Web Voyager: Lightweight Exploration Of Semi-Structured Website Data

Kapaya Katongo* Geoffrey Litt† Daniel Jackson‡

MIT CSAIL

ABSTRACT

Websites are awash with semi-structured data that is ripe for exploration but there are two significant barriers to this: accessing the data and visualizing it. Accessing the data requires web scraping because the data is often not available in a portable format such as JSON or CSV or is only available behind an API which requires programming expertise to use. Visualizing the data requires knowledge of visualization specification formats or languages.

In this paper, we present an interaction model for lightweight exploration of semi-structured website data that does not require knowledge of web scraping or data visualization formats and languages. Our key idea is to combine web scraping via programming-by-demonstration with visualization recommendation: users scrape data from a website via mouse clicks and get visualization recommendations based on the scraped data. Unlike existing solutions, our web scraping approach supports accessing a wide variety of semi-structured website data and our visualization approach is neither limited to predefined visualizations nor does it require knowledge of data visualization.

To illustrate the proposed model, we have implemented a Chrome browser extension called Web Voyager. Through case studies, we show how our model can be used to explore data on real-world websites in order to characterize the capabilities and limitations of the approach.

Index Terms: Human-centered computing—Visualization—Visualization systems and tools—;

1 INTRODUCTION

Many websites provide valuable semi-structured data that users would like to process and visualize. This data ranges from job postings on websites like LinkedIn to user reviews on websites like Yelp. To explore data through visualization, it must be in a format suitable for computation. Most website data is either not available in a portable format (such as JSON or CSV) or is only available via an Application Programming Interface (API) which requires programming expertise to use. Even with the data in hand, visualizing it requires knowledge of data visualization formats and languages.

Prior research has resulted in tools like Vispedia [1], Reform [8] and DS.js [12] which have developed solutions to overcome some of these barriers. Vispedia automatically accesses website data via web scraping and allows users to visualize it using a predefined list of visualizations, but it only works on Wikipedia. Reform allows users to scrape a site via programming-by-demonstration (PBD) and then feed it into a predefined visualization widget created by a programmer. While the visualization widgets are not restricted to any given website, they require the exact data prescribed by their authors. This means that in order to visualize data on a website, a programmer must have created a visualization widget that accepts that type of data. DS.js comes closest to breaking down the two barriers. It automatically accesses website data via web scraping and provides a programming environment to analyze and visualize the scraped data. However, only website data in HTML table elements can be accessed, which does not cover the wide variety of semi-structured data available on many websites. Furthermore, users require knowledge of programming and data visualization in order to use the provided programming environment to explore the data.

In this paper, we present an interaction model for lightweight exploration of semi-structured website data right in the context of the website. Our key idea is to combine web scraping via PBD with visualization recommendation: users scrape data from a website via mouse clicks and get visualization recommendations based on the scraped data. No knowledge of web scraping or data visualization specification formats and languages is required. Unlike existing solutions, our web scraping approach supports accessing a wide variety of semi-structured website data and our visualization approach is neither limited to predefined visualizations nor does it require knowledge of data visualization.

To illustrate this approach, we implemented a Chrome browser extension called Web Voyager. Web Voyager combines the PBD web scraping techniques we developed for Wildcard [6] to access website data with CompassQL [9], the query language for visualization recommendation that powers Voyager 2 [11] (the inspiration for our tool's name). Our contributions are as follows:

- An interaction model for lightweight exploration of semi-structured website data via PBD web scraping and visualization recommendation, and
- An implementation of the proposed interaction model via a Chrome browser extension.

Section 2 describes a concrete scenario that shows how our model can be used to explore data on a real world website. In Section 3, we outline the implementation of the web scraping approach and our use of CompassQL to recommend visualizations based on the scraped data. To characterize the capabilities and limitations of our model, we present a suite of case studies on real world websites in Section 4]. In Section 5, we relate our approach to existing work in PDB web scraping and data visualization. Finally, Section 6 discusses opportunities for future work such as giving users more control over the visualization process.

2 EXAMPLE SCENARIO

This section describes an example scenario, illustrated in Figure 1, of exploring website data with Web Voyager. Jen is looking to change jobs and wants to explore a list of job postings on LinkedIn. She is curious about the jobs available, and wonders whether there are any trends in job title or job location but unfortunately LinkedIn does not provide access to the data outside of the website. Even if it did, Jen does not have any experience with data visualization formats or languages and her curiosity is not enough motivation to learn them. Using Web Voyager,Jen can explore the data in the context of the website with a few simple clicks.

She starts out by initiating Web Voyager through the browser context menu. This renders a panel with two sections: one for the data fields that will represent the data that will be scraped (Figure 1

*e-mail: kkatongo@mit.edu
†e-mail: glitt@mit.edu
‡e-mail: dnj@csail.mit.edu

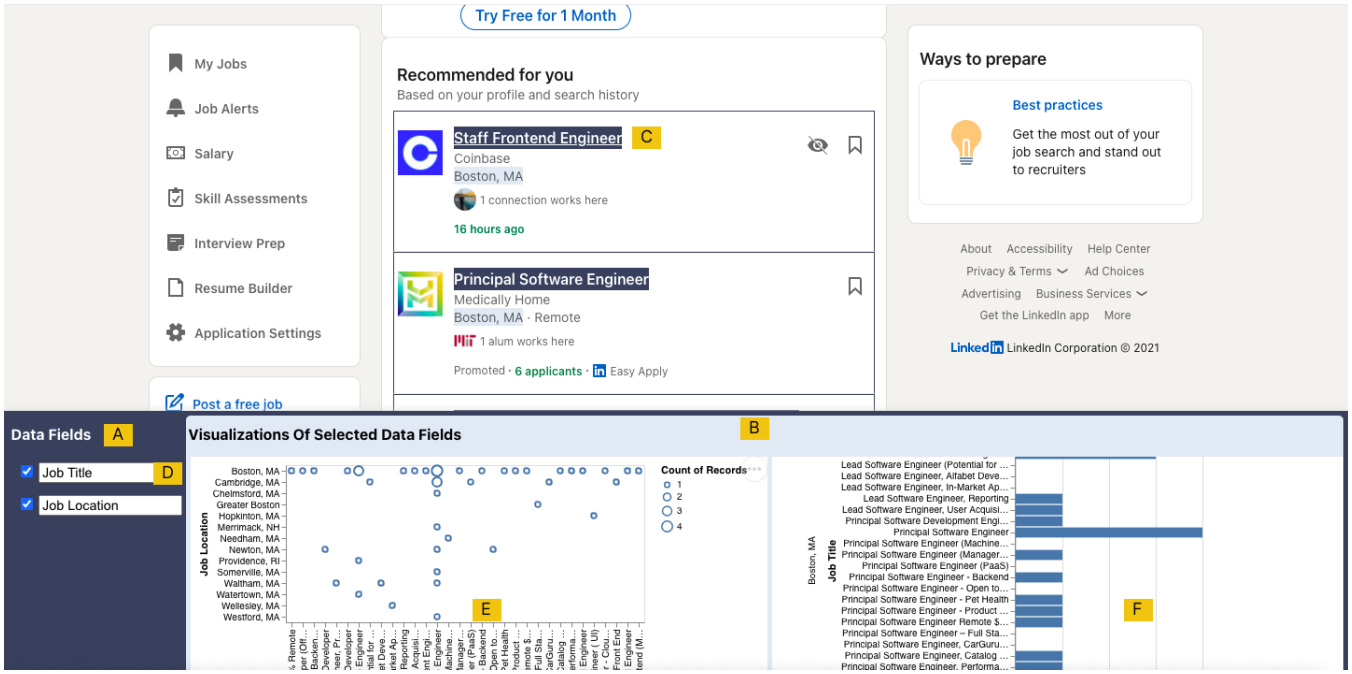

Figure 1: An overview of exploring job data on LinkedIn using Web Voyager. A) Data field section, B) Visualization section, C) Scraped job title values, D) Job title data field input and checkbox, E) First visualization recommendation based on job title and job location, F) Second visualization recommendation based on job title and job location

A) and one for the visualization recommendations (Figure 1 B). As she hovers over one of the job title values, Web Voyager highlights the job titles across all the jobs which gives her feedback about what values will be scraped (Figure 1 C). She clicks on the value to scrape all the job titles and Web Voyager adds a checkbox-activated field with the default label *A* to the data section. Because this is the first field to be scraped, it is automatically checked and results in the recommendation of a visualization based on the job title data. The recommendation is a univariate visualization with job titles on the y-axis and a count of each job title in the data on the x-axis. With this, Jen is able to easily see that the most common job title is "Principal Software Engineer."

Jen proceeds to scrape the job location values, which adds a second field (*B*) to the data section whose checkbox is unchecked. To keep track of which field corresponds to which data values, Jen clicks into the job title data field input, which shows her the corresponding values on the website by highlighting them, and changes *A* to *Job Title*. This automatically re-renders the visualization to update the y-axis label from *A* to *Job Title* (Figure 1 D). She likewise changes *B* to *Job Location*, and checks its checkbox. This results in the recommendation of two data visualizations based on both the job title and job location data. The first visualization (Figure 1 E) has job locations on the y-axis and job titles on the x-axis, with points sized by the number of jobs at a given location with a given title used as marks. The second visualization (Figure 1 F) shows a small-multiples bar chart with the count of jobs of each title in a given city. Jen continues to scrape data values and use the data field checkboxes to determine what visualizations will be recommended based on the data corresponding to checked data fields.

Using Web Voyager, Jen was able to explore the job data using a few simple clicks without any knowledge of web scraping or data visualization. In Section 4, we present a suite of case studies that show more real-world websites whose semi-structured data can be explored using our interaction model.

3 Implementation

This section outlines the implementation of our web scraping and visualization recommendation approaches. The web scraping approach is based on work we did for end-user web scraping in Wildcard [6] and the visualization recommendation uses CompassQL [9], the query language for visualization recommendation that powers Voyager 2 [11].

3.1 Web Scraping

When users demonstrate a value to scrape, Web Voyager must synthesize a program that reflects the user's general intent. This is an instance of the wrapper induction [3] problem of synthesizing a web data extraction query from examples. In the next two sections, we describe the two stages involved in this process.

3.1.1 Determing Row Elements

Given a DOM element v representing a value to scrape, we must find a set of a set of *row elements* that represent the rows of the containing data table. We could naively assume that $parent(v)$ is the row containing v, but often v is deeply nested inside its containing row; we must determine which ancestor of v is likely to be the row.

Intuitively, we solve this problem by assuming that all rows share some similar internal structure. In particular, we expect most rows to contain a value for the demonstrated column. (If there were no missing data, we'd expect *all* rows to contain data for this column.)

Formally: assume a function $select(el, s)$ which runs a CSS selector that returns the set of elements matching s within el. We generate a set of plausible candidates P, consisting of pairs of a row element and a CSS selector:

$$P = \{(r,s) \mid r \in ancestors(v) \wedge select(r,s) = \{v\}\}$$

For each candidate $(r,s) \in P$, we compute a weight function w, which is based on the number of siblings of r that have "similar structure," defined by checking whether running s within the sibling also returns a unique element.

$$w(r,s) = |\{r' \mid r' \in siblings(r) \wedge |select(r',s)| = 1\}|$$

We then choose the candidate with the highest weight. In case of ties, the candidate closer to v in the tree (i.e., lower in the tree) wins. Given a winning candidate (r,s), the full set of row elements is $\{r\} \cup siblings(r)$.

3.1.2 Synthesizing CSS Selectors For Column Values

Once we have determined the row elements, next we must choose a CSS selector that will be used to identify the demonstrated value within its row.

Given a demonstrated value v within a row element r, we generate two kinds of plausible selectors:

- selectors using CSS classes, which are manual annotations on DOM elements added by the website's programmers, typically for styling purposes (e.g. "item__price")
- selectors using positional indexes within the tree, using the `nth-child` CSS selector (e.g. `nth-child(2)`, representing the second child of an element)

The minimum criteria for a plausible selector s is that it uniquely identifies the value within the row: $select(r,s) = \{v\}$. But there may be many plausible selectors, so we must pick a best one.

We first prioritize selectors using classes, because they tend to be more robust to changes on the website and are more readable. A single selector can combine multiple classes, but we prefer using fewer classes when possible. If no plausible class-based selector can be generated (for example, if the relevant elements don't have any classes to query), we fall back to using a positional index selector. This kind of selector can always be generated regardless of the contents of the page, but tends to be less accurate and robust.

3.2 Visualization Recommendation

Web Voyager's visualization recommendation uses CompassQL, the query language for visualization recommendation that powers Voyager 2. At a high level, Web Voyager takes the selected data fields (determined by the checkboxes) and the full set of scraped data and uses them to define a CompassQL query.

Once executed, the query results in a ranked list of Vega-lite [7] visualization specifications which are used to render visualizations using Vega-lite. CompassQL provides three key features that enable this visualization recommendation based on a list of selected data fields and the complete set of data. The sections below describe each of the three features.

3.2.1 Specification

CompassQL queries are defined using Vega-lite's visualization grammar with an enumeration token to indicate which properties should be enumerated to generate multiple visualization specifications. Web Voyager creates a query with the values of the `mark` and `channel` properties set to an enumeration token (`?`). This instructs CompassQL to generate visualization specifications based on the enumeration of marks (`bar`, `point`, `circle`, `line` etc) and channels (`x`, `y`, `size`, `color` etc).

3.2.2 Choosing and Ordering

CompassQL queries can define properties that determine how to choose the top visualization specification (`chooseBy`) or how to order visualization specifications by a ranking criteria (`orderBy`). Web Voyager creates queries with the value of `orderBy` set to `effectiveness`.

3.2.3 Grouping and Nesting

CompassQL queries can define `groupBy` and `nest` properties to reduce redundancy in the list of resulting visualization specifications. The redundancy stems from the fact that there may be many visualizations with similar encodings or data. Web Voyager creates queries with `groupBy` and `nest` that group and nest visualization specifications by properties such as `channel` and `field`.

4 Evaluation

In this section, we characterize the capabilities of our interaction model through a suite of case studies on real world websites. Then, we outline the limitations of our web scraping and data visualization approaches.

4.1 Case Studies

We used Web Voyager to explore data on a collection of real world websites that we list in Table 1. The websites were chosen to showcase the variety of supported semi-structured data available on websites.

4.2 Limitations

In this section, we outline the limitations of our web scraping and data visualization approaches using real world websites as references.

4.2.1 Web Scraping

Web Voyager' web scraping approach is most effective on websites whose data is presented as a collection of similarly-structured HTML elements. Certain websites, however, have designs that make it difficult to scrape data:

Heterogeneous Row Elements. Websites like HackerNews break their content into rows, but the rows do not have a consistent layout, and contain different types of child elements. Because Web Voyager only chooses a single row selector, when scraping by demonstration, it will only select one of the types of rows, and elements in the other types of rows will not be scraped.

Infinite Scroll. Websites like LinkedIn have an "infinite scroll" feature that adds new entries to the page when a user scrolls to the bottom. As a result, the data scraped by Web Voyager will only contain values that were rendered when the scraping was performed. This can be remedied by providing a UI for users to indicate when data should be re-scraped (such as when the page is scrolled). In general, dynamically loaded data poses a fundanmental challenge to our approach.

4.2.2 Data Visualization

Our visualization recommendation approach is most effective for lightweight, open-ended exploration of nominal and quantitative data. The features that are useful for focused exploration are available in CompassQL but simply haven't been implemented yet. Below are the current key limitations:

Data Field Types. Vega-lite [7] specifications support four main data field categories: nominal, quantitative, temporal and ordinal. The current implementation of Web Voyager only designates data fields as either nominal or quantitative. The lack of temporal and ordinal field categories therefore limits the range of possible visualization recommendations. This can easily be remedied with a UI to allow users to update field categories.

Manual Visualization Specification. Users currently do not have any control over the visualization process beyond selecting which fields to visualize. This limits the effectiveness of Web Voyager for focused exploration that involves specifying what encoding channel, mark or facet the desired visualization should use. This can be remedied with a UI similar to that of Voyager 2 [11] as CompassQL [9] already provides the required functionality.

Website	Example exploration of semi-structured data

Table 1: Websites with semi-structured data that Web Voyager can be used to explore

5 Related Work

Our work builds on existing research in end-user web scraping and data visualization across a number of tools and systems.

5.1 End-user Web Scraping

The web scraping approach used by Web Voyager is inspired by that of Vegemite [5], a tool for end-user programming of web mashups. Like Web Voyager, Vegemite enables end-users to scrape data from a website into a tabular format via simple mouse clicks. Similar web scraping approaches can be seen in tools like Rousillon [2] and FlashExtract [4].

5.2 Data Visualization

Web Voyager uses CompassQL [9], the query language for visualization recommendation that powers Voyager 2 [11], for its visualization recommendation. The current implementation of Web Voyager is more similar to Voyager 1 [10] than Voyager 2 due to its lack of support for manual visualization specification. More broadly, Web Voyager relates to tools like Vispedia [1], Reform [8] and DS.js [12] that enable visualization of website data. Of the three, DS.js comes the closest to enabling lightweight exploration of semi-structured website data.

DS.js' core feature is a programming environment to analyze and explore website data right in the context of the website. The data is accessed via automatic scraping of HTML tables and CSV links on a website. While DS.js provides a more extensive set of features for visualizing website data, it can only scrape data in HTML tables which does not cover the wide variety of semi-structured data on websites. Furthermore, users require knowledge of programming and data visualization in order to use the provided programming environment to explore the data.

6 Conclusion and Future Work

In this paper, we presented an interaction model for lightweight exploration of semi-structured website data that does not require knowledge of web scraping or data visualization formats and languages. Our key idea is to combine web scraping via programming-by-demonstration with visualization recommendation.

The main area of future work is giving users more control over the visualization process. The current implementation emulates the visualization capabilities of Voyager 1 [10]: users can determine what data to visualize but not how it is visualized. While this is desirable for open-ended exploration, it is not sufficient for focused exploration. Voyager 2 [11] provides some inspiration for combining automatic and manual chart specification that we will explore in the next iteration.

References

[1] B. Chan, L. Wu, J. Talbot, M. Cammarano, and P. Hanrahan. Vispedia: Interactive Visual Exploration of Wikipedia Data via Search-Based Integration. 14(6):1213–1220. doi: 10.1109/TVCG.2008.178

[2] S. E. Chasins, M. Mueller, and R. Bodik. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pp. 963–975. ACM. doi: 10.1145/3242587.3242661

[3] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. 118(1-2):15–68. doi: 10.1016/S0004-3702(99)00100-9

[4] V. Le and S. Gulwani. FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 542–553. ACM. doi: 10.1145/2594291.2594333

[5] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces*, pp. 97–106. ACM. doi: 10.1145/1502650.1502667

[6] G. Litt, D. Jackson, T. Millis, and J. Quaye. End-user software customization by direct manipulation of tabular data. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 18–33. ACM. doi: 10.1145/3426428.3426914

[7] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-Lite: A Grammar of Interactive Graphics. 23(1):341–350. doi: 10.1109/TVCG.2016.2599030

[8] M. Toomim, S. M. Drucker, M. Dontcheva, A. Rahimi, B. Thomson, and J. A. Landay. Attaching UI enhancements to websites with end users. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems - CHI 09*, p. 1859. ACM Press. doi: 10.1145/1518701.1518987

[9] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Towards a general-purpose query language for visualization recommendation. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics - HILDA '16*, pp. 1–6. ACM Press. doi: 10.1145/2939502.2939506

[10] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. 22(1):649–658. doi: 10.1109/TVCG.2015.2467191

[11] K. Wongsuphasawat, Z. Qu, D. Moritz, R. Chang, F. Ouk, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pp. 2648–2659. ACM. doi: 10.1145/3025453.3025768

[12] X. Zhang and P. J. Guo. DS.js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pp. 691–702. ACM. doi: 10.1145/3126594.3126663