

# "Web as Data Source"

<sup>1</sup>Lokhande Dheeraj Bhimrao, <sup>2</sup>Rajesh V. Argiddi, <sup>3</sup>S. S. Apte

<sup>1,2,3</sup>Dept. of CSE, Walchand Institute of Technology, Solapur, Maharashtra, India

## Abstract

With the phenomenal growth of the WWW, rich data sources on many different subjects have become available online. Some of these sources store daily facts that often involve textual geographic descriptions. These descriptions can be perceived as indirectly geo referenced data - e.g., addresses, telephone numbers, zip codes and place names. Under this perspective, the Web becomes a large geospatial database, often providing up-to-date local or regional information. In this work we focus on using the Web as an important source of urban geographic information and propose to enhance urban Geographic Information Systems (GIS) using indirectly geo referenced data extracted from the Web. We describe an environment that allows the extraction of geospatial data from Web pages, converts them to XML format, and uploads the converted data into spatial databases for later use in urban GIS. The effectiveness of our approach is demonstrated by a real urban GIS application that uses street addresses as the basis for integrating data from different Web sources, combining these data with high-resolution imagery.

## Keywords

Web Scraping, Semantic Web, XPath

## I. Introduction

According to Tim Berners-Lee the World Wide Web is evolving from a web of documents into a web of data. Companies such as Amazon1 and Yahoo have recognized the advantage of this and defined website APIs to add value to their service. However the bulk of the web's data is obscured in (X)HTML by a layer of presentation, with different styles for each website. Additionally, these styles change over time as each website is updated with additional content or a new layout. This makes working with data across websites cumbersome.

This paper describes a software that aims to address this problem. This is a typical workflow with Our software:

1. A user wants to access city temperatures directly from bom.gov.au (the Australian Bureau of Meteorology)
2. They find a few web pages with city temperatures at bom.gov.au and give Our software the URLs and the temperatures displayed in each as a list of strings.
3. Our software retrieves and parses the webpages and finds where the temperatures strings are located in the HTML source.
4. These locations are examined and a model of the general pattern of occurrence is created that will locate the weather strings.
5. Now the user can provide a bom.gov.au URL and Our software will automatically apply the model to scrape the temperatures, e.g. on an hourly basis to generate a temperature graph from.

Throughout this process the user does not need to concern herself with the structure of a webpage but can focus in-stead on the content. The penalty for this ease of use is that Our software can only reliably be applied to invariant data with variant structure. Fortunately database-backed websites that inject their content into a common template are popular, so this restriction is not unreasonable.

The original motivation for Our software was for use in the ILIAD project [1], which is an attempt to enhance information access over Linux troubleshooting-related threads from a range of web forums. We originally created a set of Perl scripts to scrape the data, with a different script for each forum. This got us the data we wanted but was time consuming to create and didn't provide an immediate solution for scraping additional forum data from new sites.

The envisioned ILIAD system would also need to periodically recrawl these forums for new content. This presents the problem of forums updating their structure and breaking our scripts, which would then need to be manually fine-tuned. We wanted to automate ILIAD as much as possible and make it easier to scrape new websites and possible to deal with changing webpage structures. In developing Our software, we identified three ways in which a web page set (i.e. the data served from a given domain or set of URLs) can be updated:

- The content changes while the structure stays the same (e.g. a weather site with different temperatures each day displayed in the same template)
- The content stays the same while the structure changes (e.g. a web forum where the discussion has ended but the layout is updated)
- Both the content and structure change (e.g. an online shop with new products and a new layout to make shopping easier)

Scrapers generally focus on the first case of changing content with fixed structure. The second case of changing structure is where most scrapers will fail but Our software will thrive. In the third case where both content and structure change, Our software offers a partial solution for small changes for monotonically increasing data. Even in the instance of the content changing too significantly for software to generalise over, it is generally possible for a non-expert to annotate new data and retrain a model within minutes. Our software is simple to train, can handle changing web content and/or changing structure, is open source, and is written in Python so it is platform independent and easy to distribute.

## II. Related Work

Many people have the need to extract data from the web and so a number of scraping tools are already available. This section will survey the features of some of the most well-known open source web scrapers available and explain why we felt the need to develop another one.

### A. Chickenfoot

Chickenfoot [3] is a Firefox extension that adds high-level functions to Javascript that can be executed in the browser. Embedding in the browser makes Chickenfoot easy to distribute, and additionally supports interaction with Javascript. Our software only examines the raw HTML so it is unable to interpret the effect of Javascript events or AJAX calls. This turned out not to be an issue in our data, as all of our websites render properly without Javascript. However this may be a consequence of us choosing popular sites which are expected to be better engineered than average to cater for a larger audience. The drawbacks of embedding in the browser are the limitations imposed by the environment. We found

Chickenfoot to run very slowly, which may be a consequence of running in the browser, and would make scraping the amount of data we are interested in (millions of threads) impractical.

Chickenfoot is primarily aimed at interaction with the browser but can also be used for scraping with the find() command. Here is an example script for scraping search results from a Google search:

```
go("www.google.com")
enter("chickenfoot")
click("Google Search")
for(m=find("link"); m.hasMatch; m=m.next) {
  var link = m.element;
  if(link.getAttribute("class") == "l") {
    output(link.href);
  }
}
```

This script searches Google for chickenfoot and returns the links that have a class of l, which from examining the Google HTML source is an attribute unique to the search result links. The Chickenfoot functions are very high level so this is perhaps a better solution than our original Perl scripts, but it is still dependant on the structure of the webpage and so does not solve the problem of dealing with changing webpage structure. Additionally, the scripts require direct analysis of the HTML mark-up, and thus require expert knowledge.

## B. Piggy Bank

Piggy Bank [4] is a Firefox extension that aims to be a bridge between the semantic web and what we have now.

The idea is that users submit web scraping scripts along with a regular expression for the URLs it is relevant to.

Then when the user navigates to a matching webpage PiggyBank displays the scraped semantic data. It is a fine idea if the work for creating and maintaining scraping scripts could be distributed around the world, but unfortunately the community is not there yet, so Piggy Bank does not solve our scraping problem. At the time of writing only eleven scripts have been submitted.

## C. Sifter

Sifter [5] builds on top of Piggy Bank's infrastructure but tries to scrape semantic data automatically from any webpage. However the scraper has limited scope and only looks for the biggest group of links in a webpage. This is relevant to a commerce site like Amazon where the books are a series of links, but usually we will not want to extract the biggest group of links. For instance the biggest group of links in a web forum is generally navigation-related and not directly relevant to a given thread in isolation. Consequently, Sifter does not solve our scraping problem.

## D. Scrubyt

Scrubyt5 is a Ruby library that provides the most similar functionality to our software of the tools surveyed.

Scrubyt can be given an example string and will then locate the string in a webpage and extract all similar items from the webpage. This is similar to Sifter's goal of extracting product lists but Scrubyt allows control over what group to extract and is not limited to links.

Here is the Scrubyt version of the Chickenfoot example to scrape Google search results:

```
google_data = Scrubyt::Extractor.define do
  fetch "http://www.google.com/ncr"
  fill_textfield "q", "ruby"
  submit
  link "Ruby Programming Language" do
    url "href", :type => :attribute
  end
end
```

This script searches Google for ruby and then uses the known title for the official Ruby website to automatically build a model of the search results. It then extracts the links from this model.

This is a big improvement from the Chickenfoot example because it is independent of the webpage structure (apart from specifying q as the name of the search box). However the results are mediocre. In our test this script only returns the first three links because the search results are then interrupted by a YouTube video. Scrubyt, like Sifter, can only handle contiguous lists, which limits its application. Because of this limitation Scrubyt could also not scrape the Linux Questions web forum because the posts were separated by titles and user data. Scrubyt can survive a structural update because it is a content-based scraper, however the types of scraping possible were too limited for our use.

## E. Template Maker

Template Maker 7 is a Python library that takes a different approach to any of the other tools surveyed. Like Our software, TemplateMaker is first trained over a set of example webpages. However unlike Our software, TemplateMaker does not require their associated text chunks. Instead, TemplateMaker examines the differences between the HTML of each webpage to determine what content is static and what is dynamic. The dynamic content is assumed to be what is interesting in a webpage and what the user wants to extract, so TemplateMaker generates a model to scrape this data. Here is an example of how TemplateMaker is used:

```
>>> from templatemaker import Template
>>> t = Template()
>>> t.learn("<b>David and Richard</b>")
>>> t.learn("<b>1 and 2</b>")
>>> t.as_text("")
"<b>[] and []</b>"
>>> t.extract("<b>Richard and Tim</b>")
("Richard", "Tim")
```

In this example TemplateMaker was able to automatically model the dynamic content and successfully extract Richard and Tim from the test string. This simple process makes TemplateMaker the easiest tool to train in this survey. TemplateMaker works well for trivial strings like those given in the example. However we found it does not scale for larger strings from real webpages. When we tried modeling a LinuxQuestions thread with TemplateMaker, the script stalled for a few minutes before throwing a regular expression exception for trying to match too many terms. To avoid this exception we then tried a simpler hand-crafted set of web pages and from this TemplateMaker managed to return a third of the dynamic data. Scraping just a third of the data is poor performance for a relatively simple web page.

From examining the generated model we found the reason for the poor performance was that TemplateMaker did not handle duplicates well. For instance in our example webpage

when TemplateMaker was comparing the strings Richard and Tim it interpreted that the second character i was static while the surrounding text was dynamic, and so the generated model looked for content surrounding an i. The fundamental problem is TemplateMaker aims to be content neutral and so treats its input as a series of characters. Consequently it cannot use the HTML structure to determine that the text blocks for Richard and Tim should be treated as a single unit.

TemplateMaker is an interesting tool that takes a novel approach to web scraping. TemplateMaker only requires example URLs to train, so it could easily be automatically retrained after a website structural update. However through our tests we found that TemplateMaker is unsuitable for scraping large documents because of performance. We also found it too brittle in its handling of duplicates to be used reliably for web scraping. Of these tools only Scrubyt and TemplateMaker can address the problem of changing webpage structure. However neither of these tools are flexible enough for our scraping needs. Additionally none of the tools surveyed take advantage of features from examining a set of similar web pages. For these reasons we felt justified building another general purpose web scraping tool.

### III. Obtaining Spatial Information from Web Sources

To make it possible to create an environment to integrate Web pages to spatial location information, we had to meet several challenges. The first was to extract indirectly geo referenced data in textual form (such as postal addresses) from the contents of Web pages. We stress that such information, when available, is implicit and occurs as any other ordinary string mixed with HTML markup. In GIS, the process of recognizing geographic context is referred to as geoparsing, and the process of assigning geographic coordinates is referred to as geocoding. This section discusses the efforts to geoparse and to geocode Web pages.

The extracted addresses act as keys to the geocoder. The second challenge was to establish ways for transforming the extracted spatial location information in the form they are provided by the generic public to the form they are stored in a typical GIS. After that, a set of geographic coordinates corresponding to the addresses can be obtained, using an address matching function. Finally, the extracted information was inserted into the GIS database and superimposed on high-resolution imagery or maps.

The basic procedure for our application is:

- To crawl Web sites to collect the pages containing data of interest.
- To geoparse the collected pages to extract geographic indication and the relevant data.
- To make the data available in an suitable format (in our case, XML).
- To geocode the addresses into a coordinate system.
- To update the GIS database and, finally
- To integrate information from several geospatial data.

The resulting system can be used by municipalities, users with some kind of urban GIS database, or geographic database designers.

This section describes how step 2 can be accomplished by deploying the DEByE (Data Extraction By Example) example-based approach to automatically extract semistructured data. This approach is more convenient for our application because it lets the user specify a target structure for the data to be extracted. Furthermore, the user might be interested in only a subset of the information encoded in the page. Moreover, DEByE does not require the user to describe the inherent structure of a whole page.

### A. The DEByE Tool

DEByE is a tool that has been developed by the UFMG Database Group to generates wrappers for extracting data from Web pages. It is fully based on a visual paradigm, which allows the user to specify a set of examples for the objects to be extracted. Example objects are taken from a sample page of the same Web source from which other objects (data) will be extracted. By examining the structure of the Web page and the HTML text surrounding the example data, the tool derives an Object Extraction Pattern (OEP), a set of regular expressions that includes information on the structure of the objects to be extracted and also the textual context in which the data appear in the Web pages.

The OEP is then passed to a general-purpose wrapper that uses it to extract data from new pages in the same Web source, provided that they have structure and contents similar to the sample page, by applying regular expressions and some structuring operations.

DEByE currently operates as a Web service, to be used by any application that wishes to provide data extraction functionality to end users. For general data extraction solutions, a DEByE interface based on the paradigm of nested tables is used, which is simple, intuitive, and yet powerful enough to describe hierarchical structures that are very common in data available on the Web. The sample pages are displayed in the upper window, also called the source window. The lower window, also called the table window, is used to assemble example objects. The user can select pieces of data of interest from the source window and paste them on the respective columns of the table window. After specifying the example objects, the user can select the Generate Wrapper button to generate the corresponding OEP, which encompasses structural and textual information on the objects present in the sample page. Once generated, this OEP is used by an extractor module that will perform the actual data extraction of new objects and then will output them using an XML-based representation. DEByE is also capable of dealing with more complex objects, by using a so-called bottom-up assembly strategy, explained in [14]. Fig. shows a snapshot of a user session with an example object in the lower window and the extracted objects showed in HTML format in the upper window. Fig. presents an overview of the whole DEByE approach. The two modules called Graphical User Interface (GUI) and Extractor compose the DEByE tool.

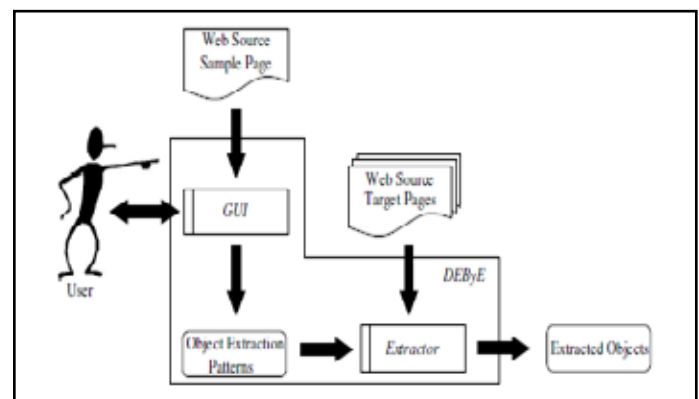


Fig. 2: Modules of the DEByE Tool and Their Role in the Data Extraction Process

### IV. Methodology

In this section, we present the full pipeline architecture of our software, from retrieving and parsing each seed and test document, to identifying the document extents within each seed document that match with the text chunk(s), and generating the model in

the form of a generalised XPath describing the positions of the text chunks in the seed documents.

To help illustrate the stages involved in this process we will use the simple HTML document in Figure as an ongoing seed document example. Assuming that the user is interested in scraping only the temperatures for tomorrow across the three cities, the list of text chunks would be <22,22,26>.

### A. Parse

The first step is to parse each HTML document into a form which is more amenable to both positional indexing (seed and test documents) and pattern generation (seed documents only). This takes the form of partitioning the document into individual nodes, as defined by the elements in the (X)HTML structure, and identifying the text associated with each. The underlying assumption here is that each text chunk follows element boundaries, and that we simply need to identify the relative position of the element which best defines the extent of each text chunk. Note that the elements form a hierarchy relative to the nesting of the HTML mark-up, and that both internal and terminal elements are indexed for their text content.

To perform the element partitioning, we chose the Python lxml module,<sup>8</sup> which uses BeautifulSoup<sup>9</sup> to resolve bad mark-up and then stores the results in a tree using the Element Tree module.<sup>10</sup> We chose to use ElementTree over

BeautifulSoup's native representation because it supports more powerful searching and traversing. To make string matching easier in later stages we prune this tree to remove content that is not directly displayed in the browser and so cannot be selected by the user, such as Javascript functions and meta tags (of which there are none in our example).

Now we have the parsed HTML document stored in an Element Tree, for each seed document, we create a reverse indexed hash table with strings as keys and element paths as values for efficiency in the later stages of processing. To represent the matching locations we use XPath, which is an XML selection language defined by the W3C.<sup>11</sup> XPath can be used to match a single element or generalised to match a set of elements, and is well supported by our chosen XML library lxml. The hash table for our example weather web page, noting the nesting of elements (e.g. Melbourne 20 22 vs. Melbourne) and also the occurrence of some strings in multiple locations (e.g. 22).

### B. Search

The second step is to identify the element which contains each text chunk associated with the corresponding seed document. Unfortunately a direct string query into our string hash table will in general not work, as the input is a list of text chunks copied from a rendered web page while Our software operates over the original HTML. For instance, the heading in the example webpage HTML is Weather forecast but in the browser the end-of-line characters (nn and nr) are ignored and the heading becomes just Weather forecast. Another case is when XML characters such as &lt; are used in the HTML, which will be rendered as < by the browser. Additionally, the hash table created in the previous step indexes the string associated with each element, but the user may not copy all the text within a tag but just a subset. In performing this search, we use the elements identified by the document parser in the first step of processing, and generate a lattice, comparing the string associated with each seed document element, with each of the text chunks associated with that document. As such, the granularity of the sub-document strings we compare each text chunk to is defined by the document mark-up.

To calculate the similarity between strings we initially tried using the basic Longest Common Substring (LCS) algorithm, which finds the longest common sequence between two strings. Ultimately, however, we found this method unsuitable in its original form, as even a valid match in our data may have extra characters embedded in it (such as newlines) that would break up the matching substring. As a result, we developed a scoring mechanism based on the output of the LCS algorithm, by first finding all the matching and non-matching substrings with the Python difflib module<sup>12</sup>. We then square the substring lengths (to bias towards longer substrings) and deduct the total non matching lengths from the matching lengths; all lengths are calculated in characters. This result is then normalised by dividing by the square of the sum of all the matching and non-matching lengths so that scores from different strings can be compared meaningfully. In summary, the score returned by our modified LCS algorithm become.

### V. Results

While developing software we fine-tuned the model generation to work well on the development sites, and were thus predictably able to achieve a high macro-averaged F-score of 0.98. When we applied Our software to the blind test sites, we were very encouraged to find that the macro-averaged F-score was almost identical at 0.97. From the experience of annotating we noticed

```
<html>
<body>
<span class="heading">
Weather
forecast
</span>
<table>
<tr>
<th>City</th>
<th>Today</th>
<th>Tomorrow</th>
</tr>
<tr>
<td>Melbourne</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>Sydney</td>
<td>25</td>
<td>22</td>
</tr>
<tr>
<td>Adelaide</td>
<td>26</td>
<td>26</td>
</tr>
</table>
</span></span>
</body>
</html>
```

Fig. 2: Example HTML Document (Weather Forecast, Doc1)

that usually when software made a mistake the cause was a variation in the webpage structure from the examples used to generate the model. And generally the more complex chunk types suffered more variation, which contributed to their lower performance. Software performed perfectly over the simplest case, type 1, achieving a precision and recall of 1.00 across both the development and training sets. These websites had no variation and so were easy for Our software to scrape. Types 2, 3, and 5 performed very well, with the lowest F-score being 0.95 for The Onion where the author details were incorrectly included when scraping certain articles. Type 6 had the lowest overall F-score as a group, with Linux Questions producing the lowest F-score in the development set. The main reason for this is that in some Linux Questions threads the responder would embed a code snippet within a sub-tag. The Linux Questions model failed to scrape this embedded code snippet because this special case was not present in the 3 seed documents used to generate the model. If we had been more careful in our choice of the seed documents, or just used a larger seed set, the performance over Linux Questions could have been improved without any change to software. To test this hypothesis we retrained the model for Linux Questions with a larger example set of six seed documents that all contained quotes in their thread posts, and then re annotated 20 web pages with this new model. As expected the model could now scrape the quotes, and as a result the recall jumped from 0.89 to 1.00. However the new model included some non-post data which made the precision fall from 1.00 to 0.95. Overall the F-score increased from 0.89 to 0.94, suggesting that this is a more balanced model and the problem was largely one of not enough data. Recall that the only manual analysis of the seed documents that was required was the user manually copying and pasting relevant text chunks into a text field (or to the command line), such that the increase from 3 to 6 seed documents still represents a minuscule amount of user effort.

## VII. Conclusion

Our software has met our goals to make web scraping easy and automatic retraining possible. It has proved a convenient tool for extracting data from the web. Our approach, based on learning patterns using XPath, allowed us to produce a system that can satisfy user needs with high precision and recall with minimal training. Our software has been tested over different domains with high effectiveness, and we also showed its adaptability by going back in time and scraping the Linux Questions site from 2002 with out re-annotating. On this evidence, we believe that this software can provide a robust and flexible solution for the problems of dealing with web data.

## References

- [1] T. Baldwin, D. Martinez, R. Penman, "Automatic thread classification for linux user forum information access", In Proceedings of the Twelfth Australasian Document Computing Symposium (ADCS 2007), pp. 72-79, Melbourne, Australia, 2007.
- [2] T. Berners-Lee, M. Fischetti, "Weaving the Web", Harper One, San Francisco, USA, 1999.
- [3] M. Bolin, M. Webber, P. Rha, T. Wilson, R. Miller, "Automation and customization of rendered web pages", In UIST '05: Proceedings of the 18th Annual ACM symposium on User Interface Software and Technology, pp. 163-172, New York, USA, 2005.

- [4] D. Huynh, S. Mazzocchi, D. Karger, "Piggy bank: Experience the semantic web inside your web browser", Web Semantics: Science, Services and Agents on the World Wide Web, 5, pp. 16-27, 2006.
- [5] D. Huynh, R. Miller, D. Karger, "Enabling web browsers to augment web sites' filtering and sorting functionalities", In UIST '06: Proceedings of the 19th Annual ACM symposium on User Interface Software and Technology, pp. 125-134, New York, USA, 2006.
- [6] M. Schrenk, "Webbots, Spiders, and Screen Scrapers", No Starch Press, San Francisco, USA, 2007.
- [7] A. Sugiura, Y. Koseki, "Internetscrap book: automating web browsing tasks by demonstration", In UIST '98: Proceedings of the 11th annual ACM Symposium on User Interface Software and Technology, pages 9{18, New York, USA, 1998. 16 [Online] Available: <http://wtr.rubyforge.org/>