Robust Web Scraping in the Public Interest with AutoScrape

Brandon Roberts
Artificial Informer Labs
Seattle, Washington
brandon@bxroberts.org

ABSTRACT

Web scraping is a foundational task in journalism and tends to be performed using custom, one-off tools. Traditional methods involve constructing HTTP requests and extracting data using XPath[2]. As web sites become more interactive, these methods require an increasing amount of manual effort to develop and maintain. This paper builds on previous work in text-based extraction techniques[8], adapts them to navigating a real browser, and proposes using Hext, a novel domain-specific language for extracting structured data from HTML. We introduce AutoScrape, an investigative-focused web scraping tool which implements this framework. AutoScrape can simplify many common journalistic data gathering tasks and reduce maintenance costs. In partnership with several non-profit media organizations, this paper will also present case studies describing common investigative tasks and illustrate the use of this framework to successfully solve each problem.

ACM Reference Format:
Brandon Roberts. 2019. Robust Web Scraping in the Public Interest with AutoScrape. In *Proceedings of Computation + Journalism Symposium (C+J Symposium).* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

Web scraping is a fundamental task for investigating all levels of government and industry. However, almost all web scrapers are implemented as custom tools tied to a specific website and use case[4]. This leads to an unnecessary duplication of work.

Many sites include session identifiers and token systems which need to be extracted in order to successfully perform HTTP requests. The increasing use of client side JavaScript in search forms complicates the use of scrapers which operate using direct server requests. Web pages often undergo changes to their style, layout, infrastructure, or query mechanism. The bulk of data extraction techniques, required for obtaining session identifiers, tokens, and data, use HTML Document Object Model (DOM) path techniques[5]. XPath[2] is a popular method for identifying HTML nodes by their location in the DOM tree. When a site layout changes, these techniques often fail, and maintenance is required for the continued operation of a scraper.

Combining crawling and navigation with extraction of structured data is a common pattern in the development of web scrapers. This introduces brittle elements to these tools and could lead to a loss of time during an urgent investigation.

AutoScrape is a new web scraper for interactive pages, with an emphasis on facilitating common journalistic tasks. By separating the two major concerns of a web scraper–crawling and data extraction–AutoScrape alleviates major obstacles across a range of scraping tasks. It operates using three principles:

(1) Driving a real browser, navigated via page text, using Selenium WebDriver: a software library for controlling web browsers with code[3].
(2) A simple syntax for filling a variety of interactive and static form input fields.
(3) Using Hext[1], a novel domain-specific language (DSL) for data extraction from HTML, to build datasets once a scrape has completed.

These techniques have been applied, in practice, with several reporting and research partners. We will present the use of AutoScrape in four case studies. These cover a variety of common tasks encountered when building stories or performing research.

2 TEXT-BASED AUTOMATED WEB NAVIGATION

The fundamental concern of a web scraper is devising a strategy for identifying types of pages and elements requiring interaction. Typically this is done by parsing the HTML DOM and extracting XPath or CSS path tags, each corresponding to some component of the page. Information contained in these elements is then typically used to fetch additional pages or data itself. As mentioned above, these techniques often fail to survive layout or form modifications.

C+J Symposium, 2019, University of Miami, Florida, USA
2019.

[1]Hext, written by Thomas Trapp, is available at http://hext.thomastrapp.com/.

To address these challenges, AutoScrape builds on text-based data extraction techniques[8] and proposes a text-based approach for identifying navigational page elements. Essentially, the paradigm of traditional scrapers is inverted. Instead of specifying tags leading to elements, AutoScrape builds a list of all tags on a page and associates them with their text. When given a user-defined ruleset for links/buttons to click and forms to seek, AutoScrape can perform a tree search of a site, interacting with elements and downloading documents/pages as the site is traversed.

The reasoning behind this is simple: while the makeup of a website may change frequently, the text of a page is typically static and/or standardized. Leveraging text, not tags, allows for the construction of robust scrapers that can survive significant site modifications. If breaking changes do occur, AutoScrape can be easily updated through its options. Examples of this will be detailed more fully in Section 5.

3 DATA EXTRACTION WITH HTML-TO-JSON DSL: HEXT

The secondary task in any web scraper is the extraction of structured data. This is typically built into the scrape process, but AutoScrape's framework deliberately separates it into a subsequent objective. Since the crawler itself simply grabs all HTML pages and documents, extraction is no longer an impedance on the scrape itself. If the extractor has become out-of-date and is no longer extracting the correct data, it can be quickly noticed; a followup scrape does not need to be performed. Separating these concerns allows time to be saved.

Hext, a combination of "HTML" and "extraction", is a DSL for extracting JSON data from arbitrary HTML documents. Its syntax closely resembles HTML and enables the organizing of data into labeled columns and performing text preprocessing operations. Figure 1 illustrates its use. Trimming, prepending strings (*e.g.*, adding a hostname to a relative URL path) and regex replacements are all trivial to implement.

Data extraction with AutoScrape using Hext templates is a simple batch processing task. If a source HTML document doesn't match the Hext template, it is simply ignored. This eliminates the need to handle non-matching pages manually and opens the possibility for massive parallelization.

The translation from a source HTML record, such as a table row, to a Hext template is straightforward enough that AutoScrape includes an automatic extractor building tool. Similar to the methods used in SiteScraper[8], users can provide an HTML document and a list of column names and values for each field. AutoScrape can use this information to build a Hext extraction template. This is done by:

(1) Asking the user to identify the DOM elements that form a single row to be extracted. This can be achieved by prompting a user to click on HTML elements or by providing a list of strings which make up the row.
(2) Finding the least common ancestor of all these selected elements.
(3) Extracting the outer HTML of the least common ancestor.
(4) Stripping unnecessary attributes (such as style tags, JavaScript, etc).
(5) Adding Hext extractor syntax to the selected elements.

Once these steps are complete, we are left with a Hext template. In the case of a significant layout change, the same process can be used with the updated HTML page to rebuild the template.

4 INTERACTIVE WEB SCRAPING WITH AUTOSCRAPE

The result of combining text-based interactive web navigation and data extraction as a secondary step is a robust framework, capable of many common journalistic tasks. AutoScrape has been developed with journalists in mind and provides simple ways to scrape difficult sites without needing to write custom software.

Currently, users can specify constraints on crawls via link text matching, fill input fields, use date selectors, click checkboxes and submit forms using several techniques. This is accomplished using command line configuration, as demonstrated in Figure 2. To simplify both parsing and writing scraper interaction plans, a text-based syntax was selected over the more complicated (but powerful) schemes of other scrapers[7].

To accommodate the range of journalistic tasks often required in investigations, AutoScrape can operate in two modes: crawl and interactive search. Crawl mode performs a depth-first search (DFS) of a web site, downloading all documents and pages encountered. Interactive search mode allows users to specify a configuration (see Figure 2) detailing types of inputs to interact with, what to enter into fields, how to submit a form, and how to look for next buttons on subsequent results pages. The two modes can be combined. In the case of a significant page redesign, the search form can still potentially be discovered via DFS and scraped per the predefined rules.

AutoScrape also includes numerous options in order to accommodate the range of tasks routinely needed in investigative work. We've partnered with several organizations in order to apply AutoScrape in the building of investigative stories. The use cases presented will display AutoScrape's abilities to tackle common problems, large and small, under a unified scraping framework.

```
<!DOCTYPE html>
  <body>
    <table>
      <tr>
        <td>
          <a href="/p?id=199">199</a>
        </td>
        <td>
          <span>01/03/1994</span>
        </td>
      <tr>
      <!-- more recs -->
    </table>
  </body>
</html>
```

```
<tr>
  <td><a @text:ID /></td>
  <td><span @text:DATE /></td>
</tr>
```

```
{
  "DATE": "01/03/1994",
  "ID": "199"
}
```

Figure 1: An example extraction of structured data from a source HTML document (left), using a simple Hext template (center) and the resulting JSON (right). If the HTML source document contained more records in place of the comment, the Hext template would have extracted them as additional JSON rows.

```
scrape.py \
  --input "i:0:Seattle[:enter:];d:0:05-30-1992" \
  --next-match "Next Page" \
  --form-match "Document Access Search" \
  --form-submit-natural-click \
  https://site.tld
```

Figure 2: An example command line invocation of AutoScrape. This will (1) run an interactive crawl of a site, starting at site.tld, (2) seek a form with the text "Document Access Search", (3) fill the first input with "Seattle" followed by the enter key (to complete a JavaScript-enabled input), (4) fill the first date field with May 30th, 1992, and (5) submit the form, simulating a real click over the submit button. Once the request has completed, AutoScrape will continue to click "Next Page" buttons until all pages have been found. All HTML pages visited during this crawl will be saved to disk.

5 CASE STUDIES

The types of investigative and research-oriented scraping tasks encountered while trialing AutoScrape can be summarized by: document gathering, single-query searches, and full range scrapes with paginated results (*e.g.*, search for all letters *a* through *z*, where the results are organized into pages). Most of these scrape paradigms were found to be dependent on interacting with dynamic JavaScript components.

The organizations we partnered with while testing AutoScrape were *PublicSource* in Pittsburgh, Pennsylvania, *The Austin Bulldog* in Austin, Texas, History Lab at Columbia University, and an independent investigative freelance journalist, Emmanuel Freudenthal. The use cases demonstrated will describe the types of problems frequently encountered while web scraping and AutoScrape's approach to resolving them. The following section is intended to not only be a demonstration of this framework's abilities, but also to summarize common scraping needs across journalistic activities.

Case 1: Background Investigations of Political Candidates

The Austin Bulldog is an investigative nonprofit focused on independent reporting in the Austin, Texas metropolitan area. One of its recurring reporting initiatives is conducting background research into candidates for local office. In 2014, the City of Austin reorganized its city council, moving from a city-wide election for all seats to a district-based representation system. This resulted in the adding of new council seats and an increase in candidates. The 2018 election cycle featured 28 candidates for council and mayor. In order to facilitate background investigations, we employed AutoScrape on ten websites containing public information; this was done on a regular basis. The first scrape established a baseline for existing records and subsequent scrapes identified new events occurring during the races.

The primary challenge was the number of different web sites in need of scraping. Secondly, many of the government-run search portals were powered by JavaScript functionality which resisted automation. (This was the result of unorthodox use of client side programming, not a deliberate attempt to stop web scrapers.) Extraction of structured data from raw HTML was greatly aided by the fact that several of the sites operated by one of the local governments shared the same layout. This reduced the number of extractors required.

When driving automated browsers, finding a workable mechanism for submitting search forms can be problematic. One page, in particular, used a combination of legacy JavaScript and highly nested DOM callbacks to drive the search functionality. The submit button merely cleared the page when clicked artificially. Dispatching of search queries

and rendering of results hung off an event triggered by clicking an ancestor element to the submit button. Unfortunately, Selenium WebDriver only provides two built-in ways to submit forms: an element click event or a form submit shim. In most cases, one of these two strategies will work to submit a form. But in this particular case, we needed to simulate a real positional click over the submit button's location on the screen. This led AutoScrape to include a *natural click* option, which successfully triggers the submit button and all overlayed elements.

The result of this scrape was the continual collection of information on all 28 candidates. A full scrape across all sites and candidates returned upwards of 300 records and could be completed in two hours when ran in parallel. While minor changes to page layout did occur, none of them required re-configuring AutoScrape.

Case 2: Local Government Procurement Monitoring

PublicSource is a nonprofit in-depth and investigative news organization based in Pittsburgh, Pennsylvania. Part of their reporting includes investigating and continually tracking government contracts. These were available on a publicly accessible search portal. Unfortunately, only part of the contract data was accessible through this portal. Additional information about the background and current status of these same contracts was held on a separate, non-interactive web site. In order to build a unified dataset about the city contracts, AutoScrape was used in both interactive search and plain crawl mode against the two sites.

Like many search forms, the Pittsburgh contracting portal accepted a single character input with a wildcard. This allowed us to use a parallelized scrape, with each scraper instance searching a character, *A-Z* or *0-9*, plus a wildcard. Subsequent search results pages were collected by following *next* button links. The second contract site was able to be simply crawled by pointing AutoScrape at the initial landing page and performing DFS.

In addition to the historical contracting data, *PublicSource* was also interested in continually scraping the interactive contracting portal to retrieve the most recent contracts. This was facilitated by AutoScrape's ability to select dates in HTML5 date input fields.

Two Hext extraction templates were required to build the entire contracting dataset. Creating them was performed using AutoScrape's template builder.

These scrapes resulted in a master list of over 130,000 contracts since 1990, totaling over 22.5 billion dollars. The two contract sites' data gave *PublicSource* reporters a full picture of the contracts which Pittsburgh was engaged in. Gathering ongoing data was performed quickly, using the same techniques.

Case 3: Massive Scrape of West African Freight Data

In addition to the work with investigative nonprofits, AutoScrape was used in partnership with a freelance journalist, Emmanuel Freudenthal, who specializes in data and conflict zone journalism in Africa. The scraping task was concerned with estimating the amount of freight handled by Camrail, a West African rail company owned by a French billionaire. In addition to Camrail's operations in industry, they also run passenger lines which have seen some of the deadliest rail disasters on the continent.

Camrail maintains a public search portal which accepts two pieces of information: an origin port and a five-to-six digit numeric code. While most of these origin-code pairs result in an error page, hitting a correct record displays information related to the train's destination and freight carried.

Unfortunately, there were several complicating factors. Primarily, the search space was large. There were 31 origination ports to select from and a total of 110,000 possible numeric codes for each. This meant a total of 3.41 million searches were required to try every possible combination.

The page's slow, dynamic search functionality featured the heavy application of JavaScript alert boxes. Further, when a successful request returned, the client side code wrote the raw HTML to the page without navigating. This hindered the detection of page loading and made using back navigation impossible. However, AutoScrape was able to save the rendered page data due to Selenium's ability to copy the currently rendered DOM.

Spawning a separate instance of AutoScrape per search was the easiest solution to many of these problems. To deal with the difficult nature of the JavaScript powered dropdown field, we injected the port's name followed by a down and enter key. This successfully triggered all the required JavaScript functionality.

The scrape was carried out by using a pool of Selenium Grid workers to operate AutoScrape's browser interactions, greatly increasing individual scrape speed and parallel throughput. This overcame the journalist's original problems when he attempted to do this scrape using FMiner, a proprietary visual-based scraping tool. A single license costs upward of 250 dollars and the tool was only able to perform one search every four seconds. The entire FMiner scrape would have taken over five months. AutoScrape was able to complete it in 12 days by using two eight-core systems in parallel.

With a single extractor template and a parallelized batch processing job, roughly one million records of Camrail freight movements were extracted–139MB of CSV data.

Case 4: Gathering Declassified Historical CIA Documents and Metadata

History Lab is an interdisciplinary academic research collective based, in part, at Columbia University. Its goal is to apply the principles of data science to the gathering and analysis of historical records. In addition to archiving documents, they also put an emphasis on building rich metadata. This gives researchers and journalists information related to origination and chain of custody for each historical record.

One of History Lab's projects is based around declassified documents available at the CIA Electronic Reading Room web site. These records span a wide portion of the Cold War era and include Presidential Daily Briefs from the 1960s and 1970s, intelligence warnings, and military analyses. The government makes these collections available on a rolling basis, adding new documents and removing others. The researchers at History Lab were looking for a simple way to archive these documents regularly and to collect metadata related to them. Information such as publication and retrieval dates, origin URL, collection name, and datatype make up the metadata for each record. The result of this data is then to be used in data analysis, journalism, and historical research.

Scraping the Electronic Reading Room collections required only a crawl. But there were problems with performing an unrestricted DFS of the site. Many of the links available on the site lead to terrorism alert and CIA emergency contact forms–these needed to be avoided in order to not file an erroneous alert with the government.

AutoScrape was designed with the ability to rank, blanket include and exclude links by the text they contain. Downloading historical documents and navigating paginated records lists–while eliminating the possibility of interacting with hazardous parts of the site–was completed by whitelisting links containing *PDF* and *next*.

In total, documents and metadata from 21 declassified CIA archives were collected. PDF records were downloaded during the crawls and metadata was extracted from the paginated HTML release pages. A single Hext extractor was used to extract structured metadata. When combined, AutoScrape was able to construct a full database of historical documents and information related to their origination.

6 CONCLUSIONS AND FUTURE WORK

Our reporting partners found the results of AutoScrape to be useful in a variety of tasks. One reporter's project was even saved by one of the scrapes. Most importantly, we've found that the combination of text-based navigation, real-browser automation, and structured data extraction as a secondary step to be a successful framework for building robust web scrapers. Using a single tool, we've been able to extract 1,138,643 records from 14 different sites in four investigations.

Despite these early successes, there are several barriers to widespread adoption throughout the investigative journalism industry at-large. AutoScrape currently utilizes a command line interface. The configuration syntax for interacting with form fields has a learning curve. Many journalists lack the skills necessary to install the required dependencies.

These issues could be remedied by integrating AutoScrape into existing interactive scraping systems like the Computational Journalism Workbench[2]. Using a web-based environment would significantly ease the process of configuring scrape parameters and facilitate running the tool itself. Automatic construction of Hext extractors can be simplified, similarly, by integrating into a web-based interface. An implementation of client-side Hext template extraction has been included in AutoScrape. Together, these improvements could help bring generalized web scraping to a broader investigative audience and is being actively explored.

While AutoScrape is currently a manually configured tool, the resulting scrapes and DFS crawl logs could be used to train a fully-automated web scraper using reinforcement or supervised machine learning. The continual application of word embedding models[6] to programming language domains[1] could be further extended to HTML code. With this, automatic formulation of scrape plans may be possible, enabling wide-scale interactive automatic scrapes of unseen web sites. Currently, this is an active area of research being explored by Artificial Informer Labs.

The result of this ongoing work is made available as an open source project at https://github.com/brandonrobertz/autoscrape-py.

ACKNOWLEDGMENTS

The author would like to thank the reporting partners for their valuable feedback and thoughtful advice on both the output and interface of AutoScrape and Lisa van Dam-Bates for her editorial support.

REFERENCES

[1] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*. 2123–2132.

[2] Paul Bradshaw. 2013. *Scraping for Journalists: How To Grab Data From Hundreds of Sources, Put It In a Form You Can Interrogate and Still Hit Deadlines.* Leanpub.

[3] Selenium Documentation. 2013. Selenium webdriver. *Selenium HQ, Feb* (2013).

[4] Emilio Ferrara, Pasquale De Meo, Giacomo Fiumara, and Robert Baumgartner. 2014. Web data extraction, applications and techniques: A survey. *Knowledge-based systems* 70 (2014), 301–323.

[2]Computational Journalism Workbench is currently in beta at http://workbenchdata.com/.

[5] Giovanni Grasso, Tim Furche, and Christian Schallhart. 2013. Effective web scraping with OXPath. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 23–26.

[6] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[7] Jussi Myllymaki. 2002. Effective web data extraction with standard XML technologies. *Computer Networks* 39, 5 (2002), 635–644.

[8] Richard Baron Penman, Timothy Baldwin, and David Martinez. 2009. Web scraping made simple with sitescraper.