ComCrawler: An End-to-End User Comment Crawler

Zhijia Chen
Computer & Information Science
Temple University
zhijia.chen@temple.edu

Weiyi Meng
Computer Science
Binghamton University
meng@binghamton.edu

Eduard Dragut
Computer & Information Science
Temple University
edragut@temple.edu

***Abstract*—User comments posted on many websites is a prominent form of online participation. There is growing interest in mining and studying user comments in applications, such as opinion mining, fake news detection, and LLM training. This paper addresses two critical challenges in crawling user comments from diverse websites at scale: entry point discovery and comment detection. Comment sections on Web pages are not commonly visible by default, but triggered by a user action, e.g., clicking an HTML element among a large number of clickable elements. Furthermore, comments are structured record-like Web data with high structure variations, making them difficult to detect and extract from the target Web page where other record-like Web data may exist. We present ComCrawler, an end-to-end automatic comment crawling solution composed of three components: entry point discovery, web record extraction, and comment section detection. Entry point discovery identifies the event prompting comment loading, while web record extraction detects and extracts record-like Web data regions. The comment section detection component distinguishes user comments from other data regions. Empirical testing on hundreds of geographically diverse websites reveals that, on average, ComCrawler explores 5.56 clicks (2% of all possible clicks) to expose the comment section. The system achieves a high end-to-end comment detection F1 score of 0.95 in offline evaluations, with ideal conditions. However, when testing it online, the F1 score decreases to 0.41, prompting further exploration in deployment. To address practical challenges like comment scarcity during visits, we propose a principled re-visiting framework based on queuing theory. ComCrawler demonstrates the capability to achieve a high 0.90 F1 score, closely approaching performance levels in ideal conditions.**

***Index Terms*—crawler, comments, Web records**

I. Introduction

Many websites take full advantage of Web 2.0 technologies to host multimedia postings and comments, transforming their audiences into active content contributors on their websites. User comments are a standard feature at many websites and are considered one of the most popular forms of public online participation [1]. Social scientists argue that commenting platforms increase user-to-user interactions and contribute to shaping a democratically valuable and vivid interpersonal discourse on topics of public interest [2]–[4]. Take news websites as an example, there are over 50K news websites in the world [5], and a large fraction of them have over 100K subscribers who actively comment [6]; together, they amass tens of millions of users who produce vast volumes of messages every day. User comments power a broad range of applications, like opinion mining [7], fake news detection [8], [9], user engagement and behavior analysis [10], which attract relentless attention from industry and academia alike. Such data is also used to generate conversational systems [11] and to enrich LLMs [12].

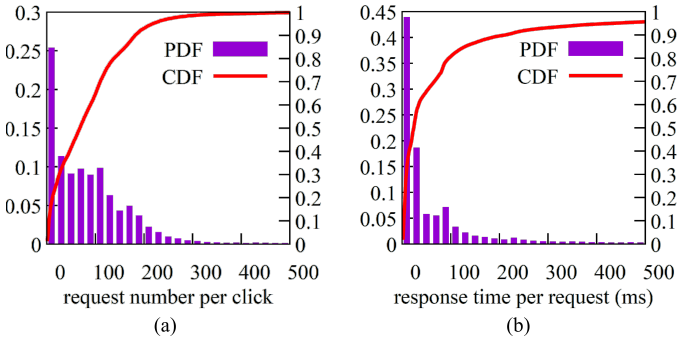
The problem we aim to solve in this work is: Given a (random) Web page, determine if the page hosts user comments and, if it does, locate and retrieve the comments.

One solution is to build crawlers tailored to specific websites; this is labor-intensive and does not scale to thousands of websites. A more sophisticated solution may apply wrapper induction/program synthesis techniques to infer wrapper programs using labeled examples from the target websites [13], [14]. But still, labeling thousands of websites is not a trivial task, let alone the wrapper maintenance challenges imposed by the changes in website templates that can break the wrappers [15], [16]. Furthermore, these solutions do not address the problem that user comments are dynamically loaded on the modern Web by specific triggering events, and websites employ various means to trigger and display user comments associated with a post. For illustration, we show three examples in Figure 1. The user comments on Fox News are loaded when the window is scrolled down to the comment section (Figure 1a). The New York Times and Tencent News load comments at the click of a comment loading element, but the former generates a modal popup (Figure 1b), whereas the latter presents the comments in a new Web page (Figures 1c and 1d). Such diversity in loading and displaying user comments requires a comprehensive crawling solution that it is able to trigger the comment loading event (i.e, entry point detection), extract Web records from a page, and detect the comment section. We describe these components of ComCrawler below.

Entry point discovery. The first component of ComCrawler addresses the problem of finding the *entry point* to dynamic comment sections, i.e., the HTML event that triggers comment loading. This is generally a user event, such as scrolling the comment section into the browser window or clicking an HTML element, e.g., a button. While the scrolling case can be solved by instructing the browser to scroll over the entire target page without knowing the exact location of comments, the clicking event is much more challenging because a Web page may have thousands of clickable elements

(a) Fox News (b) New York Times (c) Tencent News (article) (d) Tencent News (comments)

Fig. 1: Typical ways of loading comments dynamically.



(a) (b)

Fig. 2: Distribution of request number per click (a) and response time per request (b).

and an exhaustive enumeration of these elements will not only slow down the crawling process but also incur an unreasonable resource drain on a website's server. To understand the cost of a brute force search of the (unique) comment loading element, we performed an experiment to click all the clickable elements of 1,500 web pages of different websites where comments are loaded by clicking a specific HTML element, and then we record the number of requests sent after each click and the response time per request. We show the distribution of requests sent by a click and the response time of a request in Figure 2 (a) and Figure 2 (b), respectively. We notice that on average, each click sends 83 requests with the mean response time of 102 ms per request, which means that on average each click will take 8.5s without parallelism.

Web record extraction. The second component of ComCrawler aims to detect and extract the comments on the result webpage after triggering a comment-loading event. There are two key challenges – organizing the webpage into sections of distinct Web records (aka data regions [17]) on the page and detection of the section/region with user comments. We discuss the latter first because it implicitly organizes the page into sections of Web records of the same type, like ads versus comments. Comment extraction poses unique challenges. As shown in Figure 1, comments are generally organized as a structured record-like section on a webpage where each comment may be treated as a data record. Thus we can treat the comment extraction problem as a Web record extraction problem [18]. A key challenge here is that the page may contain multiple sections with Web records, e.g., records of ads. Existing *unsupervised* Web record extraction techniques largely try to find repeating patterns in the target DOM tree, assuming that records in the same group have very similar subtree structures [19], [20] and that Web records are organized in a flat list-like manner (i.e., records are linearly listed one after another). This assumption is often not true for comments because their DOM subtree structures tend to be more complex, with less regularity due to nesting replies (as shown in Figure 1a) and rich formatted contents (Figure 1d).

Comment section detection. The previous component partitions a Web page into sections of Web records. A Web page often contains multiple Web record sections such as related to ads and news articles, like the records (hot articles) at the right column of Figure 1(c), which are difficult to distinguish from comments based only on their DOM tree structures. This component aims to detect the section with user comments, if one exists.

ComCrawler is a linear ensemble of the three components: 1) an entry point discovery component that detects the correct event (or click element) that exposes the user comment section, 2) a Web record extraction component that finds candidate sections of list-like record within the result page, and 3) a comment section detection component that determines which one of the candidate sections contains user comments. 1) and 3) are implemented as classifiers and leverage the text features extracted from the HTML element attributes, whereas 2) relies on the HTML structural features. Figure 3 shows the workflow of ComCrawler: on the right-hand side are the three major components, and on the left-hand side are screenshots of real examples of the critical steps. Comments may be statically loaded (loaded with a Web page) or dynamically loaded (loaded when users trigger a particular event). We focus our presentation on the dynamic case since applying

the solution to the static case is straightforward.

An additional challenge in designing ComCrawler is that different components of ComCrawler have different optimization criteria. The comment section detection and Web record extraction components optimize accuracy, i.e., find the section with user comments among all sections and correctly identify each comment (some baseline tools do not find the correct boundaries of a comment), respectively. For the entry point detection component, the optimization goal is to avoid missing correct entry points while being as *polite* as possible. In other words, we would like to achieve a high *recall* of correct entry points while achieving a good *precision* by avoiding trying too many unnecessary clickable elements on the page. Clearly, if we simply try all clickable elements, we can achieve the perfect recall (1.0) but this will lead to very bad precision (i.e., poor politeness). On the other hand, too much politeness may lead to lower recall of correct entry points.

ComCrawler achieves high accuracy, F1 score = 0.95, on our *offline* testing dataset with simulated Web responses, while only exploring 5.56 clickable elements on average.

Deployment. When we deployed ComCrawler *online* with news articles from Google News as input, we saw a disappointing accuracy with F1 score = 0.41. After studying the reasons, we discovered that the key reasons were due to insufficient comments at the time of visiting. For example, many Web pages had only very few comments when they were visited by ComCrawler. ComCrawler was not effective for pages with less than 10 comments. Other significant causes of failure were timeouts, comment access restrictions due to subscription requirements and page-blocking objects, such as cookie acceptance forms and ads. All these issues point to another challenge – the need of a principled (re)visiting protocol of a Web page. Comments are the result of user commenting behavior. We characterize the behavior by measuring the delay between the time an article is published and the arrival of user comments. It yields a Poisson distribution. We endow ComCrawler with a queuing system whose policy is informed by the observed user commenting behavior. ComCrawler with the queuing system achieves an F1 score of 0.90 in deployment, which is only slightly off from the upper bound of 0.95. We make the following contributions in this work:

- We introduce the problem of crawling dynamically loaded comments on the Web.
- We propose an end-to-end solution, ComCrawler, that is effective in retrieving comments while being polite to the server, and it works well across different website designs and languages.
- We conduct experiments to assess the performance of ComCrawler in offline and online settings. It achieves an F1 score of 0.95 in the former and 0.90 in the latter.

II. Preliminaries

An ***HTML element*** is the basic unit of an HTML document. It consists of a start tag, text content and an end tag; an element may have nesting elements between the start and end tags. The start tag may carry several pairs of attribute names and values

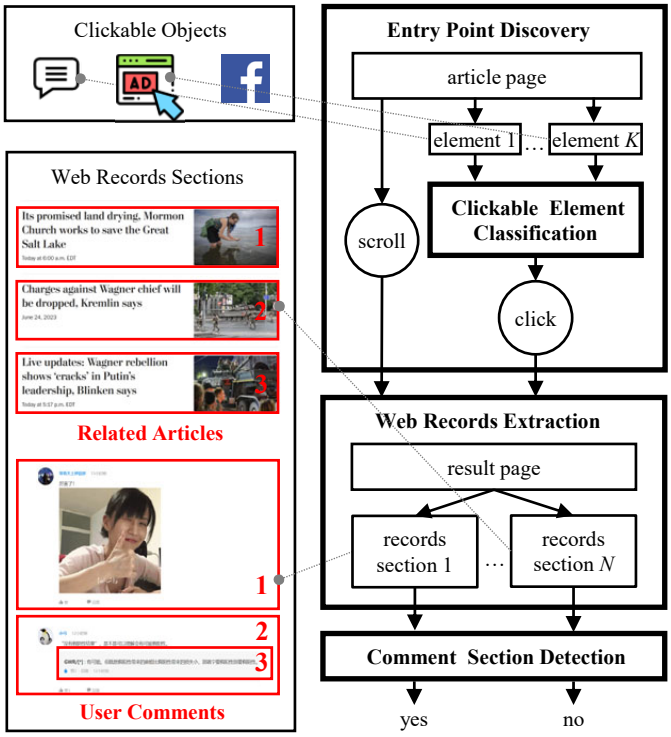


Fig. 3: Comment crawling workflow.

which are not visible to the user, but together they dictate how the browser formats and displays the content to the user. Consider the element example of `<div>` tag:

```
<div><button class="btn">text</button></div>
```

It contains a nesting `<button>` element which has content text "text", attribute name "class" and the corresponding attribute value "btn".

A ***DOM tree*** is a tree structure representation of an HTML document wherein each node represents an element in the document. The DOM tree structure is commonly exploited for Web Records extraction [17], [19], [21]. A comment is a **subtree** of a DOM tree that represents a piece of User Generated Content (UGC). As shown by the examples in Figure 1a, a comment may have nesting replies where each reply is another subtree of comment under the root comment, and a reply may also have replies. A group of continuous subtrees of comments forms a comment section.

Comment Loading Event. Using dynamic Web development techniques (e.g., AJAX), a Web browser may load comments at an arbitrary point during the life of a Web page [22]. We observe that comments are loaded either when the comment section is scrolled into the browser window or when a designated HTML element is clicked. The former event can be triggered by instructing the browser to scroll over the entire target page, while the latter requires careful consideration in searching for potential HTML elements to optimize both accuracy and efficiency.

Clickable Element. For a crawler to automatically locate the HTML element that loads the comments, it needs to find

all the clickable elements in the DOM tree. While modern Web browsers allow any element in a DOM tree to be a clickable element by adding an event listener, the <button> tag and <a> tag are two standard tags that expect click events. We performed a survey of 150 news websites that have comments, and count the frequency of tags used to implement the comment loading event. As shown in Table I, 92% of them are implemented with the two tags. One may apply advanced programming tools to monitor all the click event handlers associated with a DOM tree to cover those 7.6% minor cases, but for sake of simplicity, we focus our presentation on the <a> and <button> tags.

TABLE I: Distribution of tags used to implement comment loading element.

Tag	Percentage(%)
A	80.00
BUTTON	12.00
DIV	6.66
SPAN	0.67
LI	0.67

A ***Web Record Section*** is a page section that contains continuous list-like Web records from the same underlying data schema (SQL or non-SQL) and serves the same consumption purpose. From the perspective of the underlying DOM tree, a Web record section is composed of a cluster of sibling subtrees with a similar subtree structure. As we will discuss later, the subtree similarity may be measured using the whole subtree of a Web record, or using some common components of Web records such as the user avatar of a comment or the cover image of a related article.

III. Classification Modules

We present the entry point detection and section classification components first and leave the Web record extraction component to the next section, as the first two components use similar methods and both leverage the text features of HTML source code.

A. Entry Point Detection

One critical policy for effective Web crawling is politeness [23], that is, avoiding overloading target websites with unnecessary requests. Because a click may generate hundreds of requests and take several seconds to get a response (see Figure 2), we build a clickable element classifier to reduce the unfruitful clicks. A fundamental task in filtering out unfruitful clickable elements is distilling the differences between HTML elements that load comments and those that do not. We observe that the text content and the attribute values of a clickable element are likely to present strong hints about its purpose. More specifically, the text content of a clickable element indicates its functionality to users, e.g., "log in", "share", and "comment", while the element attributes such as `id`, `name` and `class` are popular places for programmers to include hints about the functional purpose of an object.

Listing 1: The HTML source codes of clickable elements for login, share to Facebook and comment.

```
<button data-testid="login-button" >Log In</button>
<a href="https://www.facebook.com/..." aria-label="
    Share on
    Facebook"></a>
<button id="comments-speech-bubble-top"></button>
```

For example, Listing 1 shows snippets of HTML source code from a mainstream website for clickable elements that, when clicked, will open a login box, share to Facebook, and go to a comment section, respectively. One notices that the attribute values include meaningful keywords, such as "log in", "Facebook" and "comment" that indicate the functionality of the clickable elements. It is thus tempting to construct text features for the classifier by manually selecting keywords such as "comment" or related words. Such features are not comprehensive enough and may miss misspellings, abbreviations, or keywords in other languages. Examples include "commenst", "cmt", and the German "kommentar" for "comment". We use character-based 3-gram [24] to extract text features. Before generating the 3-gram features, we pre-process the HTML source code of each clickable element by extracting the text and attribute values and discarding all non-alphabetic characters. For illustration, in Listing 1, we only keep characters in bold.

We use the fastText (https://fasttext.cc) library to train the comment element classifier in our framework. Under the hood, fastText learns a linear model with rank constraint and a fast loss approximation [25]. We choose the library for its fast inference speed and light memory consumption, which is vital for the crawler considering our ambition of global-scale crawling. We carefully modularize our framework so the classifier can be easily swapped with other more recent but heavier deep learning models.

The classifier gives us a subset of clickable elements as comment loading candidates. The next step is to click each of the candidates and analyze the content of the result page.

B. Comment Section Classifier

The comment section classifier takes the HTML source codes of a Web record section as input, which is provided by the Web records extraction component (Section IV), and it outputs a binary label indicating if the Web record section is a comment section or not. In the example of Web record sections shown in Figure 3, one notices that the page containing user comments may have other Web record sections, such as a list of articles. Those record sections will also be detected by our Web record extraction module, and we apply the comment section classifier to determine the right one. One may wonder why not work on a Web record extraction algorithm that will only extract comments, and there are several important reasons behind our design choice. First, it is difficult to separate comments from other Web records solely based on structure features. Although the structure features of nesting replies and rich format texts can help distinguish comments from other regular Web records, there is still a significant number of websites rendering comments in a simple way, like regular

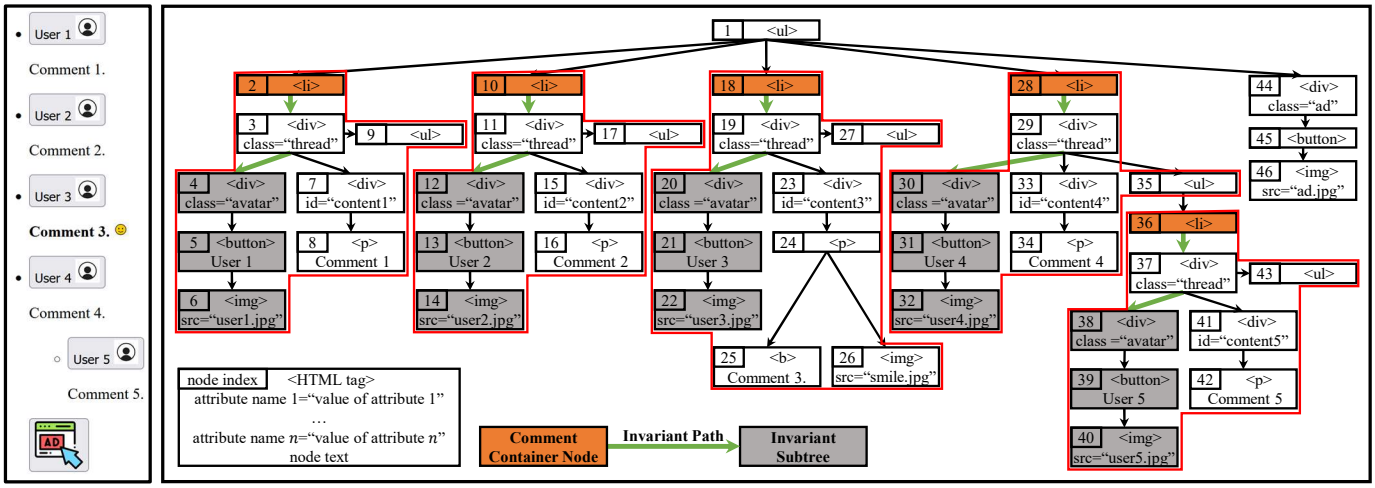


Fig. 4: Examples of comment DOM subtrees.

Web records. Second, incorporating multi-modal features – i.e., the structure features used by the Web record extraction module, the text feature used by the comment section classifier, and potentially image features used by existing Web record extraction works [17], [20], [21] – inevitably leads to a complex and heavy model that we try to avoid in the first place. Finally, decoupling the task into two independent steps enables us to improve each one of them separately.

We build the comment section classifier using the fastText model that takes the n-gram features of the HTML source codes as input. Our initial attempt used both text contents and the attribute values, the same as the comment element classifier based on the observation that the text contents in a comment section contain many keyword hints, such as "comment", "thread" and "reply". However, our experiments showed that this method performed poorly across languages. The issue is that apart from those desirable terms, the majority of the text contents are UGCs (including non-English), which introduce noise to the classifier. In addition, unlike the attribute values in the HTML source code that are mostly written in English (English is the lingua franca of programming [26]), the text contents are not, because they are for user consumption and follow the website's language. Thus, we only generate n-gram features from HTML attribute values for the comment section classifier.

IV. Web Record Extraction

After each click of a comment entry point candidate, we search for potential comment sections in the result page. As shown in Figure 1, comments are generally well structured contents arranged in a list-like manner. Hence, we may treat comments as a type of Web record with the following properties: a comment may contain (i) rich format texts and multimedia content, and (ii) one or more nested replies.

These unique features introduce significant structure irregularities to the underlying DOM tree structure leveraged by the existing Web record extraction methods, leading to unpredictable performance. For illustration, we make a synthetic running example containing five simplified comments (and an ad as noise) as shown in Figure 4, where the left part shows the rendered presentation in the browser and the right part shows the corresponding HTML DOM tree. *We use $\mathbb{E}$ to denote the DOM tree of this example for the rest of this paper.* Each node of $\mathbb{E}$ stands for an HTML element, with the tag enclosed in angle brackets, the attribute specified using an equal sign, and the text content placed at the bottom (if it exists). Each node is associated with an index in the left or top left, *and we use $\mathbb{E}[i]$ to refer to the i_{th} node and $\mathbb{E}(i)$ to refer to the subtree under $\mathbb{E}[i]$.* Our goal is to identify the root node of the subtree of each individual comment so we may extract the UGCs correctly. We call the root node of a comment as **Comment Container Node**, which is colored in orange.

We attempted to solve this step with several existing representative Web record extraction methods [17], [21] (Section V-B). However, even though these methods were designed to overcome structure variations among Web records, we still found their performance highly sensitive to the dynamic structure of comments. Specifically, these methods tend to focus on the DOM tree structure and try to align similar subtrees of Web records, which fails when there are complex comment contents or nested replies. Some works try to detect Web records from the signal processing perspective by transforming an HTML document into a sequential signal [19], [20] and looking for regions of recurring periods, which are also unstable due to the noise introduced by the complex comment contents or nested replies.

We identify that the main cause of errors is that the subtrees of the individual comments do not have record-level similarity. Nonetheless, they do contain similar sub-record components, which may be leveraged to locate and extract the comments. Based on that observation, we create a new Web record extraction method by searching Web records in a bottom-up manner using those common components, which we name record invariants. For a self-contained presentation of ComCrawler, we give here an overview of the extraction

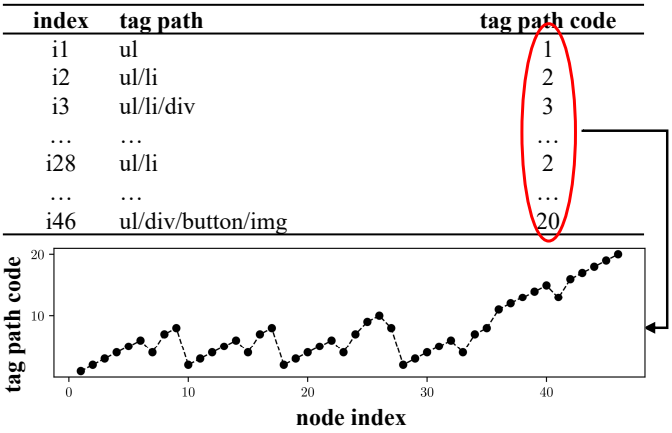


Fig. 5: Tag path code sequence of $\mathbb{E}$ (Figure 4).

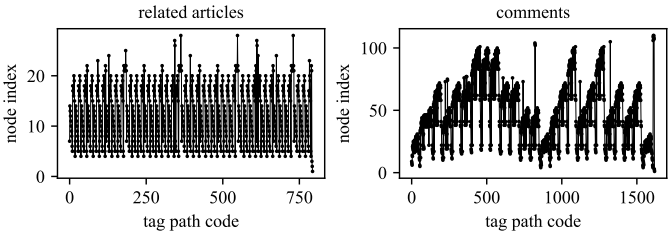


Fig. 6: Tag path code sequence of related articles (left) and user comments (right) on a Web page.

method [27].

A. Signal of Web Records

Inspired by existing works that perceive Web records as repeating tag sequences [19], [28], [29], we transform an HTML doc into a one-dimensional signal and study how Web records may be represented as a signal. More specifically, we take the DOM tree of a Web page and flatten it into a sequence of nodes, and then we represent each node by the identifier of its *tag path*, which is the concatenation of the HTML tags of the ancestor nodes and itself. For example, Figure 5 shows the tag paths of the nodes of $\mathbb{E}$. We can assign a code for each unique tag path, and then we get a sequence of tag path codes for the tree, as shown at the bottom of Figure 5. Details of the process can be found in [29].

We compare the sequential signals of regular Web records against those from comments. Figure 6 shows the signal plots of a related article section (left) and a comment section (right) that we found in the same page. We notice that the signal of the related articles section is periodic with minimal irregular variations. In contrast, there is no apparent repeating period in the signal of the comment section. Nonetheless, we observe that there are still some (sub)patterns repeating throughout the signal and wonder what contributes to these patterns and whether we can leverage them to extract complex Web records such as comments.

B. Web Record Invariants

We map the frequent patterns of the signal of comments back to the original DOM tree, and we find they generally correspond to some common components among the comments, such as avatars, likes/dislikes, and posting dates. We observe that compared to the whole record-level similarity, common components of Web records that are represented by identical DOM tree structure – i.e., the sub-record-level similarity – is a more general and stable feature of complex Web records like user comments. We define two types of common structures among Web records named *record invariants*.

Invariant Subtree: a common subtree structure that appears in every record, representing the same component. Invariant subtrees may come from data attributes of Web records that are rendered by the same template, like the Posting Date of a user post on a social media platform. They may also be parts of the Web record template that is not sensitive to the individual data record, such as the Add to Cart button that is commonly seen on e-commerce Web sites. We use invariant subtrees as landmarks to locate potential Web records. In our running example (Figure 4), the subtrees in gray color that carry user avatar are instances of an invariant subtree.

Invariant Path: a constant tag path between a Web record's container node to the occurrences of an invariant subtree. Existing works assume that subtrees of Web records are under the same parent node, and thus all the record container nodes have the same tag path. We can no longer expect such a property with the presence of nested records; however, the tag paths within each record subtree remain stable regardless of the nesting structure. So we try to find an invariant path between invariant subtrees and their corresponding record container nodes. To illustrate, we mark the invariant path of the running example with a green arrow. We use invariant subtrees as landmarks to locate Web records on the target DOM tree, and then we find their corresponding record container nodes detecting the potential invariant path.

C. Mining Frequent Patterns

One may try to detect invariant subtrees by leveraging domain-specific record attributes (such as posting date of comments) or record components (such as Add to Cart button of e-commerce products) that are expected to appear in all records [21]. However, the method may not generalize well across different websites, and such attributes or components may be difficult to find or not even exist.

We propose detecting invariant subtrees based on frequent pattern mining that is not limited to a specific domain or website. Notice that in Figure 6 (right), the signal of comments contains subsequences that form the same variation pattern, which is contributed by candidates of invariant subtrees on the original DOM tree. However, these patterns have very different values (heights), making it difficult to extract the patterns from the signal. This is because a node is represented by its HTML tag path, which depends on the node's ancestor nodes.

Intuitively, if we can represent a node in such a way that nodes with the same subtree structure are represented by the

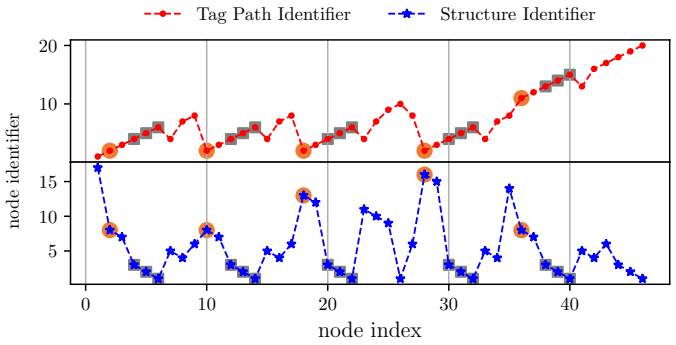


Fig. 7: Transforming the running example (Figure 4) into a signal by representing a node using an HTML tag path and subtree structure, respectively. The record container nodes are marked in orange circles, and the nodes of invariant subtrees are marked by gray squares.

same value and independent of its ancestor nodes, then we can expect the invariant subtrees to form the same code sequence and thus retrievable with frequent sequential pattern mining techniques. Naturally, we represent a node x by the identifier of its subtree structure $struct(x)$, which can be calculated recursively by:

$$struct(x) = \langle x.tag, x.attrib, struct(x.children) \rangle$$

where $x.tag$, $x.attrib$ and $x.children$ are the tag, attributes, and children of x, respectively. For example, the output of $struct(\mathbb{E}[4])$ is:

$$\langle \texttt{div}, [\texttt{class}], [\langle \texttt{button}, [], [\langle \texttt{img}, [\texttt{src}], [] \rangle] \rangle] \rangle$$

Note that $x.attrib$ and $x.children$ are arrays, and we use $[\cdot]$ to enclose their elements. We only include HTML element attribute names but not values in the structure representation because some values like "id" or "href" are generally unique to individual elements.

Figure 7 shows the signal of the running example generated by the HTML tag path and our structure representation, respectively. We see that all the invariant subtrees (highlighted in gray squares) are represented by the same sequence when using the structure representation, whereas the invariant subtree under the nested record (the last one) has different values than others when represented by the HTML tag path.

Using the structure representation, we make sure all the occurrences of an invariant subtree are represented using the same subsequence and thus transform the invariant subtree searching into a classic pattern mining problem, which can be solved by building a suffix tree. The suffix tree is a compressed trie of all the suffixes of a given string (or general sequence). It enables fast implementations of many important string operations, such as searching for repeated substrings or common substrings in $O(N)$ time [30].

We may use a suffix tree to find any patterns that have more than one occurrence. However, a Web page generally contains many trivial repetitive structures, such as page menus and paragraphs, which may lead to a huge search space and interfere with frequent patterns in comments. For example, they may happen to be a subpattern of a larger pattern from comments, leading our algorithm to use the more frequent subpattern to find the invariant subtree. Thus we set a pattern frequency threshold (F_t=10) and a pattern size threshold (F_s=10) to make sure we detect potential invariant subtree with significant structure complexity.

D. Web Records Reconstruction

After frequent pattern mining, we map the occurrences of each pattern back to the original DOM tree to get a group of candidate invariant subtrees (note that a Web page may contain multiple data regions). Then we reconstruct Web records in a bottom-up manner, searching record container nodes by matching potential invariant paths starting from each group of candidate invariant subtrees. We will illustrate the process using the running example.

We begin with the most frequent pattern $\langle 3, 2, 1 \rangle$, and the occurrences are mapped back to six candidate invariant subtrees $\mathbb{E}(4)$, $\mathbb{E}(12)$, $\mathbb{E}(20)$, $\mathbb{E}(30)$, $\mathbb{E}(38)$, and $\mathbb{E}(44)$ on the original DOM tree (Figure 4). Notice that we may have false positives when detecting invariant subtree with frequent patterns, like $\mathbb{E}(44)$ (which represents an ad) in this example. Then we try to reach their corresponding record container nodes through the potential invariant path by matching their ancestor nodes. In the first round, all the invariant subtree candidates except for $\mathbb{E}(44)$ have identical ancestor node (`div` tag with `class` attribute), so we discard $\mathbb{E}(44)$. In the next round, all the ancestor nodes of the remaining candidates are of `li` tag. And in the last round, we reach to a common ancestor node of all the candidate invariant subtrees $\mathbb{E}[1]$, which means that we have reached the end of invariant path. Thus we determine that $\mathbb{E}(1)$ represents the whole record section, and the subtrees at the last group of ancestor nodes, $\mathbb{E}(2)$, $\mathbb{E}(10)$, $\mathbb{E}(18)$, $\mathbb{E}(28)$, and $\mathbb{E}(36)$ are detected records.

In our running example, there is only one section of Web records and it is of our interest, i.e., comments. However, in practice, a Web page may contain multiple sections of Web records. Our method detects each section of Web records solely based on the DOM tree structures and does not distinguish the records according to their semantics. The comment section detection module introduced in Section III-B will detect sections that contain comment records.

V. Offline Evaluation

In this empirical study, we evaluate ComCrawler offline with ideal settings, that is, all things go right, e.g., every page has sufficient comments and is loaded properly. We test ComCrawler on datasets with simulated Web responses, and we study the performance of each individual module.

A. Data

For entry point detection, we collect a comment-loading element dataset comprising 1,500 positive clickable elements and 1,500 negative clickable elements. We first find 1,500 pages where the comment entry point is a clickable element, uniformly distributed over 150 websites of different languages

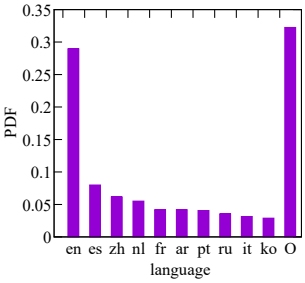


Fig. 8: Distribution of top-10 languages in our dataset. O is for Others.

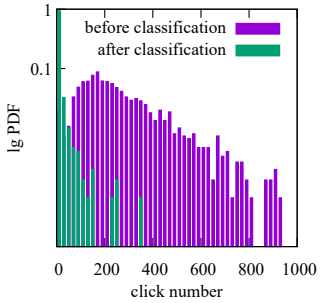


Fig. 9: Distribution of clickables per page.

and countries/regions. Then for each page, we collect the comment loading element and a random negative clickable element. To guarantee labeling accuracy, we write strict XPath (XML Path Language) expressions to extract the comment-loading elements for each website. We name this dataset **ClkElmSet**.

For the task of Web record extraction and comment detection, we manually collect 2,000 pages with each having more than 10 comments and 2,000 pages without comment, uniformly distributed over 100 websites of different languages and countries/regions. For each page, we open it in the browser to check if it has comments, and we manually trigger the event to display user comments. We then save the HTML document of the rendered page and mark the record container nodes of each comment and the container node of the comment section. We name this dataset **CmtSecSet**.

For both datasets, we carefully balance the source websites over several countries/regions. We show the language distribution of the two datasets in Figure 8. The collection and annotation of the two datasets took four weeks for a professional.

B. Baselines

There are three key components in ComCrawler, and each is compared with several baselines in the offline experiments. For entry point detection, we compare with: 1) **KeyWords**, a heuristic that matches the text contents of a clickable element to a set of predefined keywords such as *"comment", "discussion", "discuss", "reply"*. To cope with non-English websites, we translate the keywords to their target languages. 2) **TF-IDF**. We parse the HTML source codes of clickable elements into terms, compute the TF-IDF vector based on the top 25 most frequent terms, and train an SVM classifier.

For Web record extraction, we compare several representative unsupervised methods using different techniques, including 1) **PROSE** [31], a program synthesis API from Microsoft. The API [1] allows users to synthesize the Web record extraction program automatically [31] without any example or semi-automatically with a few examples [32]. We compare our method with the former use case. 2) **MiBAT** [21], a method

[1] www.microsoft.com/en-us/research/group/prose/

TABLE II: Performance of the individual components and the Ensemble.

module	method	F1	Acc.
Entry Point Discovery	Attribute-fastText (ours)	**0.97**	/
	TF-IDF	0.91	/
	KeyWords	0.35	/
Web Record Extraction	Web Record Invariants (ours)	/	**0.96**
	PROSE	/	0.82
	MiBAT	/	0.80
	DEPTA	/	0.36
	Pattern Signal	/	0.39
Comment Section Detection	Attribute-fastText (ours)	**0.96**	/
	TF-IDF	0.93	/
	Manual Features	0.81	/
	Date String Heuristic	0.76	/
End-to-End	/	0.95	/

that uses data fields with invariant structures as pivots, such as posting date of comments or price tag of products, to address the structure variation of Web records. When computing the similarity between two Web records, it only compares the sibling subtrees of the pivots. We choose date strings as the pivots. 3) **DEPTA** [17], a classic baseline that detects Web records by aligning similar subtrees. 4) **Pattern Signal** [20], a method that turns a Web page into a sequence and applies signal-processing techniques to detect regularities and patterns of Web records.

For comment section classification, we compare with: 1) **TF-IDF**, which has the exact implementation as the counterparts in the entry point detection baselines, but the TF-IDF vector size is increased to 100 because a comment section has substantially more text contents than a clickable element. 2) **Manual Features** [33], an SVM classifier trained on 14 Web page block features such as `<a>` tag ratio and date string ratio. 3) **Date String Heuristic** [21] selects comment sections based on the heuristic that every comment contains a date string. We implement a comment section filter based on this heuristic. To detect date strings, we use an open-source date parser library (https://dateparser.readthedocs.io).

C. Performance Analysis

1) Entry Point Detection: We evaluate our proposed classifier and the corresponding baselines on the ClkElmSet dataset with 10-fold cross-validation (except for the KeyWords method which does not require training). We group samples by their websites, so one website does not appear both in training and testing samples. We measure the performance of the component by the F1 score = $2 \cdot R \cdot P/(R+P)$, where R and P are recall and precision computed w.r.t the testing samples of clickable elements:

$$R = \frac{\text{\# True Positive}}{\text{\# True Positive + \# False Negative}} \quad (1)$$

$$P = \frac{\text{\# True Positive}}{\text{\# True Positive + \# False Positive}} \quad (2)$$

The F1 score of each model is given in the first block of Table II. Our Attribute-fastText method achieves the best F1

score of 0.97. The TF-IDF and the KeyWords methods score 0.91 and 0.35, respectively. The latter performs poorly because many comment-loading elements indicate their functionality by their shapes and positions and do not display any keywords to the user (the loading element at the top left of Figure 3 is an example of such a case).

2) Web Record Extraction: We test the Web record extraction methods on the **positive samples** from the CmtSecSet. A method may detect multiple groups of Web records, but we are only interested in the comment section. So for each page, we measure a method's performance by calculating its accuracy in detecting the comments of page p and ignoring other outputs that do not belong to the comment section:

$$acc_p = max(\frac{\text{\# correctly extracted comments in } s}{\text{\# ground truth comments in } p}), s \in S_p \tag{3}$$

where S_p is the set of detected Web record sections in the page. The accuracy is then averaged across all the pages. We observe that a method may segment a comment section into multiple sections (i.e., treat a comment section as multiple groups of records), which is generally not the desired behavior, and the formula gives credit only to the largest comment segment in this case. The second block of Table II shows the average accuracy. Both DEPTA and Pattern Signal methods have low performance: the former achieves 0.36 accuracy while the latter 0.39. The two methods are designed for records "with similar size and structure" (as emphasized by the authors of [20]) and are unable to cope with comments with complex content and nested structures. Their most frequent error is to split a comment section into multiple sections, some of which may include content that is not a comment. PROSE and MiBAT achieve 0.81 and 0.80% accuracy, respectively. Both methods tend to miss comments when the nesting structures are complex, and MiBAT also suffers from recall loss due to missing posting date strings. Our method locates comments by invariant subtrees and recovers comments in a bottom-up manner, and it does not measure the similarity among comments, thus it is more resilient to structure variations and nesting structures of comments, leading to an accuracy of 0.96.

3) Comment Section Detection: In the evaluation of detecting comment sections, we use the comment sections from the positive pages of CmtSecSet and apply our Web record extraction method to extract non-comment Web record sections from the negative pages. We test the Attribute-fastText method and the corresponding baselines on these record sections with 10-fold cross-validation (except for the Date String Heuristic method, which does not require training). We apply the same F1 score metric in the entry point discovery step, which is calculated based on the testing comment/non-comment sections.

As shown in the last block of Table II, the Attribute-fastText method achieves the best performance at 0.96. The TF-IDF method scores at 0.93 while the Manual Features method scores at 0.81. The good performance of the TF-IDF can be explained by the fact that the most frequent words used in the TF-IDF feature vector are terms from the HTML attribute values. This result provides additional evidence that supports our motivation to exploit HTML attributes. The Date String Heuristic method has the worst performance for two reasons: First, date string detection is a complex problem in itself. Many websites use vague date expressions in comments, such as "1d", which means 1 day ago. What makes it even more complex is the different date formats across languages. We notice that the date parser missed most of the correct Korean posting date strings, such as "오후 1시 반". The second issue is precision – non-comment records may also contain date strings, e.g., a list of published articles at some news outlet gives the dates when the articles were published.

4) End-to-End: The performance of ComCrawler is the end-to-end performance of the three components in sequence. We first evaluate ComCrawler offline using the ClkElmSet and CmtSecSet to assess its performance in ideal settings, which serves as the upper bound. We simulate Web response by randomly linking each positive/negative sample in the testing holdout of the ClkElmSet to a positive/negative sample in the testing holdout of the CmtSecSet. If a clickable element is classified as a positive, we extract Web records in the linked page and input them into the comment section classifier.

When measuring the end-to-end performance, we are interested in the final output. Thus we compare the detected comments against the ground truth comments for each testing page p and calculate $F1_p = 2 \cdot R_p \cdot P_p/(R_p + P_p)$, where

$$R_p = \frac{\text{\# correctly detected comments in } p}{\text{\# ground truth comments in } p} \tag{4}$$

$$P_p = \frac{\text{\# correctly detected comments in } p}{\text{\# detected comments in } p} \tag{5}$$

Note that when dividing by zero, we set R_p to 1 if there is no false positive and 0 otherwise, and we set P_p to 1 if there is no false negative and 0 otherwise. there is no detected comment and 0 otherwise, and we set P_p to 1 if there is no ground truth comment and 0 otherwise. In other words, R_p and P_p are set to 1 if a page does not have comments and the pipeline does not produce false positives. $F1_p$ is then averaged across all the testing pages. Our approach yields a 0.95 F1 score, with almost perfect precision, but 0.92 recall due to the loss accumulated through the entire process.

D. Politeness.

Politeness is measured in the number of clickable elements we need to trigger: the fewer, the better. We investigate the reduction of unnecessary clicks by the comment entry detection step using the 1,500 source Web pages of the ClkElmSet dataset. Figure 9 shows the PDF (in log scale) of the total number of clickable elements per page and the number of entry point candidates per page. The distribution of the number of clickable elements is heavily tailed, with an average of 261 clickable elements per page. After applying the clickable element classifier, the number of clicks is significantly reduced: the crawler explores only 2% of the clickable elements on average per page. Our classifier recommends 5.56 entry point candidates on average.

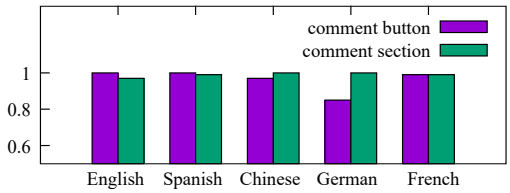


Fig. 10: F1 scores of the 5 most frequent languages held out in the training process.

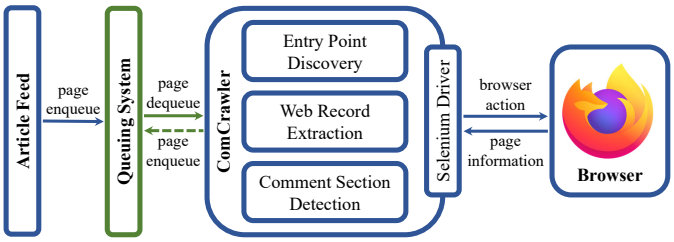


Fig. 11: System implementation.

E. Internationalization.

The clickable element and comment section classifiers are both trained on the 3-gram features extracted from HTML attribute values. We assume that the classifiers are oblivious to the language of a website because website developers generally use English words in the source code. To test the assumption, we perform a hold-out test. We present here the results for the 5 most frequent languages in our dataset. We hold websites from one language for testing and use the remaining websites for training. This corresponds to zero-shot learning, which represents a more practical setup.

Figure 10 shows the F1 score of the two classifiers on the 5 languages. The performance is nearly ideal for all the languages, except for German Web pages where the F1 score is 0.85. This is because some of the comment-loading elements from those websites contain keywords, like "Kommentare," which never appear in the samples from other languages.

VI. Online: From Theory to Practice

In theory, theory and practice are the same. In practice, they are not.– Albert Einstein

In this section, we describe the implementation of ComCrawler and the challenges we met when deploying the crawler online. We analyze these challenges and present the extra steps we take to address these challenges in order to make our ComCrawler practical.

VII. Implementation

We implement ComCrawler as an end-to-end system that takes an article page URL as input and outputs extracted comments, if detected. Figure 11 shows the architecture of our system, which consists of an article feed that provides article page URLs, a page queuing system that schedules the visiting timings of the target pages (to be discussed in the following section), and a comment crawler that contains the three core modules of ComCrawler for comment discovery and extraction. The module uses a Selenium WebDriver[2] to control a Firefox browser so we can automatically load, render, and interact with web pages.

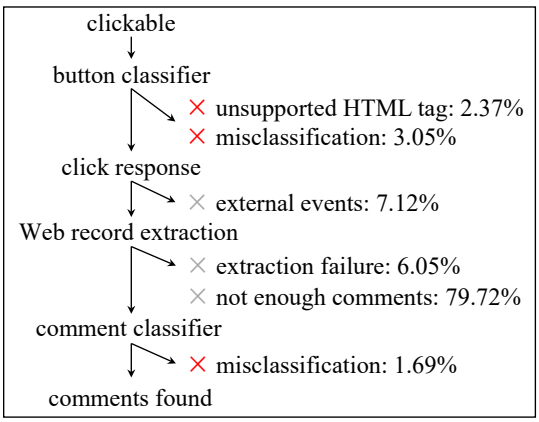


Fig. 12: Misses break down.

TABLE III: Online and offline performance of our approach.

experiment setup			F1	Rcall
offline	test on datasets		0.95	0.92
online	w/o queue & immediate visit		0.41	0.26
	w/ queue	1 visit	0.85	0.77
		2 visits	0.89	0.84
		3 visits	0.90	0.85

In our deployment, we implemented the article feed using the Google News RSS feed. We choose Google News because it aggregates news articles from a vast number of websites, and it allows us to switch across different languages and countries/regions so we can test our system comprehensively. News websites are a good representative of the complex issues encountered during comment crawling: the websites have different layouts, and adopt different commenting technologies (from custom solutions to third-party plug-ins), and represent a broad range of languages.

A. Real World Challenges

In our initial attempt for online evaluation, we ran our system on 1,640 (20 pages per country/region) unseen Web pages. It fetched one page from the feeds at a time and visited the page immediately. The crawler encountered Web pages from 82 countries/regions and 37 languages. We manually checked its output. As shown in Table III, our approach achieves a low 0.41 F1 score in detecting comments on the first attempt. Precision remains comparable to that in the offline evaluation, but recall degrades to 0.26.

To understand the misses, we investigated the entire pipeline of ComCrawler. Our prototype includes a module that saves screenshots of a page before and after each click of an entry point candidate and screenshots of the candidate comment sections. We inspected the screenshots to judge the issues. We give the detailed breakdown of misses in Figure 12, which depicts the flow of the crawler and indicates the points of

[2]www.selenium.dev

TABLE IV: Arrival time for the 1st and 10th comment after article publication.

Outlet	PUB–1st	1st–10th	PUB–10th
Daily Mail	1.48(3.02)	0.21(0.41)	1.69(3.07)
Fox News	1.94(3.16)	0.55(9.13)	2.49(9.69)
Washington Post	6.48(5.84)	0.58(3.20)	7.07(6.64)
The Guardian	0.87(1.49)	0.09(0.82)	0.96(1.71)
New York Times	2.88(5.60)	0.26(0.76)	3.14(5.71)
Average (20 outlets)	3.41(45.37)	0.42(3.75)	3.84(45.54)

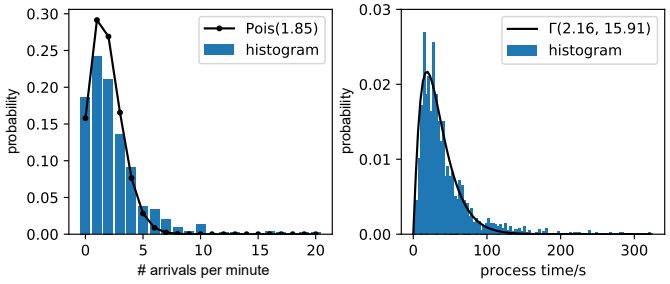


Fig. 13: Distribution of the arrival rate and service time.

failure. At each step in the figure, we indicate a miss due to the failure of our modules with a red cross and a miss due to external factors, such as an accept cookie pop-up message, with a grey cross. We give the reasons along with their portion on misses at each step.

We found that the performance discrepancy between the offline and online experiments is primarily due to the misses of comment sections, accounting for 85.77% of the misses. The crawler always visits incoming pages immediately, but very often, the page has not received a sufficient number of comments yet ($F_{th} = 10$). Other failures include external events that prevent comment sections from loading, such as login pages, blocking ad popups, and page loading failure. These factors suggest that the system should visit a page with proper timing and have a revisit strategy.

B. Page Queuing System

To address the visit timing problem, one may try to visit a page after its commenting time window is closed or delay the visit by a certain amount of time so the page can accumulate as many comments as possible. Such heuristic strategies, while possible, are not principled and are difficult to analyze to achieve the desired behavior. Instead, we add a page queuing system to ComCrawler, and we want to design the queuing system with a proper waiting time to accumulate enough comments before visiting.

In queuing theory, Kendall's notation is commonly used to classify queuing systems, which has the format $A/S/c/D$ [34], where A denotes the arrival process, S denotes the distribution of service time, c denotes the number of servers, and D is the queuing discipline. We discuss each of the attributes of our system below.

Arrival process. The Poisson distribution is widely adopted for the arriving process where arrivals occur with a known constant mean rate and independent of the time since the last event [35]. We need to validate this assumption on our source of web pages, i.e., Google News. We monitor the article arrival rate (per minute) at Google News in U.S. over a week and determine that it follows a Poisson distribution with $\lambda_0 = 1.85$. Figure 13 (left) shows the normalized histogram for the number of news arrivals per minute using the one-week sample. As shown by the solid line in the figure, the estimated probability mass function is well-fitted using the Poisson distribution.

Service process. To gain an understanding of the service process, we collect the running times of our system over the 1,640 pages used in our first attempt of online deployment reported in Section VII-A. We find that the normalized histogram of the process times is best fitted with the PDF of the Gamma distribution with $k = 2.16$ and $\theta = 15.91$. The result is shown in Figure 13 (right). Hence, the expected service time $E[S] = k\theta = 34.37s$ and the variance $\text{VAR}[S] = k\theta^2 = 546.75$. We point out that roughly 75% of the service times is waiting for the result pages to be loaded. One can ameliorate this issue by preemptively loading a number of pages. We do not address this aspect in this paper.

Number of Servers. In our experiment, we set up only one instance of crawler so the number of servers is 1. However, our system may be scaled up horizontally by running multiple crawler instances that serve the same page queuing system. We implement our queuing system with a relational database, and the crawler requests a page from the queue by issuing an SQL query to fetch and lock one row at a time from the queuing table.

Queuing Discipline: Our queuing system implements a basic first-in-first-out queuing discipline. On top of that, we enforce two policies to address the challenge of early visits: page dropping and revisiting.

Since the average processing speed of the server (crawler) is slightly slower than the arrival rate of news pages (60/34.37=1.75 vs 1.85 pages/min), our queuing system will be unstable (i.e., queuing length unbounded) if we put every page from the news feed into the queue. Thus we drop an incoming page if the current expected queuing time T_Q is greater than the expected waiting time T_W to accumulate enough comments ($>= 10$) to be extracted by our Web record extraction module. While T_Q can be easily derived from the queue length L_Q and service time E[S], specifically, $E[T_Q] = E[S] * L_Q$, T_W is hard to predict, and we can only have a good guess by investigating user commenting behaviors.

We study "user reaction", which we define as the time difference between the first comment posted for an article and the article's publication time, at 20 representative news outlets, and we present the results of 5 major outlets in Table IV. We observe that there is generally a user reaction time between an article's publication time and the arrival time of its first comment, which varies from outlet to outlet. For example, at the Washington Post, the average user reaction is 6.48 hours with a standard deviation of 5.84 hours, but the Guardian's

average user reaction time is 0.87 hours with a standard deviation of 1.49 hours. In addition, we find that it takes 0.42 hours on average for an article to accumulate 10 comments once the first comment is received. Thus, we set T_W to the average user reaction time plus the arrival time for the first 10 comments observed across the studied outlet, which is 3.84 hours.

However, we observe that the comment arrival time has a significant variance – 45.54 for the arrival time of the first 10 comments – so we may still visit too early for some pages by waiting for the average arrival time. Thus we will schedule a revisit by putting the target page at the back of the queue if we do not detect comments. Empirically, we discard a page on the third visit because we do not observe any significant recall improvement going beyond that (see Table III).

C. Online Evaluation

With the queuing system, we left the crawler running for 10 days, and the crawler visited more than 25k pages from 4,739 websites. We partition the pages into 8 categories from Google News: world, nation, business, technology, entertainment, sports, health. We performed a stratified sampling of the crawled pages by taking 100 pages per day, distributed proportionally per category. We manually investigated the data. With the queuing system, the framework achieves a 0.85 F1 score on the first visit, 0.89 on the second visit, and 0.90 on the third visit (the last 3 rows in Table III). We do not see any significant performance change after the third visit.

To verify the crawler's performance in the wild (i.e., not just news websites), we tested it on Web pages returned by the general Google search. We used the trending queries of 2023 in the U.S. We sampled 500 positive pages and 500 negative pages and checked the results manually. Our crawler achieved a 0.87 F1 score, with a precision of 0.82 and a recall of 0.92.

VIII. Related Work

While there is a rich literature on data mining leveraging *user-generated content* (UGC), such as comments and reviews, limited work discusses the process of crawling the data sources. Most of the works devoted to this topic focus on extracting UGC on specific platforms or specific types of Web pages such as Web fora and blogs. We discuss some typical research works below.

Popular online social networks, such as Facebook and Twitter, have received intense interest for effectively and efficiently crawling their user posts. Many crawling tools leverage the official APIs offered by these social networks. For example, [36], [37] give Facebook crawlers based on the Facebook Graph API and [38]–[40] crawl Twitter using its REST API. While this type of crawler has excellent performance, they focus on specific social platforms and cannot be applied to other social media platforms.

Since the goal is to get UGC from Web pages that support commenting, research efforts in this direction treat this problem as an HTML content extraction problem. They tend to target specific types of Web pages such as blogs and Web fora. [33], [41], [42] introduce blog comment extraction strategies based on the comments' HTML structure and visual appearance; [33], [43] use the text feature such as HTML tags and comment keywords to train comment/non-comment classifiers, which is similar to our approach in regard to finding the comment section. Apart from the structure and text features, [42] tries to understand the structure and layout of a blog page by utilizing the Functional Semantic Units (FSUs) that help users understand a Web page. It uses FSUs to build a frequent-based mining approach for extracting comment areas.

Forum threads can also be treated as user comments. [44] proposes a two-step crawling solution to collect forum thread pages, where the first step is an inter-site crawler that locates forum sites on the Web and the second is an intra-site crawler that finds thread pages by learning the context of links that lead to thread pages. [45] treats the thread crawling problem as a URL-type recognition problem. [21], [46] focus on identifying thread comments by detecting comments based on posting structure, and the latter also uses certain domain constraints (e.g., post-date) to design better similarity function to circumvent the influence of free-format comment contents.

The works most related to ours make two key assumptions that our work addresses: (1) assume that user postings are loaded with the web page (i.e., static case) and (2) assume that an oracle gives the comment section. In the modern Web (1) is easily violated in practice as more and more websites load comments dynamically to relieve server resources and (2) is a strong assumption – we show that it is not easy to identify the comment section in general, particularly, when a page has few comments. Our contribution in this paper is that we address (1) and (2); to our knowledge, no other work addresses these problems.

IX. Conclusion

We introduced the problem of detecting dynamically loaded user comment sections on the Web. We identified the typical ways comments are loaded on a Web page and described a framework, ComCrawler, to find comments across different websites independent of country/region and language. ComCrawler achieved a high F1 score of 0.95 when tested offline. We deployed ComCrawler online, and we noticed that its performance dropped drastically to a 0.41 F1 score. We analyzed the reasons for the failures and identified that many of them could be addressed if a page was properly queued and revisited. This observation was drawn by analyzing user commenting behavior at a large number of websites. We used that analysis to determine the parameters of the underlying queuing system of the crawler. The new system achieved an F1 score of 0.90. We contend that our tool is useful to a broad range of practitioners who need to mine/analyze user-generated content from many websites and need to transparently access such data from websites in different languages and application domains. Our future work is to learn and maintain wrappers to comment-loading elements and comment sections.

REFERENCES

[1] P. Weber, "Discussions in the comments section: Factors influencing participation and interactivity in online newspapers' reader comments," *New Media & Society*, vol. 16, no. 6, pp. 941–957, 2014.

[2] A. Mullick, S. Ghosh, R. Dutt, A. Ghosh, and A. Chakraborty, "Public sphere 2.0: Targeted commenting in online news media," in *ECIR*. Springer, 2019, pp. 180–187.

[3] F. Toepfl and E. Piwoni, "Public spheres in interaction: Comment sections of news websites as counterpublic spaces," *Journal of Communication*, vol. 65, p. 465–488, 2015.

[4] M. Ziegele and O. Quiring, "Conceptualizing online discussion value: A multidimensional framework for analyzing user comments on mass-media websites," *Annals of the Int. Comm. Assoc.*, vol. 37, no. 1, pp. 125–153, 2013.

[5] J. Ye and S. Skiena, *MediaRank: Computational Ranking of Online News Sources*, 2019, p. 2469–2477.

[6] L. He, C. Han, A. Mukherjee, Z. Obradovic, and E. Dragut, "On the dynamics of user engagement in news comment media," *WIRDMKD*, vol. 10, no. 1, 2020.

[7] H. Liu, Z. Zhang, P. Cui, and Y. e. a. Zhang, "Signed graph neural network with latent groups," in *KDD*, 2021, pp. 1066–1075.

[8] K. Shu, L. Cui, S. Wang, D. Lee, and H. Liu, "defend: Explainable fake news detection," in *KDD*, 2019, pp. 395–405.

[9] R. Zafarani, X. Zhou, K. Shu, and H. Liu, "Fake news research: Theories, detection strategies, and open problems," in *KDD*, 2019, pp. 3207–3208.

[10] A. Wang, R. Ying, P. Li, N. Rao, K. Subbian, and J. Leskovec, "Bipartite dynamic representations for abuse detection," in *KDD*, 2021, pp. 3638–3648.

[11] K. Gupta, M. Joshi, A. Chatterjee, S. Damani, K. N. Narahari, and P. Agrawal, "Insights from building an open-ended conversational agent," in *Proceedings of the First Workshop on NLP for Conversational AI*. Florence, Italy: Association for Computational Linguistics, Aug. 2019, pp. 106–112. [Online]. Available: https://aclanthology.org/W19-4112

[12] A. Halevy and J. Dwivedi-Yu, "Learnings from data integration for augmented language models," *arXiv preprint arXiv:2304.04576*, 2023.

[13] N. Kushmerick, *Wrapper induction for information extraction*. University of Washington, 1997.

[14] H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. Yu, "Fully automatic wrapper generation for search engines," in *WWW*, 2005, pp. 66–75.

[15] K. Lerman, S. N. Minton, and C. A. Knoblock, "Wrapper maintenance: A machine learning approach," *JAIR*, vol. 18, pp. 149–181, 2003.

[16] S. Ortona, G. Orsi, M. Buoncristiano, and T. Furche, "Wadar: Joint wrapper and data repair," *VLDB*, vol. 8, no. 12, pp. 1996–1999, 2015.

[17] Y. Zhai and B. Liu, "Web data extraction based on partial tree alignment," in *WWW*, 2005, pp. 76–85.

[18] W. Liu, X. Meng, and W. Meng, "Vision-based web data records extraction," in *Proc. 9th international workshop on the web and databases*, 2006, pp. 20–25.

[19] Y. Fang, X. Xie, X. Zhang, R. Cheng, and Z. Zhang, "Stem: a suffix tree-based method for web data records extraction," *KIS*, vol. 55, no. 2, pp. 305–331, 2018.

[20] R. P. Velloso and C. F. Dorneles, "Extracting records from the web using a signal processing approach," in *CIKM*, 2017, pp. 197–206.

[21] X. Song, J. Liu, Y. Cao, C.-Y. Lin, and H.-W. Hon, "Automatic extraction of web data records containing user-generated content," in *CIKM*, 2010, pp. 39–48.

[22] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou, "Ajax crawl: Making ajax applications searchable," in *ICDE*, 2009, pp. 78–89.

[23] C. Castillo, "Effective web crawling," in *ACM SIGIR Forum*, vol. 39, no. 1, 2005, pp. 55–56.

[24] J. Fürnkranz, "A study using n-gram features for text categorization," *OFAI*, vol. 3, pp. 1–10, 1998.

[25] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," in *EACL*, April 2017, pp. 427–431.

[26] J. Mandl, "Why are all programming languages in english?" shorturl.at/crxB7, 2016, accessed: 2021-10-22.

[27] Z. Chen, W. Meng, and E. Dragut, "Web record extraction with invariants," *Proceedings of the VLDB Endowment*, vol. 16, no. 4, pp. 959–972, 2022.

[28] G. Miao, J. Tatemura, W.-P. Hsiung, A. Sawires, and L. E. Moser, "Extracting data records from the web using tag path clustering," in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 981–990.

[29] R. P. Velloso and C. F. Dorneles, "Extracting records from the web using a signal processing approach," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 197–206.

[30] M. J. Zaki and W. Meira Jr, *Data Mining and Machine Learning: Fundamental Concepts and Algorithms*. Cambridge University Press, 2020.

[31] M. Raza and S. Gulwani, "Automated data extraction using predictive program synthesis," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.

[32] ——, "Web data extraction using hybrid program synthesis: A combination of top-down and bottom-up inference," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1967–1978.

[33] H.-A. Kao and H.-H. Chen, "Comment extraction from blog posts and its applications to opinion mining." in *LREC*, 2010, pp. 1113–1120.

[34] A. O. Allen, *Probability, statistics, and queueing theory*. Academic press, 2014.

[35] S. Katti and A. V. Rao, "Handbook of the poisson distribution," 1968.

[36] F. Erlandsson, R. Nia, M. Boldt, H. Johnson, and S. F. Wu, "Crawling online social networks," in *ENIC*, 2015, pp. 9–16.

[37] C. Aliprandi and A. E. e. a. De Luca, "Caper: Crawling and analysing facebook for intelligence purposes," in *ASONAM*, 2014, pp. 665–669.

[38] G. Dziczkowski, L. Bougueroua, and K. Wegrzyn-Wolska, "Social network-an autonomous system designed for radio recommendation," in *CASoN*, 2009, pp. 57–64.

[39] M. Bošnjak, E. Oliveira, J. Martins, E. Mendes Rodrigues, and L. Sarmento, "Twitterecho: A distributed focused crawler to support open research with twitter data," in *WWW*, 2012, p. 1233–1240.

[40] P. Noordhuis, M. Heijkoop, and A. Lazovik, "Mining twitter in the cloud: A case study," in *CLOUD*, 2010, pp. 107–114.

[41] D. Cao, X. Liao, H. Xu, and S. Bai, "Blog post and comment extraction using information quantity of web format," in *AIRS*, 2008, p. 298–309.

[42] F. Chun-Long and M. Hui, "Extraction technology of blog comments based on functional semantic units," in *CSAE*, 2012, pp. 422–426.

[43] M. Neunerdt, M. Niermann, R. Mathar, and B. Trevisan, "Focused crawling for building web comment corpora," in *CCNC*, 2013, pp. 685–688.

[44] L. Barbosa, "Harvesting forum pages from seed sites," in *ICWE*, 2017, pp. 457–468.

[45] J. Jiang, X. Song, N. Yu, and C.-Y. Lin, "Focus: learning to crawl web forums," *TKDE*, no. 6, pp. 1293–1306, 2012.

[46] M. Bank and M. Mattes, "Automatic user comment detection in flat internet fora," in *DEXA*, 2009, pp. 373–377.