Web Scraping Made Simple with SiteScraper

Richard Baron Penman
The University of Melbourne
Victoria, Australia
rbp@csse.unimelb.edu.au

Timothy Baldwin
The University of Melbourne
Victoria, Australia
tim@csse.unimelb.edu.au

David Martinez
The University of Melbourne
Victoria, Australia
davidm@csse.unimelb.edu.au

ABSTRACT

The web contains a huge amount of data, but most is not in an immediately machine-readable form for indexing or semantic processing. Web scrapers are conventionally used to extract data from web documents, by parsing these and extracting out data points relative to their structure. Document mark-up is often volatile, however, and each change to the mark-up of a given site (e.g. the addition of advertisements, or the adoption of a new format) will mean the scraper needs to be manually updated. Our tool SiteScraper gets around this issue by automatically learning XPath-based patterns to identify where a user-defined list of strings occurs in a given web page set. To train, SiteScraper is given a small set of example URLs from a given website and the strings that the user wishes to scrape from each. This is used to generate an XPath query describing where to find the desired strings, which can be applied to scrape these from any webpage with a similar structure. Importantly, the user interacts with SiteScraper at the level of content, not mark-up, so no specialist knowledge is required, and if the structure of a website is changed but the content stays constant, then SiteScraper can automatically retrain its model without human intervention. In our tests SiteScraper attained an average precision of 1.00 and an average recall of 0.97 over 700 webpages across a range of popular websites.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Abstracting methods; H.2.8 [**Information Systems**]: Data mining

General Terms

web scraping, XPath

Keywords

web scraping, semantic web

1. INTRODUCTION

According to Tim Berners-Lee the World Wide Web is evolving from a web of documents into a web of data [2]. Companies such as Amazon[1] and Yahoo[2] have recognised 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 tool called SiteScraper[3] that aims to address this problem. This is a typical workflow with SiteScraper:

- A user wants to access city temperatures directly from `bom.gov.au` (the Australian Bureau of Meteorology).
- They find a few webpages with city temperatures at `bom.gov.au` and give SiteScraper the URLs and the temperatures displayed in each as a list of strings.
- SiteScraper retrieves and parses the webpages and finds where the temperatures strings are located in the HTML source.
- These locations are examined and a model of the general pattern of occurrence is created that will locate the weather strings.
- Now the user can provide a `bom.gov.au` URL and SiteScraper 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 instead on the content. The penalty for this ease of use is that SiteScraper 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 SiteScraper 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 re-crawl these forums for new content. This presents

[1] `http://aws.amazon.com/associates/`
[2] `http://developer.yahoo.com/`
[3] `http://code.google.com/p/sitescraper/`

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 SiteScraper, 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:

1. The content changes while the structure stays the same (e.g. a weather site with different temperatures each day displayed in the same template)
2. The content stays the same while the structure changes (e.g. a web forum where the discussion has ended but the layout is updated)
3. 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 [6]. The second case of changing structure is where most scrapers will fail but SiteScraper will thrive. In the third case where both content and structure change, SiteScraper offers a partial solution for small changes for monotonically-increasing data (see Section 7). Even in the instance of the content changing too significantly for SiteScraper to generalise over, it is generally possible for a non-expert to annotate new data and retrain a model within minutes.

SiteScraper 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.

In the remainder of this paper, we survey the major public-domain scrapers (Section 2), and show a brief tutorial to illustrate the use of SiteScraper (Section 3). Then we describe the technical details of SiteScraper (Section 4), and present an extensive evaluation of SiteScraper's considerable utility over a number of domains (Sections 5 and 6). Finally, we discuss the implications of this work, and future directions for development efforts (Section 7), before concluding the paper (Section 8).

2. PREVIOUS 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.

2.1 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. SiteScraper only examines the raw HTML so it is unable to interpret the effect of Javscript 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.

2.2 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 Piggy Bank 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.[4]

2.3 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.

2.4 Scrubyt

Scrubyt[5] is a Ruby library that provides the most similar functionality to SiteScraper 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

[4] http://simile.mit.edu/wiki/Category:Javascript_screen_scraper
[5] http://scrubyt.org/

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 LinuxQuestions 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.

2.5 WWW-Mechanize

WWW-MECHANIZE[6] is a Perl library for simulating a web browser session. It supports cookies, form filling, link navigation, but, like SITESCRAPER, not Javascript. It does not support high-level scraping itself but is often used as part of a scraping solution because of its ability in navigating a website.

2.6 TemplateMaker

TEMPLATEMAKER[7] is a Python library that takes a different approach to any of the other tools surveyed. Like SITESCRAPER, TEMPLATEMAKER is first trained over a set of example webpages. However unlike SITESCRAPER, 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")
```

[6]http://search.cpan.org/dist/WWW-Mechanize/
[7]http://code.google.com/p/templatemaker/

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 webpages 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 webpage.

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 can not 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 webpages. For these reasons we felt justified building another general-purpose web scraping tool.

3. A BRIEF SITESCRAPER TUTORIAL

Before launching into the implementation details let us present a description of SITESCRAPER usage in the form of a simple command line interaction, as shown in Figure 1.

First, SITESCRAPER requires a small sample set of **seed documents**, each of which is paired with a list of **text chunks**. The text chunks associated with a given seed document represent the data points in that document that the user wishes to scrape, and are intended to be copied from the browser-rendered version of the document. The first three lines in Figure 1 specify two seed documents (with fictional URLs) and their associated text chunks. Note that while in this example the number of text chunks is fixed across the documents, this is not a requirement of the system, and wouldn't be the case, e.g. when specifying the individual posts in variable-length user form threads.

Second, the user imports SITESCRAPER and generates a model from the set of paired seed documents and text chunks.

Finally, the user applies the trained model to a **test docu-**

```
>>> input_1 = "bom.gov.au/1", ["22", "22", "26"]
>>> input_2 = "bom.gov.au/2", ["19", "25", "24"]
>>> input = input_1, input_2
>>>
>>> import sitescraper as ss
>>> model = ss.trainModel(input)
>>>
>>> ss.applyModel("bom.gov.au/1", model)
["22", "22", "26"]
>>> ss.applyModel("bom.gov.au/3", model)
["24", "29", "28"]
```

Figure 1: Example command line interaction with SiteScraper

ment by simply specifying the URL and the trained model. SiteScraper returns a list of text chunks identified in the test document. In Figure 1, the user can be seen to have reapplied the trained model to one of the seed document URLs to confirm the consistency of the model, and then to an unseen test document from the same site.

Note that while this example is via the command line and requires basic knowledge of Python, we have also developed a simple web interface to enable less programming-inclined users to use the system easily.

4. METHODOLOGY

In this section, we present the full pipeline architecture of SiteScraper, 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 2 as an ongoing seed document example (a rendered version is presented in Figure 3). 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*⟩.

4.1 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,[8] which uses BeautifulSoup[9] to resolve bad mark-up and then stores the results in a tree using the ElementTree module.[10] We chose to use ElementTree over

```
<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>
```

Figure 2: Example HTML document (weather forecast, $\mathbf{Doc_1}$)

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 can not 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 ElementTree, for each seed document, we create a reverse-indexed hashtable 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.[11] 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 hashtable for our example weather webpage is shown in Table 1, 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*).

4.2 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 hashtable will in general not work, as the input is a list of text chunks copied from a rendered webpage while SiteScraper operates over the original HTML. For instance, the heading in the example webpage HTML is $Weather\backslash n\backslash rforecast$ but in the browser the end-of-line characters ($\backslash n$ and $\backslash r$) are ignored and the heading becomes just *Weather forecast* (see Figure 3). Another case is when XML

[8] http://codespeak.net/lxml/
[9] http://www.crummy.com/software/BeautifulSoup/
[10] http://effbot.org/zone/element-index.htm
[11] http://www.w3.org/TR/xpath

Key	Value
Weather\n\rforecast	`/html/body/span[1]`
City	`/html/body/table/tr[1]/th[1]`
Today	`/html/body/table/tr[1]/th[2]`
Tomorrow	`/html/body/table/tr[1]/th[3]`
City Today Tomorrow	`/html/body/table/tr[1]/td`
Melbourne	`/html/body/table/tr[2]/td[1]`
20	`/html/body/table/tr[2]/td[2]`
22	`/html/body/table/tr[2]/td[3]`, `/html/body/table/tr[3]/td[3]`
Melbourne 20 22	`/html/body/table/tr[2]/td`
26	`/html/body/table/tr[4]/td[2]`, `/html/body/table/tr[4]/td[3]`

⋮

Table 1: A fragment of the location hashtable for Doc$_1$

Figure 3: Rendered web page for Doc$_1$

characters such as `<` are used in the HTML, which will be rendered as `<` by the browser. Additionally, the hashtable 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[12]. 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 becomes:

$$\frac{\sum_i \textit{matching length}_i^2 - \sum_i \textit{non-matching length}_i^2}{(\sum_i \textit{length}_i)^2}$$

The resulting similarity score is in the interval $[-1, 1]$, with the majority less than zero because most string pairs will have more non-matching parts than matching.

In the case of us matching the heading strings of *Weather\n\rforecast* from the HTML and *Weather forecast* from the browser, the score would be as follows:

matching = {Weather, forecast}
non-matching = {\n\r}
similarity score $= \frac{7^2+8^2-2^2}{17^2} = 0.377$

A score of 0.377 in this case shows that the strings have more matching parts than non-matching. These strings are not a perfect match (because of the newline characters) but are certainly the closest match in the document for *Weather forecast*.

Here is the result of comparing the heading with an unrelated string of *20* for today's temperature in Melbourne:

matching = {}
non-matching = {Weather forecast, 20}
similarity score $= \frac{0^2-16^2-2^2}{18^2} = -0.802$

The score of -0.802 is close to the minimum of -1, which shows these strings are highly unrelated.

Locating the *Weather forecast* heading in the HTML is straightforward in this example because all the other strings are temperatures and clearly unrelated. However if we try to locate today's temperature in Adelaide, *26*, then we face the problem of ambiguity. From the hashtable in Table 1, we can see that *26* has two matching locations for today and tomorrow in Adelaide so if SiteScraper tries to build a model based on just this single example then it will not know which location to pick. To help disambiguate in cases such as this, it is best to train SiteScraper over a number of seed documents.

Assume we further provided SiteScraper with the second seed document shown in Figure 4 and the text chunk list $\langle$*19,25,24*$\rangle$. In this second example, the third text chunk (TC$_3$ = *24*) uniquely identifies tomorrow's temperature Adelaide and provided the basis for disambiguating the third text chunk in our first example. Conversely, the combination of the first and second text chunks for the first and

[12] `http://docs.python.org/library/difflib.html`

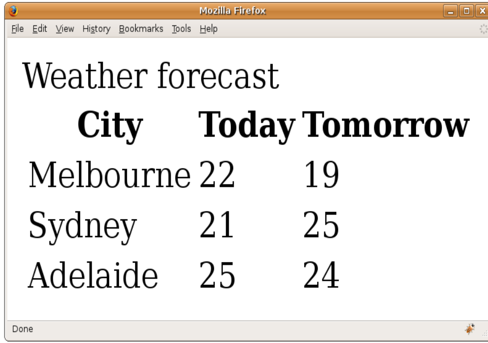

Figure 4: Rendered web page for Doc_2

Similarity	Doc_1	Doc_2	Total
Melbourne tomorrow	1	0	1
Sydney tomorrow	1	1	2
Adelaide today	0	1	1

Table 2: Similarity results for different positions in Doc_1 and Doc_2 for TC_2 (tomorrow's temperature in Sydney, i.e. *22* and *25*, resp.)

second seed documents allows us to disambiguate the positions of these data points in the respective documents.

Procedurally, SiteScraper disambiguates the positions for the text chunks across multiple seed documents as follows. It first calculates the similarity for each pairing of text chunk and element in a given seed document. It then calculates the total similarity for each labelled edge by summing across the similarity lattices — e.g. for the pairing of */html/body/table/tr[2]/td[3]* and text chunk 1 across the two seed documents. The location with the highest total similarity for a given text chunk is considered to be the best match. Table 2 shows the results of these similarity calculations for the case of the second text chunk (TC_2) across Doc_1 and Doc_2, for the three positions where there is at least one match. On the basis of the indicated calculations, SiteScraper would correctly identify TC_2 as corresponding to tomorrow's temperature in Sydney, i.e. `/html/body/table/tr[3]/td[3]`

The more seed documents (and associated text chunks) the user provides to SiteScraper, the more likely that duplicates can be resolved and the correct locations isolated. For some websites the desired data will appear at multiple locations on every webpage — for instance tomorrow's temperature may be at the top of the page and again further down as part of the weekly forecast. In this case, choosing either location is valid because they represent the same data point, but for consistency we take the first appearing location on the basis that it is likely to be more prominent in the webpage.

4.3 Generalise

The third step is to generalise the best-matching locations and combine them when appropriate. The idea to use XPath to represent a string location was inspired by Sifter (see Section 2.3), a tool that faced similar technical problems to SiteScraper in abstracting a location. Sifter starts with a direct XPath to a string, such as the following for tomorrow's temperature in Melbourne:

```
/html/body/table/tr[2]/td[3]
```

This XPath points to the second temperature, but what Sifter wants is a bounding box around a related set of data, in this case all of tomorrow's temperatures. To match this set Sifter needs to relax the specificity of the XPath to increase coverage and produce an XPath equivalent to:

```
/html/body/table/tr/td[2]
```

In this new XPath the table row element tag lacks a specific index so it will match all rows in the table.

SiteScraper's abstraction needs differ from those for Sifter, but the general idea still applies. At this stage SiteScraper has a set of XPaths to the matching content and needs to decide which XPaths represent content that are related, as opposed to finding spuriously matching content. For simple data scraping cases such as tomorrow's temperature in Melbourne this abstraction stage is not necessary, however it becomes necessary for more complex webpages where the amount of data varies.

SiteScraper will only abstract a set of XPaths if they satisfy either of the following conditions:

1. The locations of the input strings are sequentially ordered. The reasoning is that if there are gaps between locations then the user is trying to select a subset and does not want them all. For example if the input for Doc_1 was *22* and *26*, which skips Sydney, then the user must explicitly want just temperatures from Melbourne and Adelaide.

2. The number of siblings for the abstracted tag varies across the example documents, to show that the field is dynamic. The example webpages for Doc_1 and Doc_2 have just three rows so all of tomorrow's temperatures could simply be extracted with three XPaths. However if a further example was given from the website that had ten cities then the XPaths would need to be abstracted to match all the cities in both cases.

Once SiteScraper has determined all the potential abstractions it must choose the set that matches the maximum number of strings. To do this it orders the abstractions by the number of locations they successfully match. Each location can only match once in a given document, so abstractions that overlap with a higher-ranking one are discarded.

To illustrate this process, consider the case of providing SiteScraper with the single seed document Doc_1 and all six temperature strings for today and tomorrow. From this, SiteScraper would find abstraction possibilities along the two columns and the three rows. The rankings for these results are shown in Table 3. After filtering out the duplicates, we are left with the column abstractions, which is what we wanted.

4.3.1 *Attributes*

We found that the type of XPaths used in the previous example do work, but are brittle. If, for instance, an extra `span` tag is inserted into some weather webpages to feature an advertisement, then the model to extract the heading will break because it is expecting this content to be under the first `span`. To take account of this we used attributes when available instead of indices. In our example the `span` tag has a `class` attribute set to `heading` so then the heading XPath would become:

Ranking	Abstraction	# matches	After filtering	Use?
1	`/html/body/table/tr/td[2]`	3	3	Yes
2	`/html/body/table/tr/td[3]`	3	3	Yes
3	`/html/body/table/tr[2]/td`	2	0	No
4	`/html/body/table/tr[3]/td`	2	0	No
5	`/html/body/table/tr[4]/td`	2	0	No

Table 3: Ranking of abstraction possibilities for Doc$_1$ and the text chunks $\langle 20,22,25,22,26,26 \rangle$

```
/html/body/span[@class=heading]
```

Now the model can still match if an advertisement is inserted because it is not dependent on the index.

This change can lead to a loss of accuracy if there are adjacent tags with identical attributes. However in development, we found that when this is the case the data is usually related so the user wants both anyway. This potential loss of accuracy was worth the more frequent increase in coverage. If the attribute was an `id`, which should be unique, then we can be very confident of matching the right location.

5. EVALUATION

To evaluate the performance of SiteScraper over different scraper needs, we identified three defining parameters for the text chunk types:

1. Whether there was a single or multiple text chunks to extract;
2. Whether the number of text chunks was static or would vary and require abstraction;
3. Whether the text chunk was simple, such as a single number, or complex, such as a news article.

We then chose a number of popular websites and grouped them by these properties, as shown in Table 4. For each text chunk type there are two development sites and one test site. We used the development sites when building SiteScraper to experiment with what features are effective, and the test sites only at the end of the development cycle as a blind test of SiteScraper's true capabilities.

For each of these chosen sites we collected 103 webpages that contained data we wanted to extract with SiteScraper, of which 3 were randomly selected as seed documents and the remaining 100 used as test documents. There are a total of 18 websites in Table 4, making for a total of $18 \times 103 = 1854$ webpages that needed to be collected and hand-annotated for text chunks. This would take considerable time to do fully manually, so we tried to automate this process as much as possible.

We identified two overarching types of websites in our combined data set that require two distinct approaches to document selection and text chunk generation:

1. Monolithic websites (such as web forums or news sites) where the webpages are connected by links and can be crawled. Before crawling we took the URLs for a few example webpages from each website that contained the content we wanted to scrape. From these URLs we created a regular expression that matched the example URLs and could be used to identify other webpages with the desired content. Then we set the crawler loose on each website to follow links until it had 103 URLs that matched the regular expression.

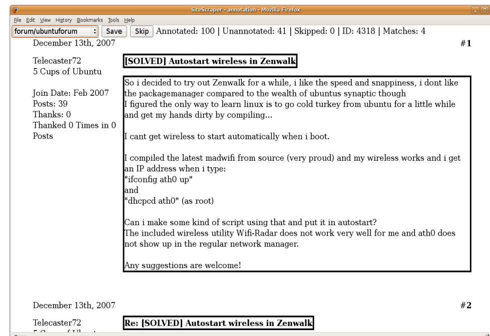

Figure 5: Text chunk annotation interface

2. Search-based websites (such as a stock sites or, obviously, search engines) where the webpages are created dynamically from a query and are not necessarily linked to from elsewhere. This kind of website can not be crawled because the webpages are not connected by links so we needed to use their searching facilities. For each search-based website we crafted a URL template that would interface to their search engine. For Google this URL was `www.google.com/search?q=`. Then for each genre (see Table 4) we chose 103 queries and combined each with the URL template to collect our data.

While collecting the data we manually examined each URL to filter out webpages that did not contain the type of data we wanted to scrape and so were irrelevant, such as a contact pages.

Now we were ready to begin scraping the collected data. For the seed documents, we manually scraped the text chunks from within a web browser. For the test documents, on the other hand, we first trained a model over the seed documents, and then fed the output for each test document through the annotation interface shown in Figure 5. For each webpage the interface highlights the text that SiteScraper has scraped, and the user adjusts this result by selecting missing content or deselecting invalid content. This process leads to four different cases:

1. Text that was returned by SiteScraper and did not require adjustment (True Positive = TP);
2. Text that was not returned by SiteScraper and was not adjusted (True Negative = TN);
3. Text that was returned by SiteScraper but was deselected (False Positive = FP);
4. Text that was not returned by SiteScraper but was selected (False Negative = FN).

	Text chunk type	Genre	Development sites	Test site
1	Multiple, static, simple text	Finance	asx.com.au money.ninemsn.com.au	au.finance.yahoo.com
2	Single, static, complex text	News	theage.com.au theonion.com	theaustralian.news.com.au
3	Single, dynamic, simple text	Weather	finance.google.com au.weather.yahoo.com.au	www.bom.gov.au
4	Multiple, dynamic, simple texts	Finance	imdb.com amazon.com	ebay.com
5	Multiple, dynamic, moderate texts	Search	au.yahoo.com altavista.com	google.com
6	Multiple, dynamic, complex texts	Forum	linuxquestions.org stackoverflow.org	ubuntuforums.org

Table 4: Data sets used for experimentation

These four cases can be used to judge the performance of SiteScraper by calculating the precision, recall, and F-score, commonly-used evaluation metrics in the fields of information retrieval and natural language processing. Precision measures the correctness of the predicted text chunks and is defined as:

$$Precision = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

Recall, on the other hand, measures the proportion of relevant text chunks that the system was able to identify, and is defined as:

$$Recall = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

Both of these measurements are necessary to get an accurate perspective on SiteScraper's performance. If SiteScraper is very conservative in judging which text chunks to return then the results may be high in precision but low in recall, because much valid content was not returned. Alternatively if SiteScraper is very liberal in its judgements then it will return a lot more content and the recall should rise, but the precision is likely to fall as the predictions get noisier.

Finally, F-score combines precision and recall to give an overview of performance, and is defined as:

$$F\text{-}score = Precision \times Recall$$

Our results for SiteScraper in Table 5 and Table 6 are given in terms of these three measurements. As an overall evaluation across all sites, we provide macro- and micro-averages of each of the precision, recall and F-score in each table. The macro-average is simply the arithmetic mean of the individual values, while the micro-average is calculated by summing up the TPs, FPs and FNs across all the sites and calculating an average directly from the totals.

6. RESULTS

While developing SiteScraper 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 (see Table 5). When we applied SiteScraper to the blind test sites, we were very encouraged to find that the macro-averaged F-score was almost identical at 0.97 (see Table 6).

From the experience of annotating we noticed that usually when SiteScraper made a mistake the cause was a

Site	P	R	F
asx.com.au	1.00	1.00	1.00
finance.google.com	1.00	1.00	1.00
theage.com.au	1.00	1.00	1.00
theonion.com	0.95	1.00	0.95
weather.ninemsn.com.au	1.00	1.00	1.00
au.weather.yahoo.com.au	1.00	1.00	1.00
imdb.com	1.00	1.00	1.00
amazon.com	0.99	1.00	0.99
au.yahoo.com	1.00	0.98	0.98
altavista.com	0.97	1.00	0.97
linuxquestions.org	1.00	0.89	0.89
stackoverflow.org	1.00	0.90	0.90
Macro-average	0.99	0.99	0.98
Micro-average	0.98	0.99	0.97

Table 5: Results over the development sites (P=Precision, R=Recall, F=F-score)

variation in the webpage structure from the examples used to generate the model. And generally the more complex chunk types (see Table 4) suffered more variation, which contributed to their lower performance. SiteScraper 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 SiteScraper to scrape. Types 2, 3, and 5 performed very well, with the lowest F-score being 0.95 for TheOnion where the author details were incorrectly included when scraping certain articles. Type 6 had the lowest overall F-score as a group, with LinuxQuestions producing the lowest F-score in the development set. The main reason for this is that in some LinuxQuestions threads the responder would embed a code snippet within a sub-tag. The LinuxQuestions 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 LinuxQuestions could have been improved without any change to SiteScraper.

To test this hypothesis we retrained the model for LinuxQuestions with a larger example set of six seed documents that all contained quotes in their thread posts, and then re-annotated 20 webpages 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

Site	P	R	F
au.finance.yahoo.com	1.00	1.00	1.00
theaustralian.news.com.au	1.00	0.99	0.99
www.bom.gov.au	1.00	0.98	0.98
ebay.com	1.00	0.87	0.87
google.com	1.00	0.96	0.96
ubuntuforums.org	1.00	1.00	1.00
Macro-average	1.00	0.97	0.97
Micro-average	0.99	0.91	0.90

Table 6: Results over the test sites (P=Precision, R=Recall, F=F-score)

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.

The final case not mentioned yet is type 4, which performed near perfectly in the development set but poorly for Ebay in the test set, which had the lowest F-score of any website. This was due to the same problem as LinuxQuestions of having significant variations in the structure. Ebay has a number of different ways to show the price of an item depending on the state of the auction, whether the seller has paid for promotion, and whether the item is available for direct sale or auction. These variations confused SiteScraper from extracting every item price. Once again, here we expect that an expanded seed document set (recalling that the original model was training on only 3 seed documents) would improve performance.

7. DISCUSSION

SiteScraper had two original goals, as explained in Section 1:

- Simple to scrape fine-grained data.
- Possible to automatically retrain after structural changes.

To scrape a new webpage SiteScraper just needs text chunks copied and pasted from a few seed documents. The results in Table 6 show that SiteScraper can accurately learn a model from this lightweight input, so this first goal was achieved. Scraping the web forums performed worse than average, which is unfortunate given that scraping web forums was the motivation for SiteScraper, but this is perhaps expected given that their structure is more complex. Also, as noted above, providing more seed documents with greater variety seems to be the solution to this problem, so it would not appear to be an inherent limitation of SiteScraper.

It is worth noting here that we originally included BBC News[13] as one of our news sites. However, we found that the BBC uses so many different templates for different article types that manually extracting the text from articles proved very time-consuming, and we eventually changed to scraping The Onion[14] instead. In this sense, our data set is slightly skewed towards more internally consistent websites.

The second goal of surviving a changed structure is possible to achieve with SiteScraper under certain circumstances. When the structure of a forum is updated the content of the threads stays the same so SiteScraper can retrain its model automatically using the original content with the updated HTML. This is possible because web forums maintain historical content which the model can be retrained over. News sites also maintain historical content so the same technique could be applied to them.

To test this idea we tried using older versions of the LinuxQuestions threads from the Internet Archive[15] that were crawled in 2002. These webpages would have the same thread content within an older structure. Unfortunately we found that the Internet Archive only archived the sticky threads, which are administrator threads that 'stick' to the top of the thread list, so we could not reuse the same data as before. After copying and pasting the post strings from the sticky threads we were able to successfully train models for both the old and new web forum versions using the same content. We annotated 20 threads over these new models and found they both attained a perfect precision of 1.0 while the 2002 threads had a recall of 0.91 and 2008 0.94. The recall for these results are higher than achieved in the original experiment (see Table 5), however this can be attributed to the threads being administrator discussions about forum policy so there are less code snippets, which is what damaged the recall previously. The 2002 model performed marginally worse than for 2008 because occasionally the first post had a different set of attributes and was not matched. This experiment proved SiteScraper could be automatically retrained when the structure was updated using the same content.

We found that the structure of the tags in the 2002 SiteScraper model were almost identical to 2008, but the attributes were entirely different. In 2002, LinuxQuestions had no external CSS files and as a result many of the style attributes were embedded in the tags. By 2008, however, the most common attribute was the `class` settings, and the style definitions were being made in external CSS files. This is good news for SiteScraper because it means LinuxQuestions, and hopefully many other websites, will not need to update their HTML structure as frequently as in the past because re-styling of the website can be performed externally to the basic document markup.

Even with this move towards separating content from structure through CSS we found that structural updates are common — over the two months we worked on SiteScraper the structure for `theaustralian.com.au`, `amazon.com`, and all three stock sites were updated. That is almost a quarter of our data in just two months, which means handling structural updates is a real problem that needs addressing.

It would theoretically be possible to automatically retrain stock and weather websites too, but more complicated because these sites generally do not maintain historical data. However stock and weather data are independent of the websites that display them so if models were trained for a set of websites that used the same data source then when one model was broken from an update it could be retrained using current data from the other models.

[13]http://news.bbc.co.uk

[14]http://www.theonion.com

[15]http://www.archive.org

Retraining commerce websites and search engines is a more challenging problem. These kind of websites do not maintain reliable historical data — prices change and search ranking algorithms are tweaked — and the data they display is generated by them and so cannot be verified elsewhere. If SiteScraper is lucky then the structure change will not happen simultaneously with the content change so that the model could be retrained in between. Otherwise these types of websites fall into the third update category identified in Section 1 of suffering both changing content and structure, which is beyond the scope of SiteScraper. The only hope is retraining based on a specially-crafted query with an expected result, such as the Scrubyt example used in searching Google for *ruby*. However this strategy is clearly not robust.

Where SiteScraper is able to handle dynamically-changing content and structure is where the content is largely the same (and structure is arbitrarily different). This tends to occur with user forums, where posts can be added to a thread at any given point, but in a monotonically-increasing fashion. Assuming that the level of change over the seed documents is relatively small (i.e. old threads are chosen), the combination of partial string matching and generalisation (see Section 4) can abstract away from the text chunks to identify that all posts in the thread are the target of the crawl.

7.1 Future work

Our results for SiteScraper were good but not perfect. To improve, SiteScraper's model needs to be less brittle and able to handle greater variation. Currently SiteScraper puts all its effort into generating a model of data in a webpage, and then this model is mechanically applied to each subsequent webpage. If the model does not match then it returns nothing, so a potential improvement on failure would be to try relaxing criteria in the model in an attempt to locate relevant data.

One interesting project we came across that is related to this idea is the Internet Scrapbook [7], which is a web scraping system focused on handling changing news content. It was developed back in 1998 and was basically trying to achieve the functionality of what we know today as RSS. It approached this by searching for content it thought less likely to change, such as the heading *Economy*. It would then make the model of the desired data relative to these fixed points. To scrape the page these fixed points were located and then the desired data could be found relative to them. SiteScraper locates everything as absolute from the root of the document, so it could potentially be made more robust through this relative approach.

Another potential for improvement is examining sub-tag content. Currently SiteScraper only breaks down strings by tags, but sometimes there are multiple types of data within the same tag. For instance, unlike the other search engines in our test data, Google combines the URL of a result and the size of the document within the same tag:

```
<cite>www.google.com.au/ - 6k</cite>
```

So if a user uses SiteScraper to extract the URLs from a Google search they will get the document sizes too, which is not ideal. To deal with cases like this SiteScraper would need to support the analysis of sub-tag content.

SiteScraper also needs more work to replicate what the user sees in a rendered webpage. For a start, SiteScraper does not examine embedded IFrames for additional content. This would add some complexity to generating the model, but would definitely be possible. As mentioned in Section 2, SiteScraper does not interpret the effect of Javascript events. This could be addressed by embedding SiteScraper in the browser as a plugin, or using a library such as Watir[16] to interface with the browser.

A final feature that needs implementing is determining whether the structure of two webpages match. With that functionality we could determine when a webpage had changed and the model needs retraining. This would also make the data collection process we did for evaluation easier because then we would not have to manually filter out webpages that lacked the desired data. This would make scraping a diverse website like BBC News more practical.

8. CONCLUSION

SiteScraper 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. SiteScraper has been tested over different domains with high effectiveness, and we also showed its adaptability by going back in time and scraping the LinuxQuestions site from 2002 without re-annotating. On this evidence, we believe that SiteScraper can provide a robust and flexible solution for the problems of dealing with web data.

9. REFERENCES

[1] T. Baldwin, D. Martinez, and R. Penman. Automatic thread classification for linux user forum information access. In *Proceedings of the Twelfth Australasian Document Computing Symposium (ADCS 2007)*, pages 72–9, Melbourne, Australia, 2007.

[2] T. Berners-Lee and M. Fischetti. *Weaving the Web*. HarperOne, San Francisco, USA, 1999.

[3] M. Bolin, M. Webber, P. Rha, T. Wilson, and 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*, pages 163–172, New York, USA, 2005.

[4] D. Huynh, S. Mazzocchi, and D. Karger. Piggy bank: Experience the semantic web inside your web browser. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5:16–27, 2006.

[5] D. Huynh, R. Miller, and 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*, pages 125–134, New York, USA, 2006.

[6] M. Schrenk. *Webbots, Spiders, and Screen Scrapers*. No Starch Press, San Francisco, USA, 2007.

[7] A. Sugiura and Y. Koseki. Internet scrapbook: 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] `http://wtr.rubyforge.org/`