001.
23. Myllymaki. Effective web data extraction with standard XML technologies. *CN*, 39(5), 2002.
24. Myllymaki and Jackson. Robust web data extraction with xml path expressions. *IBM Research Report*, RJ10245, 2002.
25. Sahuguet and Azavant. Building intelligent web applications using lightweight wrappers. *DKE.*, 36(3), 2001.