Scrappy: Simple Web Scraping

Amir Ghazvinian[1] **Sean Holbert**[2] **Nikil Viswanathan**[2]

Stanford University
[1]Department of Biomedical Informatics
[2]Department of Computer Science
{amirg, sholbert, nikil}@stanford.edu

ABSTRACT
A tremendous amount of interesting data exists on the web; however, much of this data resides in a semi-structured format that hinders extraction and data analysis. Writing programs to extract this data requires substantial tedious effort and also presents a technical blockade for those without programming experience. To provide access to these new sources of data, we present a simple browser-integrated visual interface that facilitates data acquisition over multiple similar web pages. Here, we describe our system, nicknamed Scrappy, and show that it enables data acquisition by eliminating technical and effort barriers to gathering this data.

Keywords
Web Scraping, Data Acquisition

ACM Classification Keywords
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION
As the Internet has continued to grow, the amount of data publicly available has snowballed into an unbelievable size. While this data has a lot of power and potential, availability doesn't necessarily translate into accessibility; due to the structure and immense size of the web, many questions cannot easily be answered using this data. Web scraping provides a great solution to facilitate digesting web data, providing highly specific rules on what data to gather and aggregate. Still, web scraping requires a sophisticated understanding of programming, web technologies such as HTML, and the structure of data on the web (e.g., the Document Object Model (DOM)). As a result, programmers who want to collect data from the web often find themselves writing tedious, complicated scripts and people without technical experience often find themselves unable to collect the data at all.

Here, we present a browser-integrated visual interface for automated data acquisition over multiple web pages with the goal of reducing the current barriers to extracting relevant data on the web. We attempt both. Our system, Scrappy, is designed for sites that are composed of many pages of similar semantic structure. As a simple example, the Internet Movie Database (IMDB) contains several movie pages, where each page consists of data for one movie and all of the movie pages have similar structure. Other examples include Amazon products and Yelp businesses, among others. These sites generally use a templating engine to transform database records into well-formatted, user-readable content. Our tool takes advantage of the structural similarity across pages to allow users to scrape content from multiple pages after creating a template corresponding to just one example page.

In this paper, we describe the implementation of our system, present our observations and results from using Scrappy on a number of web domains, and discuss the challenges of designing a simple, flexible tool for data acquisition on the web.

RELATED WORK
There are a number of enterprise products and research projects out there that try to solve the problem of data acquisition and attempt to make web scraping more accessible.

In the research domain, the Thresher project is most similar to ours in approach and scope in that it attempts to address the problem of facilitating scraping for non-technical end-users [1]. Using Thresher, users create a "wrapper" which specifies the semantic relations between data on a page in RDF format. Using this wrapper, Thresher provides a fairly sophisticated and reliable solution for scraping and interacting with semantic content on single pages. We extend this approach in a number of ways. First, we address data aggregation across pages and attempt to automate the scraping process over multiple pages. Second, we provide a set of suggestions for scrapable content on the page, with the aim of making the data acquisition task one of recognition rather than recall. Additionally, in the interest of making scraping accessible to non-technical users, we created Scrappy as a Firefox add-on, which is easy to install and works across platforms. Thresher uses MIT's Haystack browser, an

impressive technology, but leaves to user constrained to a much less commonly used browser.

There are also several prominent enterprise tools for acquiring data from the web, such as Mozenda [2] and Visual Web Ripper [3]. Both of these tools run in a custom browser and require significant effort to set-up properly and to understand or use effectively. Mozenda, perhaps the most usable of the available enterprise tools, is designed to collect data for multiple similar elements listed on the same web page, such as a series of product reviews. We focus here on applying a template across multiple similarly structured pages rather than within a single page as Mozenda does, but this serves as a promising area for future exploration.

THE SYSTEM

Scrappy is designed to scrape web content from sites that are composed of many pages of similar semantic structure. The system is implemented as a Firefox browser extension, and works in three main stages to scrape web data. First, a user navigates to a page that he would like to scrape and generates a template for the content that he would like from that page. Next, the user selects a set of links that point to pages matching the content template defined by the user. Finally, the user selects an output data format and Scrappy crawls the links specified by the user and scrapes content corresponding to the user's template. We discuss each of these stages in detail in this section.

Template Creation

The first step in acquiring data from the web using Scrappy is to create a template that can be applied to scrape data from multiple pages of similar structure (Figure 1). In order to specify such a template, the user navigates to an example page and begins template mode. Upon entering template mode, our system does four things:

1. We modify the structure of the current page to improve the user's ability to select content on the page for scraping. For any elements on the page that match the pattern <tag1>*text1*<tag2>*text2*</tag2></tag1>, we enclose *text1* in a tag so that the user may easily highlight and select just *text1* without selecting *text2*.

2. We suggest elements on the page that may be of interest for scraping. At this stage, suggestions are purely structural. Here we suggest <table> elements as well as several other types of elements (e.g., <h1>, <a>,) as long as these elements are not contained within a <table> element.

3. We highlight suggested elements in yellow on the page and allow users to hover over and select any element (suggested or not) for scraping. When a user hovers over a scrapable element, this element is temporarily highlighted in blue on the page. When the user selects an element, the element highlight becomes permanently green and the user is asked to name the field he is scraping.

4. We disable links on the page so that a user does not accidentally navigate away from the page while scraping.

The four processes above form the core of Scrappy's template creation mode. However, we also enable users to refine the suggestions made by the system by specifying the location of a page with a similar template. If a user chooses to do this, we compare the content on the template page to that of the additional page, removing content contained on both pages. This typically allows the system to avoid or remove suggestions that include elements in the page's header, footer, or sidebar, which are common across all pages with the same template.

Figure 1. Screenshot of a portion of an IMDB page in template creation mode. Items in highlighted in yellow indicate suggestions made by our system, while items highlighted in green indicate elements selected by the user for scraping.

Link Selection

Once the user has selected a content template, he must specify a set of pages that match that template in order to scrape data from these pages. The user navigates to a page that contains links to a set of pages matching the template specified in the previous step. Here, the user can select the

links that he wishes to scrape simply by click on them. When the user hovers over a link in link selection mode, Scrappy highlights a set of similar links. Clicking on any of these links will select and highlight all of them for scraping, which saves the user time in specifying links for scraping. The system determines link similarity by the similarity of the XPath queries required to reach those links in the page's DOM [4].

Of course, the link selection process requires that a page actually exists with all links that the user wishes to scrape. Improving the flexibility of link selection will be an important area for future work.

Scraping

Once the user has specified a content template and has selected a set of links to crawl, he simply selects an output format (currently CSV and JSON are supported) and begins the scrape. At this point, Scrappy launches threads to the pages given by the user in the link selection process and searches for the elements corresponding to the content template. For each field in the template page, we have previously generated an XPath query, which we use to query the DOM of each new page in order to identify the correct element on the page. If we are unable to locate the element with the given XPath query, which is possible with small differences across pages, we generalize the query and attempt to find the element a second time. Once we have finished scraping all pages in this manner, Scrappy compiles the results from all pages, converts the data to the appropriate format, and prompts the user to save the data to his file system.

RESULTS

We conducted preliminary tests of the system on a set of four users, running Scrappy on a number of different sites

Data	URL	Successfully Scraped	Comments
IMDB Movies	imdb.com	Title, summary, year, genre, cast, rating, director, etc.	Success!
Amazon Music Store	amazon.com/ music	Artist ,genre, album, album length, album price, song name	Successfully scraped data into 3 tables linked by an "id" column.
Android App Store	market.android.com	App name, description, 'last updated', app size, app genre, app version, supported android OS, Price, content rating.	Success!
Google Finance	finance. google.com	Failed, but could scrape name and shares	Some major fields like market value and daily change could not reliably scrape. Google actively tries to prevent scraping.
South American countries on Wikipedia	en.wikipedia.org/ wiki/ South_america	Failed	User tried scraping the right-aligned details table of South American countries. Differences in DOM structure for different countries made scrape unreliable.
LinkedIn connections data	linkedin.com/ connections	first name, last name, title, most recent job position, professional experience summary	This scrape browsed data for all of a user's connections on LinkedIn. Some anchor elements that were selected in the template didn't scrape due to Javascript actions LinkedIn associated with the links
ESPN Basketball Stats	http://espn.go.com/nba/statistics	Complete 'GAME LOG' and 'STATS' tables for each player, player name, 2010-11 season stats	Some of the subheaders in one of the tables didn't scrape. This was fixed by the user by comparing Scrappy's output and the original template.

Table 1. The results of scraping several domains using Scrappy.

to test its scraping capabilities. Table 1 lists the sites we tested with fields that were successfully scraped, along with comments relevant to the domain. Users generally felt the interface was intuitive and understood how they were marking up the data on a page in relation to the output file(s). Some suggested minor UI changes for entering the second example URL, but otherwise they were generally satisfied with the UI. Out of the users and scrapes we tested, only two scrapes really were categorized as failures (see Table 1).

DISCUSSION

Scrappy demonstrates that technical skills are not a requirement for scraping data from the web, and the application provides a fairly reliable platform for scraping many different data domains on the web. While Scrappy is powerful, it only solves the first step of data acquisition in the long pipeline of data manipulation processes leading to meaningful visualization and analysis. Scrappy can be easily coupled with other user-friendly data cleaning and integration products like Wrangler [5] to help refine the data pulled from the web. We noticed first hand that data cleaning products were especially useful for numeric values pulled from the web, as they often include units or edge case values (i.e., for a "price" attribute in some cases it might say "Free" instead of $0).

The inability of Scrappy to scrape particular sites in tests highlights some of the limitations of the system. For example, in one test, a user attempted to scrape data from infoboxes on Wikipedia pages for South American countries. However, despite the boxes containing somewhat structured fields and information, the structural variation across pages prevented us from scraping the pages successfully. This limitation of the system suggests the need to tie pages together semantically as well as structurally, as the data in Wikipedia infoboxes conforms to a defined vocabulary of field names.

There are also many areas for future work. First, allowing users to perform aggregation of specified fields on a page would be very powerful. For example, if you are pulling LinkedIn data, and you want to know how many jobs someone had in the past, you might want to group all instances of jobs into the same field.

We also observed while developing Scrappy that even sites that generate pages according to a template can have small variations across the domain. For example, some IMDB pages include an additional widget in the page for users of the site to find movie show times near them; this widget only appears in pages with movies currently in theaters. We currently allow some generalization of DOM queries to allow more flexibility in finding data across pages, but accounting for small variations in the page is a major challenge that will have to be addressed more robustly in future work.

In this work, we have presented a system that allows a non-technical user to scrape data from the web through a simple, browser-integrated interface. While there is much remaining work to be done in making the system more flexible and usable across a wide variety of sites, our initial user studies have shown the tool to be both relatively easy to learn and useful in scraping data from a number of different domains. We hope that the availability of such a tool will lower the barrier to scraping data and make the web of data more accessible for analysis in the future.

ACKNOWLEDGMENTS

We would like to thank Jeff Heer and Diana MacLean for their feedback and ideas in implementing this project.

REFERENCES

1. Hogue and D. Karger, "Thresher: Automating the Unwrapping of Semantic Content from the World Wide," *Proc. 14th Int'l Conf. World Wide Web (WWW),* pp. 86-95, 2005.
2. Mozenda. http://mozenda.com. Accessed June 8, 2011.
3. Visual Web Ripper. http://www.visualwebripper.com/. Accessed June 8, 2011.
4. XML Path language (XPath) specification. http://www.w3.org/TR/xpath, 1999.
5. Wrangler: Interactive Visual Specification of Data Transformation Scripts. Sean Kandel, Andreas Paepcke, Joseph Hellerstein, Jeffrey Heer. ACM Human Factors in Computing Systems (CHI), 2011