Industrial & Applications Paper

open proceedings

ComCrawler: General Crawling Solution for Article Comments

Zhijia Chen
Meta
Menlo Park, CA, USA
zhijia@meta.com

Weiyi Meng
Binghamton University
Binghamton, NY, USA
meng@binghamton.edu

Eduard Dragut
Temple University
Philadelphia, PA, USA
edragut@temple.edu

ABSTRACT

Commenting is a prominent feature to enhance user engagement on websites. User comments have powered many applications, such as opinion mining, fake news detection, and LLM training. However, comments are difficult to collect at scale from diverse websites, as they are often hidden until triggered by specific user interaction, such as clicking on a designated page element. Moreover, comments may contain rich formatted contents and nesting structures, making them difficult to detect and extract from the target Web page. This study presents ComCrawler, an end-to-end comment crawling solution that leverages neural network models and Web record extraction algorithms to automate the process of triggering, extracting, and classifying comments. The system achieves a high end-to-end comment detection F1 score of 0.95 in offline evaluations, with ideal conditions. However, the system's performance degrades significantly in real-world online tests, prompting further exploration in deployment. To address practical challenges like comment scarcity during visits, a principled re-visiting framework based on queuing theory is proposed, which helps ComCrawler to achieve a high 0.90 F1 score in practice.

1 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 [46]. 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 [34, 42, 54]. Take news websites as an example, there are over 50K news websites in the world [48], and a large fraction of them have over 100K subscribers who actively comment [23, 24]; 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 [30, 47], fake news detection [3, 39, 49], user engagement and behavior analysis [4, 41, 45, 53], which attract relentless attention from industry and academia alike. Such data is also used to generate conversational systems [21] and to enrich LLMs [22].

The problem we aim to solve in this work is: Given a (random) Web page, determine if the page hosts a commenting section and, if it does, locate it 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. The issue may be alleviated by applying wrapper induction/program synthesis techniques to infer wrapper programs using labeled examples from the target websites [28, 52]. 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 [29, 37]. A more scalable solution is to infer the Web API of the comment system and request comments directly from the comment system's server [10], which is much more efficient than composing/generating wrapper rules for each website. However, such a solution has limited coverage as it is only applicable when the target websites adopt commenting systems in the knowledge base and the Web API is correctly inferred. Thus, a more general solution is valuable for broader comment crawling.

The problem is further complicated by the fact that user comments tend to be dynamically loaded on the modern Web by specific triggering events, and websites employ various means to trigger and display user comments. 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 button, but the former generates a modal popup (Figure 1b), whereas the latter presents the comments in a new 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 present ComCrawler, a general comment crawling framework for dynamically loaded comments. As illustrated in Figure 2, ComCrawler consists of the following key steps:

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 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. Thus at this step, we aim to detect and trigger the comment loading event in a polite manner.

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. 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 [14, 31]. 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 [13, 15, 19, 43] and that Web

© 2025 Copyright held by the owner/author(s). Published in Proceedings of the 28th International Conference on Extending Database Technology (EDBT), 25th March-28th March, 2025, ISBN 978-3-89318-099-8 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

Series ISSN: 2367-2005

10.48786/edbt.2025.85

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

Figure 1: Typical ways of loading comments dynamically.

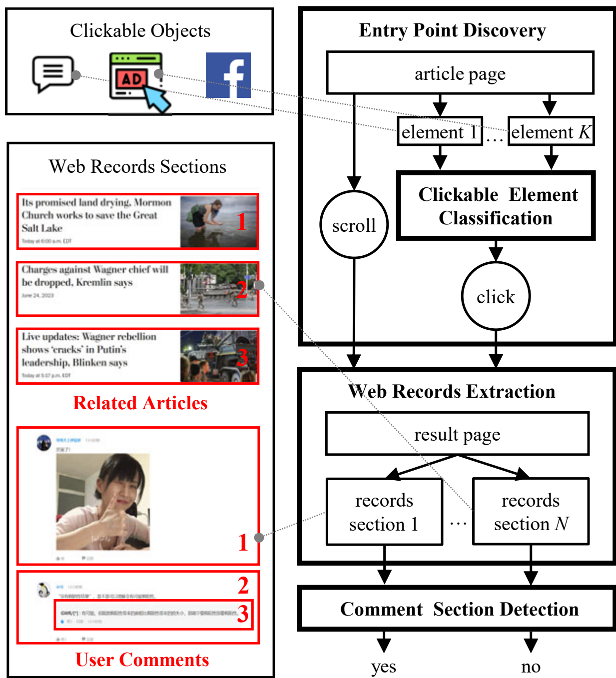


Figure 2: Comment crawling workflow.

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). We propose a new Web record extraction algorithm that exploits the common structural components among Web records to address this unique challenge of comments extraction.

Comment Section Detection. While the Web record extraction method addresses the unique challenges of comments, it is likely to extract many other Web records on the target page as well, such as the list of related articles or ads; thus the last step of ComCrawler is comment section detection which aims to detect the comment section from the Web record extraction output.

The above steps try to solve the problem in an ideal environment, where triggering the right entry point always loads the comments, and the target page have received enough number of comments so that the common structural signal is detectable. However, these are hard to guarantee in practice. We found that the page loading may failed due to many external factors such as network failure; and as comments are accumulated over time, visiting too early may cause the framework to give up on a pre-maturely. We address these issues with principled (re)visiting protocol and endow ComCrawler with a queuing system whose policy is informed by the observed comments accumulation statistics. This helps ComCrawler achieve an F1 score of 0.90 in deployment, which is only slightly off from the 0.95 F1 score that we observed in theoretical offline evaluation.

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 system, 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.

2 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, [1, 18] give Facebook crawlers based on the Facebook Graph API[1] and [7, 17, 36] 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.

[1] https://developers.facebook.com/docs/graph-api

[8, 12, 27] introduce blog comment extraction strategies based on the comments' HTML structure and visual appearance; [27, 35] 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, [12] 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 and then extract comments from the comment area, which is identified by the frequent presence of FSUs in a comment.

Forum threads can also be treated as user comments. [6] 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. [25] treats the thread crawling problem as a URL-type recognition problem [2]. [5, 40] 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.

3 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 which are not visible to the user, but together they dictate how the browser formats and displays the content to the user.

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 [16, 19, 40, 51]. 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.

Clickable Element. For a crawler to automatically locate the HTML element that loads the comments, the first step is to find all the clickable elements in the DOM tree. While modern Web browsers allow any element in a DOM tree to be clickable by adding an event listener, the `<button>` tag and `<a>` tag are two standard tags that expect click events. For the sake of simplicity, we will only consider the `<a>` and `<button>` tag as clickable elements in this work.

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. On 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.

4 COMMENT DISCOVERY AND DETECTION

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

4.1 Entry Point Discovery

One critical policy for effective Web crawling is politeness [9], that is, avoiding overloading target websites with unnecessary requests. So instead of traversing every `<button>` and `<a>` elements on a page, we build a clickable element classifier to reduce the unfruitful clicks. We observe that the text content and the attribute values of a clickable element are likely to present strong hints about its purpose. 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 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 [20] 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[2] 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 [26]. 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 neural network 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.

[2]https://fasttext.cc

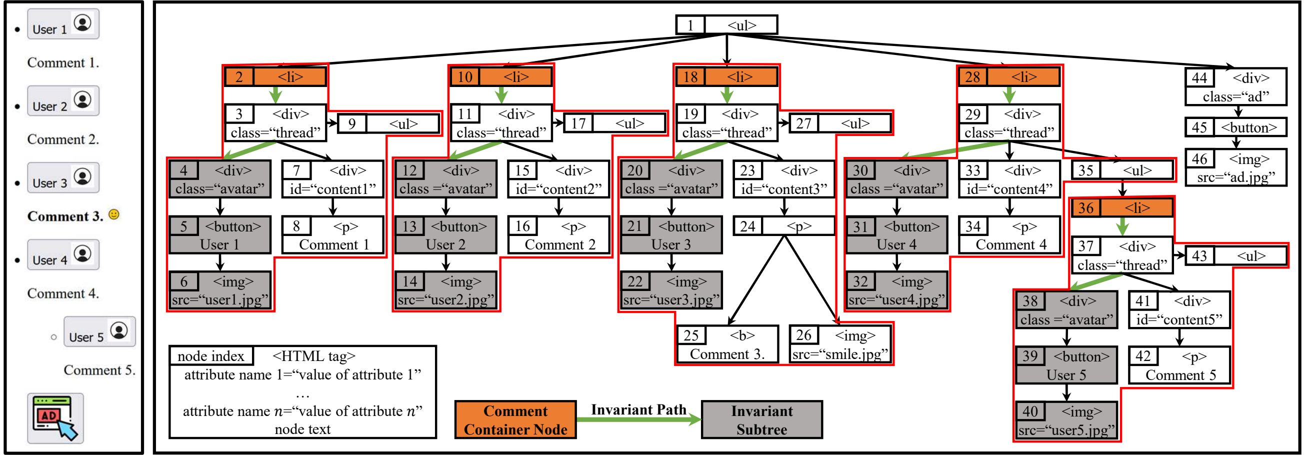


Figure 3: Examples of comment DOM subtrees.

4.2 Comment Section Detection

The comment section detection module takes the HTML source codes of a Web record section as input, which is provided by the Web records extraction component (Section 5), and it predicts if the Web record section is a comment section or not. In the example of Web record sections shown in Figure 2, 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 detection module 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 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 detection module, and potentially image features used by existing Web record extraction works [40, 43, 51] – 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.

Similar to the comment button classifier, we build the comment detection module 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 [32]), 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 dection module.

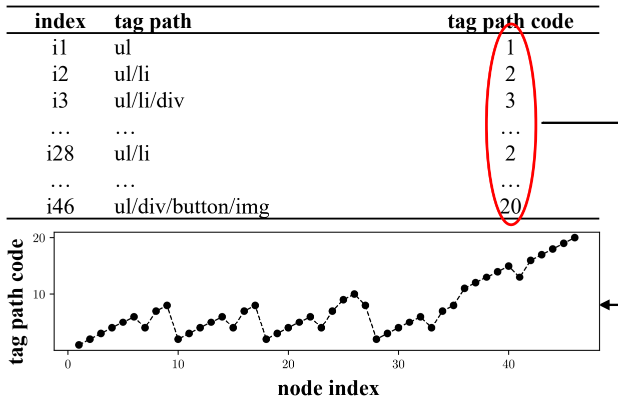
5 WEB RECORD EXTRACTION

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 3, 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 [40, 51] (Section 6.2). 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.

We identify that the main cause of errors is that these methods generally do not consider the nesting behavior of Web records, and the rich formatted contents of comments easily introduce too much structural noise that break down the algorithms. We



index	tag path	tag path code
i1	ul	1
i2	ul/li	2
i3	ul/li/div	3
…	…	…
i28	ul/li	2
…	…	…
i46	ul/div/button/img	20

Figure 4: Tag path code sequence of $\mathbb{E}$ (Figure 3).

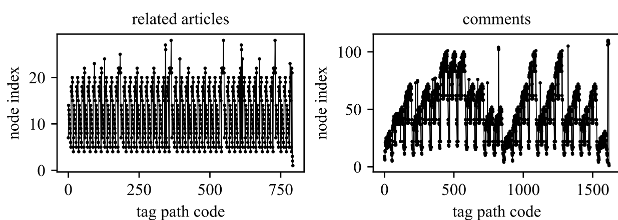


Figure 5: Tag path code sequence of related articles (left) and user comments (right) on a Web page.

observed that we can no longer enforce record level structural similarity among comments, however, 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 method [11].

5.1 Signal of Web Records

Inspired by existing works that perceive Web records as repeating tag sequences [19, 33, 44], 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 4 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 4. Details of the process can be found in [44].

We compare the sequential signals of regular Web records against those from comments. Figure 5 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.

5.2 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 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 3), 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 first align the invariant subtrees on a DOM tree, and then we find their corresponding record container nodes by detecting the potential invariant path.

5.3 Mining Frequent Patterns

We detect invariant subtrees based on frequent pattern mining from the DOM tree. Notice that in Figure 5 (right), the tag path 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) due to the nesting behavior, making it difficult to extract the patterns from the signal.

Intuitively, if we can represent a node in such a way that nodes with the same subtree structure are represented by the same value, then we can expect the invariant subtrees to form the same code sequence. 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:

```
⟨div, [class], [⟨button, [], [⟨img, [src], []⟩]⟩]⟩
```

Note that $x.attrib$ and $x.children$ are arrays, and we use `[·]` 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.

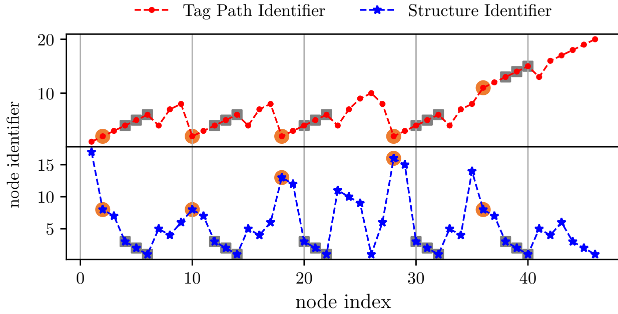


Figure 6: Transforming the running example (Figure 3) into a signal by representing a node using an HTML tag path and subtree structure, respectively. The record container nodes are marked in orange circles, and the nodes of invariant subtrees are marked by gray squares.

Figure 6 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 [50].

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.

5.4 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 3). 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 4.2 will detect sections that contain comment records.

6 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.

6.1 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 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**.

6.2 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)

Table 1: 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
	DEPTA	/	0.36
	Pattern Signal	/	0.39
Comment Section Detection	Attribute-fastText (ours)	**0.96**	/
	TF-IDF	0.93	/
	Manual Features	0.81	/
End-to-End	/	0.95	/

PROSE [38], a program synthesis API from Microsoft [3] that allows users to synthesize the Web record extraction program automatically. 2) **DEPTA** [51], a classic baseline that detects Web records by aligning similar subtrees. 4) **Pattern Signal** [43], a method that turns a Web page into a sequence and applies signal-processing techniques to detect Web records based on their signal pattern and frequencies.

For comment section classification, we compare with: 1) **TF-IDF**, which has the same implementation as the counterparts in the entry point detection baselines, 2) **Manual Features** [27], an SVM classifier trained on 14 Web page block features such as <a> tag ratio and date string ratio.

6.3 Performance Analysis

6.3.1 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, which is given in the first block of Table 1. 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 2 is an example of such a case).

6.3.2 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:

$$acc_p = max(\frac{\text{\# correctly extracted comments in } s}{\text{\# ground truth comments in } p}), s \in S_p \quad (1)$$

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 1 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 [43]) 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 achieves 0.81% accuracy, and tend to miss comments when the nesting structures are complex. Our method locates comments by invariant subtrees and recovers comments in a bottom-up manner, which is more resilient to structure variations and nesting structures of comments, leading to an accuracy of 0.96.

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

6.3.3 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 1, 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.

6.3.4 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. 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 dection module.

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

$$R_p = \frac{\text{\# correctly detected comments in } p}{\text{\# ground truth comments in } p} \quad (2)$$

$$P_p = \frac{\text{\# correctly detected comments in } p}{\text{\# detected comments in } p} \quad (3)$$

And $F1 = avg(F1_p)$. 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.

7 IMPLEMENTATION AND DEPLOYMENT

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.

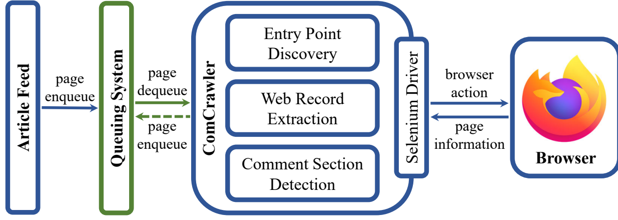


Figure 7: System implementation of ComCrawler.

7.1 Real World Challenges

In our initial attempt for deployment, we connected ComCrawler with the Google News Feed, fetching one page from the latest feeds at a time. However, when we manually compare the output comments with the target pages, we got a very disappointing results of 0.41 F1 score: the precision remains comparable to what we saw in the offline evaluation, but the recall degraded to 0.26. We investigate the input and output of each step, and found that the performance discrepancy 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 is just published and has not received a sufficient number of comments yet. 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.

7.2 Page Queuing System

To address the visit timing problem, one may try to delay the visit by a long time so the page can accumulate as many comments as possible. Such heuristic strategies, while possible, are not principled and difficult to analyze to achieve the desired behavior. Instead, we add a page queuing system to ComCrawler, and we aim to design the queuing system with a proper expected waiting time T_W to accumulate enough comments before visiting.

To determine T_W, we studied the user commenting behavior across difference websites in another work [23], and we found that it takes 3.41 hours on average for an article to accumulate 10 comments (the threshold for our Web record extraction module). Thus, we control page arrival rate of the queuing system to set $E[T_W] = 3.41$ hours. Furthermore, considering the variance of the comment arrival time, we may still visit too early for some pages by waiting for $E[T_W]$. Thus we will schedule a revisit by putting the target page at the back of the queue if we do not detect comments but at least one comment loading element is detected. Empirically, we discard a page on the third visit because we do not observe any significant recall improvement going beyond that (see Table 2).

7.3 Implementation

Figure 7 shows the implementation of ComCrawler , 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[4] to control a Firefox browser so we can automatically load, render, and interact with web pages.

[4]www.selenium.dev

Table 2: Online and offline performance of ComCrawler .

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.

7.4 Online Evaluation

We deployed ComCrawler and validated its performance in a 10-days time window. The crawler visited more than 25k pages from 4,739 websites. We partition the pages into 8 categories according to Google News and 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 2). We do not see any significant performance change after the third visit.

8 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.

Acknowledgment

This work was supported in part by the U.S. National Science Foundation 1546480, 1546441, 1838145 and 2137846 grants.

REFERENCES

[1] Carlo Aliprandi and Antonio E et al. De Luca. 2014. Caper: Crawling and analysing Facebook for intelligence purposes. In *ASONAM*. 665–669.
[2] Abdullah Aljebreen, Weiyi Meng, and Eduard Dragut. 2021. Segmentation of Tweets with URLs and its Applications to Sentiment Analysis. *AAAI* 35, 14 (May 2021), 12480–12488.
[3] Abdullah Aljebreen, Weiyi Meng, and Eduard C. Dragut. 2024. Analysis and Detection of "Pink Slime" Websites in Social Media Posts. In *the WebConference*. 2572–2581.
[4] Jumanah Alshehri, Marija Stanojevic, Eduard Dragut, and Zoran Obradovic. 2021. Stay on Topic, Please: Aligning User Comments to the Content of a

News Article. In *Advances in Information Retrieval*. Springer International Publishing, 3–17.

[5] M. Bank and M. Mattes. 2009. Automatic User Comment Detection in Flat Internet Fora. In *DEXA*. 373–377.

[6] Luciano Barbosa. 2017. Harvesting forum pages from seed sites. In *ICWE*. 457–468.

[7] Matko Bošnjak, Eduardo Oliveira, José Martins, Eduarda Mendes Rodrigues, and Luís Sarmento. 2012. TwitterEcho: A Distributed Focused Crawler to Support Open Research with Twitter Data. In *WWW*. 1233–1240.

[8] Donglin Cao, Xiangwen Liao, Hongbo Xu, and Shuo Bai. 2008. Blog Post and Comment Extraction Using Information Quantity of Web Format. In *AIRS*. 298–309.

[9] Carlos Castillo. 2005. Effective web crawling. In *ACM SIGIR Forum*, Vol. 39. 55–56.

[10] Zhijia Chen, Lihong He, Arjun Mukherjee, and Eduard C Dragut. 2024. Comquest: Large Scale User Comment Crawling and Integration.. In *SIGMOD Conference Companion*. 432–435.

[11] Zhijia Chen, Weiyi Meng, and Eduard Dragut. 2022. Web Record Extraction with Invariants. *Proceedings of the VLDB Endowment* 16, 4 (2022), 959–972.

[12] F. Chun-Long and M. Hui. 2012. Extraction technology of blog comments based on functional semantic units. In *CSAE*. 422–426.

[13] AnHai Doan, Jeff Naughton, Akanksha Baid, Xiaoyong Chai, Fei Chen, Ting Chen, Eric Chu, Pedro DeRose, Byron Gao, Chaitanya Gokhale, et al. 2009. The case for a structured approach to managing unstructured data. *arXiv preprint arXiv:0909.1783* (2009).

[14] Yongquan Dong, Eduard C. Dragut, and Weiyi Meng. 2019. Normalization of Duplicate Records from Multiple Sources. *IEEE Transactions on Knowledge and Data Engineering* 31, 4 (2019), 769–782.

[15] Eduard C. Dragut, Thomas Kabisch, Clement Yu, and Ulf Leser. 2009. A hierarchical approach to model web query interfaces for web source integration. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 325–336.

[16] Eduard C Dragut, Weiyi Meng, and Clement T Yu. 2012. Deep web query interface understanding and integration. *Synthesis Lectures on Data Management* 7, 1 (2012), 1–168.

[17] Grzegorz Dziczkowski, Lamine Bougueroua, and Katarzyna Wegrzyn-Wolska. 2009. Social network-an autonomous system designed for radio recommendation. In *CASoN*. 57–64.

[18] F. Erlandsson, R. Nia, M. Boldt, H. Johnson, and S. F. Wu. 2015. Crawling Online Social Networks. In *ENIC*. 9–16.

[19] Yixiang Fang, Xiaoqin Xie, Xiaofeng Zhang, Reynold Cheng, and Zhiqiang Zhang. 2018. STEM: a suffix tree-based method for web data records extraction. *KIS* 55, 2 (2018), 305–331.

[20] Johannes Fürnkranz. 1998. A study using n-gram features for text categorization. *OFAI* 3 (1998), 1–10.

[21] Khyatti Gupta, Meghana Joshi, Ankush Chatterjee, Sonam Damani, Kedhar Nath Narahari, and Puneet Agrawal. 2019. Insights from Building an Open-Ended Conversational Agent. In *Proceedings of the First Workshop on NLP for Conversational AI*. Association for Computational Linguistics, Florence, Italy, 106–112. https://doi.org/10.18653/v1/W19-4112

[22] Alon Halevy and Jane Dwivedi-Yu. 2023. Learnings from data integration for augmented language models. *arXiv preprint arXiv:2304.04576* (2023).

[23] Lihong He, Chao Han, Arjun Mukherjee, Zoran Obradovic, and Eduard Dragut. 2020. On the dynamics of user engagement in news comment media. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* (2020), e1342.

[24] Lihong He, Chen Shen, Arjun Mukherjee, Slobodan Vucetic, and Eduard Dragut. 2021. Cannot predict comment volume of a news article before (a few) users read it. In *Proceedings of the International AAAI Conference on Web and Social Media*, Vol. 15. 173–184.

[25] Jingtian Jiang, Xinying Song, Nenghai Yu, and Chin-Yew Lin. 2012. Focus: learning to crawl web forums. *TKDE* 6 (2012), 1293–1306.

[26] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2017. Bag of Tricks for Efficient Text Classification. In *EACL*. 427–431.

[27] Huan-An Kao and Hsin-Hsi Chen. 2010. Comment Extraction from Blog Posts and Its Applications to Opinion Mining.. In *LREC*. 1113–1120.

[28] Nicholas Kushmerick. 1997. *Wrapper induction for information extraction*. University of Washington.

[29] Kristina Lerman, Steven N Minton, and Craig A Knoblock. 2003. Wrapper maintenance: A machine learning approach. *JAIR* 18 (2003), 149–181.

[30] Haoxin Liu, Ziwei Zhang, Peng Cui, and Yafeng et al. Zhang. 2021. Signed Graph Neural Network with Latent Groups. In *KDD*. 1066–1075.

[31] Wei Liu, Xiaofeng Meng, and Weiyi Meng. 2006. Vision-based web data records extraction. In *Proc. 9th international workshop on the web and databases*. 20–25.

[32] Jenny Mandl. 2016. Why are all programming languages in English? shorturl.at/crxB7. Accessed: 2021-10-22.

[33] Gengxin Miao, Junichi Tatemura, Wang-Pin Hsiung, Arsany Sawires, and Louise E Moser. 2009. Extracting data records from the web using tag path clustering. In *Proceedings of the 18th international conference on World wide web*. 981–990.

[34] Ankan Mullick, Sayan Ghosh, Ritam Dutt, Avijit Ghosh, and Abhijnan Chakraborty. 2019. Public Sphere 2.0: Targeted Commenting in Online News Media. In *ECIR*. Springer, 180–187.

[35] M. Neunerdt, M. Niermann, R. Mathar, and B. Trevisan. 2013. Focused crawling for building Web comment corpora. In *CCNC*. 685–688.

[36] Pieter Noordhuis, Michiel Heijkoop, and Alexander Lazovik. 2010. Mining twitter in the cloud: A case study. In *CLOUD*. 107–114.

[37] Stefano Ortona, Giorgio Orsi, Marcello Buoncristiano, and Tim Furche. 2015. Wadar: Joint wrapper and data repair. *VLDB* 8, 12 (2015), 1996–1999.

[38] Mohammad Raza and Sumit Gulwani. 2017. Automated data extraction using predictive program synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 31.

[39] Kai Shu, Limeng Cui, Suhang Wang, Dongwon Lee, and Huan Liu. 2019. defend: Explainable fake news detection. In *KDD*. 395–405.

[40] Xinying Song, Jing Liu, Yunbo Cao, Chin-Yew Lin, and Hsiao-Wuen Hon. 2010. Automatic extraction of web data records containing user-generated content. In *CIKM*. 39–48.

[41] Marija Stanojevic, Jumanah Alshehri, Eduard Dragut, and Zoran Obradovic. 2019. Biased News Data Influence on Classifying Social Media Posts. In *NewsIR co-located with SIGIR*, Vol. 2411. 3–8.

[42] Florian Toepfl and Eunike Piwoni. 2015. Public Spheres in Interaction: Comment Sections of News Websites as Counterpublic Spaces. *Journal of Communication* 65 (2015), 465–488.

[43] Roberto Panerai Velloso and Carina F Dorneles. 2017. Extracting records from the web using a signal processing approach. In *CIKM*. 197–206.

[44] Roberto Panerai Velloso and Carina F Dorneles. 2017. Extracting records from the web using a signal processing approach. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 197–206.

[45] Andrew Wang, Rex Ying, Pan Li, Nikhil Rao, Karthik Subbian, and Jure Leskovec. 2021. Bipartite Dynamic Representations for Abuse Detection. In *KDD*. 3638–3648.

[46] Patrick Weber. 2014. Discussions in the comments section: Factors influencing participation and interactivity in online newspapers' reader comments. *New Media & Society* 16, 6 (2014), 941–957.

[47] Fan Yang, Eduard Dragut, and Arjun Mukherjee. 2020. Predicting Personal Opinion on Future Events with Fingerprints. In *COLING*. 1802–1807.

[48] Junting Ye and Steven Skiena. 2019. *MediaRank: Computational Ranking of Online News Sources.* 2469–2477.

[49] Reza Zafarani, Xinyi Zhou, Kai Shu, and Huan Liu. 2019. Fake news research: Theories, detection strategies, and open problems. In *KDD*. 3207–3208.

[50] Mohammed J Zaki and Wagner Meira Jr. 2020. *Data Mining and Machine Learning: Fundamental Concepts and Algorithms.* Cambridge University Press.

[51] Yanhong Zhai and Bing Liu. 2005. Web data extraction based on partial tree alignment. In *WWW*. 76–85.

[52] Hongkun Zhao, Weiyi Meng, Zonghuan Wu, Vijay Raghavan, and Clement Yu. 2005. Fully automatic wrapper generation for search engines. In *WWW*. 66–75.

[53] Zhuojie Zhou, Nan Zhang, and Gautam Das. 2015. Leveraging history for faster sampling of online social networks. *arXiv preprint arXiv:1505.00079* (2015).

[54] Marc Ziegele and Oliver Quiring. 2013. Conceptualizing online discussion value: A multidimensional framework for analyzing user comments on mass-media websites. *Annals of the Int. Comm. Assoc.* 37, 1 (2013), 125–153.