Full-Page Wrapper Generation for Unsupervised Deep Web Data Extraction

Full-Page Wrapper Generation for Unsupervised Deep Web Data Extraction

Chia-Hui Chang, Naufal Said, Oviliani Yenty Yuliana, and Yu-Hao Wu

Abstract—Web data extraction is a key component in many business intelligence tasks, such as data transformation, exchange, and analysis. Many approaches have been proposed, with either labeled training examples (supervised) or annotation-free training pages (unsupervised). However, most research focuses on extraction effectiveness. Not much attention has been paid to extraction efficiency. In fact, most unsupervised web data extraction ignores wrapper generation because they could work alone without any supervision. In this paper, we argue that wrapper generation for unsupervised web data extraction is as important as supervised wrapper induction because the generated wrappers could work more efficiently without sophisticated analysis during testing. We consider two approaches for wrapper generation: schema-guided finite-state machine (FSM) approaches and data-driven machine learning (ML) approaches. We exploit unique mandatory templates to improve the FSM-based wrapper, and proposed two convolutional neural network (CNN)-based models for sequence-labeling. The experimental results show that the FSM wrapper performs well even with small training data, while the CNN-based models require more training pages to achieve the same effectiveness but are more efficient with GPU support. Furthermore, FSM wrappers can work as a filter to reduce the number of training pages and advance the learning curve for wrapper generation.

Index Terms—Full-page wrapper generation, wrapper verification, FSM based wrapper, neural sequence wrapper.

1 Introduction

Web information extraction refers to the task of extracting information of specific types from webpages and generating structured data for information integration applications. Depending on the information source and extraction target, we can define different extraction tasks and design methods with different automation degree. For example, extracting business names and their address pairs from company websites can facilitate business database construction, while extracting job information from posts in newsgroups is essential for building a job search service. The input pages of these **traditional information extraction** usually comes from different web resources, but the extraction target is more specific. On the other hand, **deep Web data extraction** refers to extracting data from the search result pages generated by the same server-side program. In other words, the extraction target can be any data embedded in the page, but the input pages come from the same website.

For the former extraction tasks, since the input webpages come from different websites, we usually require labeled training samples to specify the extraction targets. Therefore, it is generally necessary to apply *supervised machine learning* methods to generate wrapper programs. For example, SoftMealy [1] and Stalker [2] are two early works on Web information extraction, which induce and generalize extraction rules based on domain-independent token taxonomy trees and represent wrappers as finite state transducers.

On the other hand, the input pages for deep Web data extraction are dynamic pages that are generated from the same server-side script with some template. Thus, we can utilize the regularity of search results to discover extraction patterns without labeled training examples. In other words, *unsupervised approaches* are possible for deep Web data extraction. For example, IEPAD [3] and MDR [4] are two pioneer works that accept a single web page as input and output the main search results (also called record set or data-rich section). However, a single webpage input is limited to extracting **record-level** data. To deal with **page-level** or **full-page** data extraction, extraction systems require the input of multiple pages. Compared with extracting record-level data from a single page, extracting page-level data from multiple pages with the same template is much more challenging and still attracted considerable research attention, e.g. EXALG [5], RoadRunner [6], FivaTech [7], TEX [8], DCADE [9].

Supervised and unsupervised data extraction work very differently. The task of supervised approaches is to learn data extraction rules from manually labeled training examples and output a wrapper that implement the programming logic of the given schema, while the task of the unsupervised method is to explore templates in the input page and align data items (with the same type) into a structured format. Therefore, supervised methods are usually accompanied by wrapper generation, while unsupervised methods usually stop at template derivation and ignore wrapper generation.

Ideally, a deep Web wrapper extracts the data instances from testing pages on the same website. If the constructed wrapper fails on a testing page from the same website, we need to regenerate wrappers as shown in the "Fail" link of Fig. 1(a). Note that, even if the wrapper works

- *C.-H. Chang is with the Department of Computer Science and Information Engineering, National Central University, Taiwan, 32001*
  E-mail: chia@csie.ncu.edu.tw
- *O. Y. Yuliana is with Faculty of Business and Economics, Petra Christian University, Indonesia, 60236*
  E-mail: oviliani@petra.ac.id
- *N. Said and Y.-H. Wu are with National Central University*
  E-mail: naufal.said19@gmail.com and yuhao8888@gmail.com

Manuscript received Sep 14, 2021; revised xxx xx, 2021.

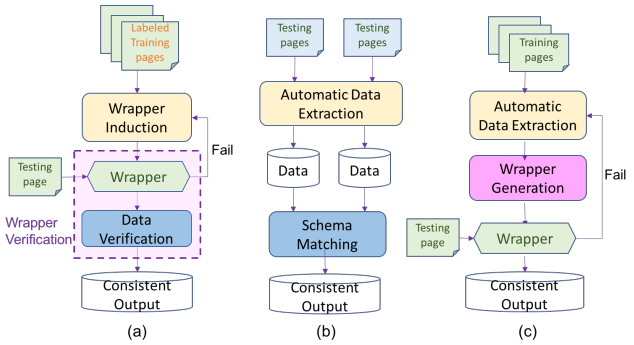

Fig. 1. Three frameworks for (a) Supervised wrapper induction and (b) Unsupervised data extraction without wrapper generation and (c) Unsupervised data extraction with wrapper generation. The problem we address in this paper is the purple module "wrapper generation" from the output of automatic data extraction systems.

properly, the extracted data may still be inconsistent with the schema. Thus, a data verification procedure is required to ensure that the wrapper produces valid extracted data [10], [11]. The procedure of verifying whether a wrapper is operating correctly and producing valid extracted data is called wrapper verification (see the dotted purple rectangle in Fig. 1(a)). This loop of wrapper induction and verification defines the life cycle of the wrapper generation process.

On the other hand, since unsupervised approaches can be executed without labeled training page, it can be deployed directly without generating a wrapper. Thus, no wrapper maintenance work is required. However, since the web page are subject to change, we need a schema matching module to integrate the output of the new page and the old page to ensure that the output are consistent (see Fig. 1(b)). The inherent problem with this setting is the efficiency issue for a large number of testing pages. This is because the complex analysis procedure during automatic data extraction must be done once for every page. Meanwhile, the effectiveness of the final output depends on the schema matching module, where the performance may degrades because of pipe-lined error. The issue may get worse for full-page data extraction because the number of data items could increase to hundreds or thousands. Compared with record-level data extraction, where only tens of attributes/fields are to be matched, the complexity of full-page schema matching is greatly increased.

In this paper, we propose the idea of wrapper generation for unsupervised full-page data extraction as shown in Fig. 1(c). The benefit is that this schema (and template) guided wrapper does not require additional verification. If the wrapper fails, it implies some change in the new page and we can add it to the old training pages for new schema and template induction (see the loop in Fig. 1(c)). Meanwhile, schema-guided wrapper does not require complex analysis like that in training phase but simply align the testing page with the schema and template.

Given the output of unsupervised full-page data extraction, we define the problem of full-page wrapper generation and consider two approaches for wrapper generation: data-driven machine learning (ML) approach and schema-guided finite-state machine (FSM) construction. Data-driven ML approaches take the output of automatic data extraction methods as annotated examples and train a sequence-labeling model; while schema-guided FSM methods use the derived schema and template to construct FSMs and apply the universal FSM driver to extract data from testing pages. The experimental results show that the data-driven ML methods are easy to build but require a large number of training data, while the schema-guided FSM approach is more efficient in both the training and testing phases.

We organize the rest of the paper as follows. Section 2 discusses and compares the different existing approaches to Web scraping services and wrapper generation. We define the problem of full-page data extraction and its wrapper generation in Section 3. Section 4 and 5 present our schema-guided FSM wrapper generation as well as the three sequence-labeling models. The experimental results are presented in Section 6. Finally, we conclude the paper and outline the directions for future research.

2 Related Work

Web data extraction is the core technique for Web scraping tools, which can automate the collection of information from websites. For example, electronics retailers, hotels, and supermarket chains use it extensively to perform market research, such as monitoring the product daily prices offered by their competitors. Web data extraction accepts webpages fetched by web crawlers and outputs structured data for subsequent use. Most academic research ignores the problem of page download and focuses on web data extraction from input pages, while commercial tools need to support both page crawling and data extraction at the same time to provide a total solution.

Most commercial web scraping tools, e.g. Dexi.io (https://dexi.io/), Import.io (https://www.import.io/), ParseHub (https://www.parsehub.com/), and OctoParse [12], etc. adopt a supervised approach, requiring users to annotate the extraction targets through a graphical user interface. Some of them are designed for programmers, while others are for users with no coding background. For example, Dexi.io helps IT professionals extract, process, transform and aggregate important data from websites. In other words, Dexi.io users must have HTML, CSS, and javascript backgrounds. Compare to dexi.io, OctoParse is a scraping tool with no nodejs learning or programming required. As demonstrated in OctoParse 7.x[1], users can create a paging loop to tell the system which page the data exists as well as create a loop item to repeat the click action on the data. Although no actual coding is required, users need to have clear logic in order to complete the interleaving operations of page downloading and data extraction. Recently, OctoParse 8.x Beta[2] has adopted unsupervised analysis from a single page for structured data preview to reduce annotation efforts.

As mentioned in the first section, the studies of web data extraction have evolved from supervised approaches to unsupervised approaches. For supervised wrapper induction, two major tasks are performed for wrapper maintenance, i.e., wrapper verification and wrapper re-induction [13].

1. https://www.youtube.com/watch?v=j_JWaMnsXWQ
2. https://helpcenter.octoparse.com/hc/en-us/articles/900001194903-Video-Advanced-Mode

Wrapper verification is used to verify whether a wrapper is operating correctly and producing valid extracted data [10], whereas wrapper re-induction is used to repair the wrapper by gaining new labeled training data to learn new extraction rules.

For unsupervised web data extraction that accepts a single page input, the extraction systems, such as AutoRM [14], SYNTHIA [15], and Dual-TLBO [16]), process each page independently to extract the data-rich area (usually the search result). Since the different input pages may generate different output schema, an additional step to integrate all extracted data (schema matching) is required for these "wrapper-free" unsupervised extraction systems. In contrast, the advantage of unsupervised web data extraction for multi-page input is that it allows the system to compare elements from multiple pages to determine whether they are templates or data, options, or fixed appearances. Therefore, such a system can infer the complete pattern, i.e. full-page schema of the input page. Example systems include EXALG [5], RoadRunner [6], FivaTech [7], TEX [8], and DCADE [9].

However, most of these full-page unsupervised data extraction methods do not address the problem of wrapper generation. The only work to generate wrappers for unsupervised full-page web data extraction was RoadRunner. However, as mentioned in [5], several issues exist in RoadRunner, including the assumption of union-free grammar, examining training pages one by one, no HTML tags in the data fields, and exponential search complexity in the size of the schema, etc. Thus, subsequent researches have focused on aligning multiple input pages at the same time to reduce the limitation of local page alignment.

In practice, the derivation of the template and schema of a website is not the end of the unsupervised data extraction. We still need a wrapper that can extract all embedded data from the testing page that conforms to the template and schema. In other words, unsupervised web scraping services not only need to infer web templates and schema from input webpage, but also have to generate wrappers to extract data from the testing page of the same information source.

Another motivation of generating wrappers for unsupervised web data extraction methods comes from the need for high throughput data extraction, which is a major concern of commercial web scraping tools. For example, FastWrap [17] transforms a visual-based schema into a browser-less wrapper to reduce the inefficiency of page rendering and JavaScript execution in visual-based wrappers and speed up the testing/wrapping process. However, FastWrap is limited to flat data records, allowing no optional fields and no compound data values or nesting of records, whereas the proposed methods in this work do not have such constraints.

3 Problem Definition

In order to provide unsupervised web scraping services, we built a prototype system[3] to create data APIs for deep Web pages such that users can easily extract, transform and load data. In this paper, we adopt DCADE [9] for full-page schema induction. DCADE (Divide-and-Conquer Alignment with Dynamic Encoding) is a full page web data extraction system which adopts a divide-and-conquer approach to align the text nodes from input pages. DCADE features dynamic encoding (Fig. 3) such that the text contents of the text nodes could be abstracted to some common code for alignment.

3. Web data ETL system: http://140.115.54.44:8001

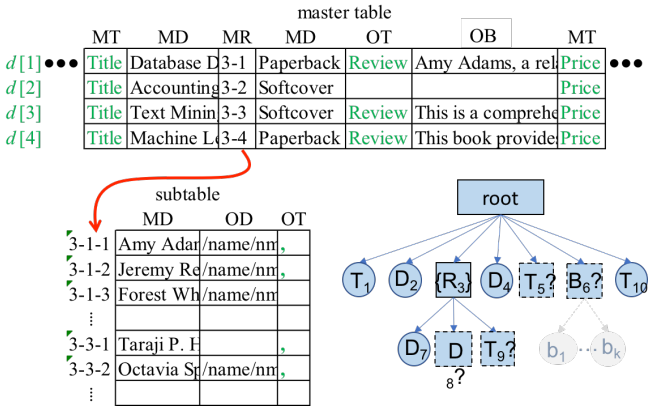

Fig. 2. Example output of full-page data extraction system DCADE and its corresponding page schema.

Formally, given m input webpages that are represented as lists of text nodes, DCADE follows EXALG's [5] definition of structured data to output a master table (of m rows) and k subtables for k record sets (i.e. one subtable for each record set). Depending on the abstraction level of the input pages, there could be hundreds or thousands of text nodes.

- Each column in the aligned table is considered a template (T) if it has the same text content for all rows. Otherwise, it represents a basic data type, D (i.e., the target of unsupervised web data extraction).
- A subtable denotes a type of set where the records (R) occur repetitively in the pages.
- A data column that merges more than one heterogenous text node is called a bag (B).
- For each column, if elements are missing in a row, we consider it an optional type (O); otherwise, we call it mandatory (M).
- If the adjacent columns play the same roles and complement each other's occurrence, they could be merged using a disjunctive type of constructor.

Page Schema Example Fig. 2 shows a part of one master table (with five columns labeled as MD, MR, MD, OT, OB, and MT in order) and a subtable (with three columns labeled as MD, OD, and OT) from the output of DCADE in Fig. 2, the page schema in this segment contains a seven-tuple $<T_1, D_2, \{R_3\}, D_4, T_5?, B_6?, T_{10}>$ with three mandatory templates (T_1="Title", T_5="Review:", and T_{10}="Price"), two mandatory basic data (D_2 and D_4), a set $\{R_3\}$, and a bag B_6, where record set $\{R_3\}$ is composed of two basic data (D_7 and D_8) and one template ($T_9 =$ ","), while B_6 is a composite data corresponding to the OC column in the master table and can match multiple text nodes without restriction (denoted by $b_1, \ldots, b_k$). Note that optional data are denoted by dotted rectangles as shown by T_5, B_6, D_8 and T_9.

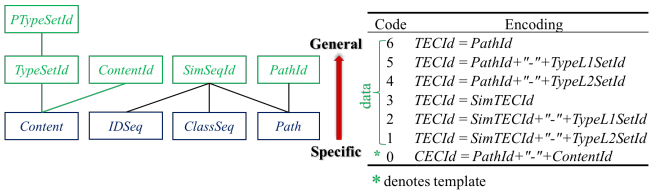

Fig. 3. Encoding scheme used in DCADE [9] based on basic attributes: content, id, class, and tag path of each node.

Wrapper Generation Given the aligned tables obtained from full-page data extraction systems, the task of the wrapper generation is to produce a program that can check whether the given testing page complies with the page schema and aligns the text nodes in the testing page with the corresponding page schema, achieving the goal of data extraction.

In this paper, we propose two kinds of wrapper generation from the output of full-page data extraction: the FSM-based method and ML-based method, as described in Section 4 and 5.

4 FSM-BASED WRAPPER GENERATION

Almost all supervised data extraction adopts FSM-based wrappers, but with varying degrees of expressiveness. For example, Kushmerick et al. [10] proposed a fixed architecture called HLRT (head-left-right-tail), but missing data is not permitted. Muslea et al. [2] proposed disjunctive landmark automata to improve the expressiveness of the extraction rules. Hsu and Chang [1] adopted a hierarchical structure and contextual rules to handle missing items. Dalvi et al. [**?**] applied a semi-supervized approach to learn wrappers effectively from automatically labeled training data (using dictionaries and regular expressions). However, no matter what wrapper architecture is used, the previous work on record set extraction is only a part of the full-page wrapper architecture.

Given the leaf node list of a test webpage, our goal is to align each leaf node with some column in the master table or the subtable based on the column labels (mandatory or optional, template or data). Except for set and bag, each column can only consume one leaf node. As shown in Fig. 4, we use the aligned master table and subtables to generate FSMs in the training phase (left) and design an universal wrapper to verify whether the testing page (represented by a node list) conforms with the generated FSMs in the testing phase (right).

The earliest work on full-page wrapper generation is proposed for FivaTech [7] by Change et al. in [18], who modeled the FSM driver as a constraint satisfaction problem. We followed the track to build full-page wrapper for DCADE output. Chang et al. [19] split input pages into training and testing set, and apply unsupervised data extraction, UWIDE, to infer website template and schema. By constructing an FSM-based wrapper, they can efficiently extract data from the remaining input pages with a performance 2.7 times faster than unsupervised methods.

The challenge here is how to speed up the processing time when several optionals appear adjacently. The idea is

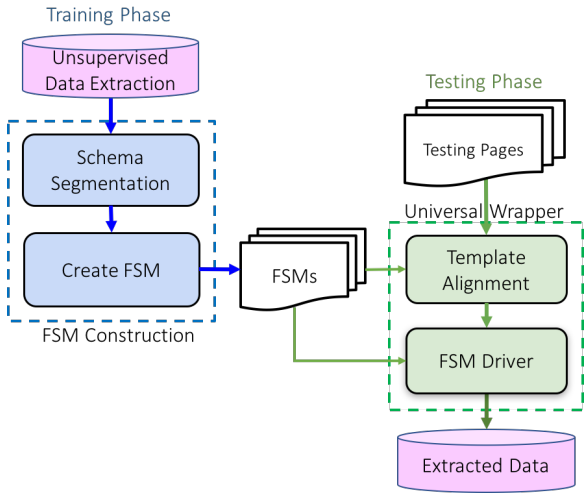

Fig. 4. Finite-State Machine (FSM)-based wrapper generation.

to use mandatory templates (MT) to divide the entire page schema into smaller sub-problems to reduce the complexity. However, not all mandatory templates are used for segmentation because mandatory templates and optional templates may have the same code. Therefore, in the training phase, we first find the "unique" mandatory template to segment the full-page schema, and then construct a FSM for each segment. Following the same logic, in the testing phase, we must perform mandatory template alignment before the execution of individual FSM driver (see Fig. 4).

4.1 Training Phase: FSM Construction

The training phase (Algorithm 1) consists of two parts: the first part selects all the unique mandatory templates to partition the full schema into segments, whereas the second part creates an FSM for each segment. Let $Schema$ denote the unsupervised extraction output that contains the master table $mainT$ and a set of subtables $subT$. Collecting distinct MT works as follows. First, we collected all MT (line 1) columns in the master table $mainT$ and then checked whether each mandatory template t exists in the previous segment ($MTList[t-1]$, $MTList[t]$) or the next segment ($MTList[t]$, $MTList[t+1]$). Note that we checked the existence of t not only in the master table $mainT$ but also in the subtables (line 5). If t is distinct from its adjacent segments, we add t to *UniqMT* (line 6). If in the end of the for-loop, $UniqMT$ is the same as $MTList$ (line 9), we proceed to the next step at line 14. Otherwise, the same procedure from line 2 to 13 is executed again to ensure each mandatory template is unique in the adjacent segments.

The second part of Algorithm 1 creates an FSM for each non-empty segment in $UniqMT$ (line 16), and adds each created fsm to the list, $FSMList$ (line 20). Each FSM consists of a linked list of nodes generated by $createStates$ (line 18) and a set of arcs added by $createARCs$ (line 19). The output of the $FSMConstruction$ algorithm is the list of distinct MTs ($UniqMT$) and the set of FSMs ($FSMList$).

Algorithm 2 shows the procedure to create arcs. For each node nd in $nodeList$, we add arcs from node $nd.prev$ to nd (line 3) and all the "possible" following nodes because of

Algorithm 1 FSMConstruction ($Schema$)

1: $MTList \leftarrow getMT(Schema.mainT)$;
2: **loop**
3: $UniqMT \leftarrow \emptyset$;
4: **for** $t \leftarrow 1$ to $|MTList| - 1$ **do**
5: **if** $!Schema.CheckExist(MTList[t], MTList[t-1], MTList[t+1])$ **then**
6: $UniqMT.add(MTList[t])$;
7: **end if**
8: **end for**
9: **if** $UniqMT = MTList$ **then**
10: **b**reak;
11: **end if**
12: $MTList \leftarrow UniqMT$;
13: **end loop**
14: $FSMList \leftarrow \emptyset$;
15: **for** $t \leftarrow 1$ to $|UniqMT|$ **do**
16: **if** $UniqMT[t] - UniqMT[t-1] > 1$ **then**
17: $arcs \leftarrow \emptyset$;
18: $nodeList \leftarrow createStates(mainT, UniqMT[t-1], UniqMT[t])$;
19: $fsm \leftarrow createArcs(nodeList, arcs)$;
20: $FSMList \leftarrow FSMList \cup fsm$;
21: **end if**
22: **end for**
23: **return** $UniqMT, FSMList$

Algorithm 2 createARCs($nodeList, arcs$)

1: **for all** nd in $nodeList$ **do**
2: **if** $nd.prev = \emptyset$ **then** continue; **end if**
3: $arcs.add(nd.prev, nd, countInst(nd.prev, nd))$;
4: $temp \leftarrow nd$;
5: **while** $temp.Type$ is $(OT|OD|OR|OC)$ **&** $temp.nxt \neq \emptyset$ **do**
6: $count \leftarrow countInst(nd.prev, temp.nxt)$;
7: $arcs.add(nd.prev, temp.nxt, count)$;
8: $temp \leftarrow temp.nxt$;
9: **end while**
10: **if** $nd.Type$ is $(MC|OC)$ **then**
11: $arcs.add(nd, nd, countInst(nd, nd))$;
12: **end if**
13: **if** $nd.Type$ is $(MR|OR)$ **then**
14: $subT \leftarrow Schema.subT[nd.RecSet]$;
15: $nodeList2 \leftarrow createStates(subT, 1, |subT|)$;
16: $fst \leftarrow nodeList2.firstnode()$;
17: $fst.prev \leftarrow nd.prev$;
18: $last \leftarrow nodeList2.lastnode()$;
19: $last.nxt \leftarrow nd.nxt$;
20: $createARCs(nodeList2, arcs)$;
21: **for all** arc e ends with nd in arcs **do**
22: $e.end \leftarrow fst$;
23: **end for**
24: $nodeList.replace(nd, nodeList2)$;
25: $arcs.add(last, fst, countInst(last, fst))$,
26: **while** $fst.TYPE$ is option **&** $fst.nxt \neq last$ **do**
27: $count \leftarrow countInst(last, fst.nxt)$;
28: $arcs.add(last, fst.nxt, count)$;
29: $fst \leftarrow fst.nxt$;
30: **end while**
31: **end if**
32: **end for**
33: **for all** nd in $nodeList$ **do**
34: Compute the cumulated transition count ctc for nd;
35: Compute $e.prob$ for all $e \in arcs$ starts with nd with Eq. (1);
36: Compute feature value distribution used in Eq. 3
37: **end for**
38: **return** $FSM(nodeList, arcs)$;

optional nodes (line 5-9), bags (line 10-12), or record sets (line 13-31). Specifically, we create arcs as follows:

- If nd is optional (line 4-5), it can be skipped, and we add a new arc from $nd.prev$ to its next node $temp.nxt$, where $countInst(a, b)$ computes the number of transitions from node a to b. The process repeats until no optional node is found or until the end of the segment (line 5-9).
- If nd is a bag, denoted by MC or OC (line 10), we add an arc from nd to itself (line 11).
- If nd is a record set, either MR or OR (line 13), we retrieve the subtable $subT$ pointed to by $nd.RecSet$ (line 14) to create a corresponding fsm for the whole segment (1, $|subT|$). To link the new fsm with the existing one, we assign the first and last node of $nodeList2$ with a previous link to $nd.prev$ and next link to $nd.nxt$ respectively before the recursive call to $createARCs$ for all nodes in the new $nodeList2$ (line 20). We then redirect all arcs e that ends with nd to the first node of the $nodeList2$ (line 22) and replace the current node nd with $nodeList2$ (line 24). Since a record set allows multiple records in one page, we add an arc from the last node of the new fsm to the first node (line 25) and the following nodes if they are optional (line 26-30).

Finally, we calculate the cumulated transition count ctc originated from nd and normalize the transition probability (line 34-35) with Eq. 1 and compute the feature value distribution probability for each node (line 36).

$$e.prob = \frac{e.count + \alpha}{ctc + \alpha \times deg}, \alpha = \begin{cases} 0.1 & ctc \leq 100 \\ 1 & otherwise \end{cases} \quad (1)$$

where deg is the number of outgoing links from nd and α is the smoothing parameter ($\alpha > 0$).

Example 4.1. Fig. 5 shows the FSM created from the segment in Fig. 2. The order that the system adds transition arcs for each node is marked by colored numbers. Since a segment from $mainT$ starts and ends with MT, they could act as the begin and end states for the FSM. However, a $subT$ may start or end with optional nodes, we therefore add $nd.prev$ and $nd.nxt$ to the first and last nodes in $nodeList2$ (line 16 -19) to link two $nodeLists$. When the last column of the input schema is an optional node (e.g. T_9), the system can follow the $last.nxt$ link to add arcs from previous optional nodes (e.g. D_8 and T_9) to the following node (D_4).

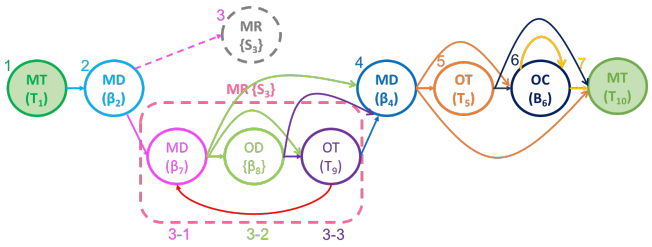

Fig. 5. Illustration of FSM for Fig. 2.

4.2 Testing Phase: Universal Wrapper

The testing phase requires a universal wrapper to verify whether a page complies with the constructed FSM represented by $UniqMT$ and $FSMList$. The step for the universal wrapper includes aligning the page with $UniqMT$ for page segmentation (line 3) and verification of each segment with the corresponding FSM (line 20). The complete algorithm is shown in Algorithm 3.

The design of the template alignment is to accelerate the verification process by dividing the input page into segments. In this paper, we adopt the Needleman-Wunchman algorithm with the following matching score: 1 for the match value, -4 for the mismatch value, and -2 for the gap penalty. The output of the pairwise string alignment algorithm adds null values to $UniqMT$ and the $page$ to ensure they are well aligned (line 3) with equal length, as shown in Fig. 6. The output of the sequence alignment is the indices P and S for the testing page and schema, respectively by tracing back of the scoring matrix. Note that the codes (e.g. 9-143, 9-203) represent the least common encoding (as shown in Fig. 3) for data cells aligned in the same column of the schema. For mandatory templates in $UniqMT$, only text nodes with the same text content and similar path are considered a match.

Let P_j and S_j denote the aligned page index and the schema index. When S_j is null, it indicates a new segment is found in the page (line 7-8). If P_j is null, i.e., MT not found, or the corresponding codes in the testing page and schema mismatch (line 9-12), then the algorithm returns FALSE and exit. Let I_{mt} and I_{fsm} be the index to $UniqMT$ and $FSMList$. The $UniqMT$ index I_{mt} is increased (line 14) when the end of a new segment is reached, i.e., S_j is not null, while the FSM index I_{fsm} is increased (line 16) when the gap between $UniqMT[I_{mt}]$ and $UniqMT[I_{mt}-1]$ is greater than one. This is the same condition that we use to create an FSM at line 16 in Algorithm 1.

For each new segment from the last position L_p (initially 0) to P_j in the input $page$, we call $FSM.Driver$ (line 20) to verify whether the new segment is acceptable (line 18-27). If the Driver returns TRUE, the repeat loop (line 23-25) stops and returns current $output$ by concatenating with the aligned $AlignSeg$ (line 22). Because an FSM in the $FSMList$ could be an optional (just like the fsm-2 example in Fig. 6), each segment is given the chance to match with other FSM from L_{fsm} up to I_{fsm} computed above (line 21). If no FSM returns TRUE, Algorithm 3 returns FALSE and exit.

Algorithm 3 UniversalWrapper($page$)

1: Input: $Schema, UniqMT, FSMList$;
2: $Output \leftarrow \emptyset$
3: $P, S \leftarrow Schema.seqAlign(UniqMT, page)$
4: $newSeg \leftarrow FALSE$;
5: $I_{mt}, I_{fsm}, L_{fsm}, L_p \leftarrow 0$;
6: **for** $j \leftarrow 1$ **to** $|P|$ **do**
7: **if** $S_j = \emptyset$ **then**
8: $newSeg \leftarrow TRUE$; continue;
9: **else if** $P_j = \emptyset$ **then**
10: **return** $FALSE, Output$;
11: **else if** $page[P_j] \neq Schema.mainT[S_j]$ **then**
12: **return** $FALSE, Output$;
13: **end if**
14: I_{mt}++;
15: **if** $UniqMT[I_{mt}] - UniqMT[I_{mt}-1] > 1$ **then**
16: I_{fsm}++;
17: **end if**
18: **if** $newSeg$ **then**
19: **repeat**
20: $Success, AlignSeg \leftarrow$ $FSMList[L_{fsm}++].Driver(Page, L_p, P_j)$;
21: **until** $Success$ — $L_{fsm} > I_{fsm}$
22: $Output \leftarrow Output + AlignSeg$;
23: **if** !$Success$ **then**
24: **return** $Success, Output$;
25: **end if**
26: $newSeg \leftarrow FALSE$;
27: **end if**// $Schema.mainT[S_j] = page[P_j]$
28: $Output \leftarrow Output + page[P_j]$;
29: $L_p \leftarrow P_j$;
30: **end for**
31: **return** $TRUE, Output$;

Finite-state machine (FSM) Driver

Whereas an FSM can easily reject an input if no proper state can be found, the more serious problem is when multiple candidate states (or transitions) are found for an input. To assign the best state for each text node, or equivalently, to find the best state sequence for an input segment, we apply dynamic programming as shown in Algorithm 4. The first loop (line 3 to 10) finds candidate states for each text node in the input segment (line 4) and the next for-loop implements dynamic programming to compute the maximum score for subsegment from the start to current position (line 13 to 17). Suppose $dp[i][j]$ represents the maximum score for the subsegment from $start$ to $start+i$ (i.e., $page(start, start{+}i)$), with the last text node $page[start+i]$ labeled with state $cand[i][j]$, then $dp[i][j]$ can be recursively defined by $dp[i-1][k]$ as shown in Eq. (2), where k ranges from 1 to $cand[i-1].leng$ (the number of candidate states).

$$dp[i][j] = \max_k\{dp[i-1][k] + NBscore(Page[start+i]|cand[i][j]) + arcs.prob(cand[i-1][k], cand[i][j])\} \quad (2)$$

Note the state transition probability is defined in Eq. (1) in Algorithm 2, while the $NBscore$ of assigning $start+i$

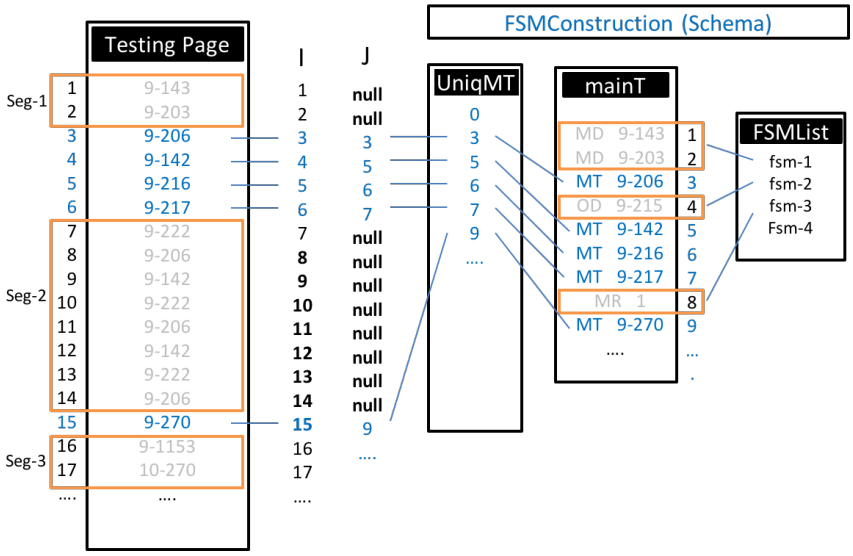

Fig. 6. Illustration of the template alignment with a testing page.

Algorithm 4 FSM.Driver ($Page, start, end$)

1: Input: $FSM(nodeList, arcs)$;
2: $max\_states \leftarrow 0$;
3: **for** $i \leftarrow 0$ **to** $end - start$ **do**
4: $cand[i] \leftarrow findCand(Page[start+i], nodeList)$
5: **if** $cand[i] = \emptyset$ **then**
6: **return** $FALSE$ //testWeb rejected
7: **else if** $cand[i].leng > max\_states$ **then**
8: $max\_states \leftarrow cand[i].leng$;
9: **end if**
10: **end for**
11: Initialize dp[end-start+1][max_states];
12: Initialize $prev\_state$[end-start+1][max_states];
13: **for** $i \leftarrow 1$ **to** end-start+1 **do**
14: **for** $j \leftarrow 1$ **to** $cand[i].leng$ **do**
15: compute $dp[i][j]$ with Eq. (2) and
assign $prev\_state[i][j]$ with the argmax of $dp[i][j]$
16: **end for**
17: **end for**
18: $last \leftarrow end - start + 1$;
19: $j^* \leftarrow arg\max_j dp[last][j]$;
20: $Output[last] \leftarrow NodeList[cand[last][j^*]]$;
21: $k \leftarrow prev\_state[i][j^*]$;
22: **for** $i \leftarrow last - 1$ DOWN **to** 1 **do**
23: $Output[i] \leftarrow NodeList[cand[i][k]]$;
24: $k \leftarrow prev\_state[i][k]$;
25: **end for**
26: **return** $TRUE, Output$;

text node to state $cand[i][j]$ is computed by the following:

$$NBscore(v|state) = \sum_c logP(v.c|state) \quad (3)$$

where c belongs to one of the five derived codes (including $ContentId$, $PathId$, $TypeSet$, $PTypeSet$, and $SimSeqID$) from DCADE based on the text content and tag path of text node v, as illustrated in Fig. 3 and described in section 5.1.

We maintain the index of the previous state that results in the maximum $dp[i][j]$ in $prev\_state[i][j]$, such that the algorithm can trace back using the dp and $prev\_state$ matrices to find the best state sequence for the whole segment. Let j^* be the index that leads to the maximum score in the last row of the dp matrix, denoted by $\max_j dp[last][j]$, where $last$ is initialized with $end - start + 1$ (line 19). With the $prev\_state$ matrix, we can obtain its previous state from $k \leftarrow prev\_state[last][j^*]$ (line 21). Finally, we trace back through $prev\_state[i][k]$ for every text node $i < last$ (line 22 to 25) and determine the best state sequence $Output[i]$ for each text node $Page[start + i]$ (line 20 and 23).

The design of aligning the input text node sequence with the $UniqMT$ alignment to divide the input page into segments in Algorithm 3 not only accelerates the verification process but also improves the output state sequence accuracy, as we see in the experiment section.

5 ML-Based Wrapper Generation

As we can see, constructing FSM-based wrappers requires sophisticated algorithm design. If we treat unsupervised data extraction as an oracle machine to generate annotated training examples, constructing ML-based wrappers will be relatively simple. By taking input pages as labeled training examples (assuming each text node in the input page is assigned a label of the column id), we can train a sequence-labeling model to predict the column id for each text node of a testing page. Formaly, For an output schema with one master table and k subtables, we can model the problem as k+1 sequence-labeling problems.

Each column has a distinct label and the total number of labels for each sequence-labeling problem is the number of columns in the aligned tables. We prepare the training data for the main sequence-labeling problem and k subsequence-labeling problems, respectively as follows:

- The main sequence-labeling problem: For each input training page, we assign each text node a label corresponding to the column in the master table. Text

nodes that belong to the same subtable are assigned the label of their corresponding record set.
- Sub-sequence labeling problem: For each record set, we only regard text nodes aligned in the same subtable as our training examples. Similar to the main sequence-labeling problem, we assign the same label to the text nodes aligned in the same column.

5.1 Baseline: CRFSuite

We adopted CRFSuite [20] to train the sequence-labeling model and used four coding schemes defined by DCADE, including *TypeSet/PTypeSet*, *ContentId*, *PathId* and *SimSeqId* as the features of each text node to support the alignment process. These encoding schemes are derived from four basic attributes of each text node, including text content, tag path, class sequence and ID sequence (see the blue boxes in Fig. 3).

- $TypeSet/PTypeSet$ Encoding: Every text content is encoded token by token into a type set ($TypeSet$) and its parent type set ($PTypeSet$).
- $ContentId$ Encoding: Two nodes with the same text *Content* is given the same *ContentId*.
- $PathId$ Encoding: Tag paths with similarity higher than a threshold are clustered and assigned the same $PathId$.
- $SimSeqId$ Encoding: Two nodes of similar ID sequence, class sequence, and path are assigned the same $SimSequId$.

Based on the four encoding schemes, DCADE further defined composite codes named content equivalence class (CEC) and typeset equivalence class (TEC), with a total of six levels (see the right table in Fig. 3). Through CRFSuite, we can directly implement these combinations, for example, concatenate two feature values of the current text node to generate a feature similar to $TECID$, or combine the current feature value with the previous or next one. Since a web page may have thousands of leaf nodes after parsing, there may be hundreds of different $ContentID$. This could be a major efficiency issue during training.

5.2 CNN-based Neural Wrappers

While CRFSuite is a well-designed package for sequence-labeling problems, it relies on the predefined encoding schemes given by the DCADE and could not utilize the original attributes, i.e. content, id, class and tag path. Therefore, we have proposed two CNN-based neural networks and include proper embedding layers to accept the text content and tag path input. For tag paths, we allocated a maximum of 30 tag tokens for each text node and applied an embedding layer to generate dense token vectors with a dimension of 5. The 5×30 matrix was then fed as input for a CNN layer with five filters and a window size of 3 to extract features for a tag path. Similarly, the text contents were processed by a separate embedding layer followed by a CNN layer by padding all word sequences to the same length of 50. The output of the two CNN layers are concatenated with other derived features and the text node index before being fed into a fully connected layer (see Fig. 7).

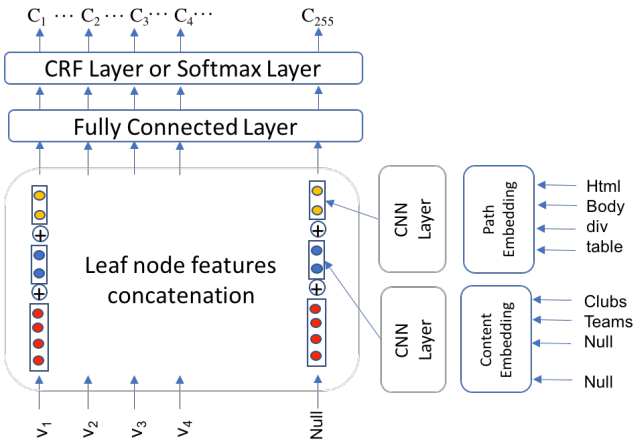

Fig. 7. Illustration of an input page to the neural network model where v_i denotes the features for text node i.

Since the number of output labels is quite large (up to 400 in average), the CNN-CRF model may consume too many memory. Therefore, we have considered two output layers: one is the CRF layer and one is MLP with softmax output. To address the out of memory issue, we divide the input sequence into two segments and consider them as two examples for training.

6 Experiments and Results

To the best of our knowledge, there is no research on the construction of full-page wrappers in the past. Instead, unsupervised full-page data extraction usually ends with data alignment, and no wrappers are generated. Although Roadrunner can generate wrappers, the wrapper may change after aligning with each testing page and the final wrapper also depends on the order of the testing pages. In this paper, we used two datasets to show the benefits of generating wrappers for unsupervised data extraction (see Table 1. The first dataset contains the output results provided by DCADE [9] from Github[4]. The datasets include nine websites used in EXALG [5] and 40 websites from eight categories in TEX [8]. The average number of text nodes for each page is 599 (ranging from 10 to 6,052), which were merged and aligned into 278 columns with a distribution of 49 (17.6%) data columns and 229 (83.4%) template columns for the master table. In addition, each website contains an average of two record sets (ranging from 0 to 10), and each record set contains an average of eight columns (1 template and 7 data columns).

TABLE 1
Dataset Statistics

Datasets	Sites	#Pages	Sets	MaxLN	#Labels
EXALG+TEX	9+40	1,444	2	599	279
SWDE	13	13,145	0	634	405

Because the number of webpages per website for the first dataset is limited (30 pages per website), it is not easy

4. https://github.com/Oviliani/DCADE

TABLE 2
Performance Comparison on EXALG-TEX Dataset

Method	Performance on Master Table			Performance on Record Sets			Total Exe. Time (sec)	
	Precision	Recall	F-1	Precision	Recall	F-1	Training	Testing
DCADE (upper bound)	0.959	0.979	0.961	0.907	0.887	0.876	n/a	1.72
FSM (no Seg)	0.932	0.953	0.935	0.772	0.857	0.788	25	2.46
FSMs (with Seg)	0.936	0.957	0.939	0.760	0.860	0.784	11	1.22
CRFSuite (Baseline)	0.942	0.957	0.942	0.849	0.887	0.846	397	6.38
CNN-CRF	0.940	0.961	0.943	0.765	0.865	0.775	2,188	0.44
CNN-MLP	0.943	0.962	0.945	0.784	0.898	0.808	58	0.64

to show the benefits of wrapper generation. Therefore, we also collected 13 websites (with no record sets) from SWDE (Structured Web Data Extraction) used by Lockard et al. [21]. The dataset contains an average of 1,489 web pages and each page has an average of 634 text nodes. The master table contains a total of 405 columns. For the evaluation, we used the ground truth provided by the DCADE and followed their definition to evaluate the precision, recall and F1 for each column and average them for the master tables and record sets, respectively.

$$\bar{P} = \frac{\sum_{c=1}^{|Ext\ column|} P_c}{|Ext\ column|},\ \bar{R} = \frac{\sum_{c=1}^{|GT\ column|} R_c}{|GT\ column|},$$
$$F = \frac{2 \times \bar{P} \times \bar{R}}{\bar{P} + \bar{R}} \tag{4}$$

$$P_c = \frac{\#correct\ aligned\ nodes\ in\ the\ extracted\ column\ c}{\#nodes\ in\ the\ extracted\ column\ c}$$
$$R_c = \frac{\#correct\ aligned\ nodes\ in\ the\ extracted\ column\ c}{\#nodes\ in\ the\ ground\ truth\ column\ c} \tag{5}$$

In the following experiments, we compared the FSM-based models by setting template segmentation on or off to demonstrate the effectiveness of the template segmentation. Meanwhile, we also compared two CNN based models with the wrapper built from CRFsuite to show the effectiveness of deep neural networks.

6.1 Small Dataset: EXALG+TEX

Since each website in the first dataset has an average of only 30 pages, the performance of the data-driven ML models may not be good if we split them into training and testing sets. Therefore, we use self-testing on this dataset, that is, using the training data as testing data. We use DCADE to prepare the aligned master table and subtables for each website, and show the average performance and total execution time (30 pages) of DCADE in Table 2. Table 2 also shows the performance of the FSM and ML-based wrappers on the master table and record sets, as well as the execution time of training and testing.

In terms of data extraction in the main table, the performance of the FSM or ML wrappers have reached 0.935 to 0.945 F1, which is close to its upper bound, i.e. the performance of DCADE (0.961 F1). As for the extraction performance in subtables, CRFSuite (0.846 F1) is better than other methods (0.775 to 0.808 F1).

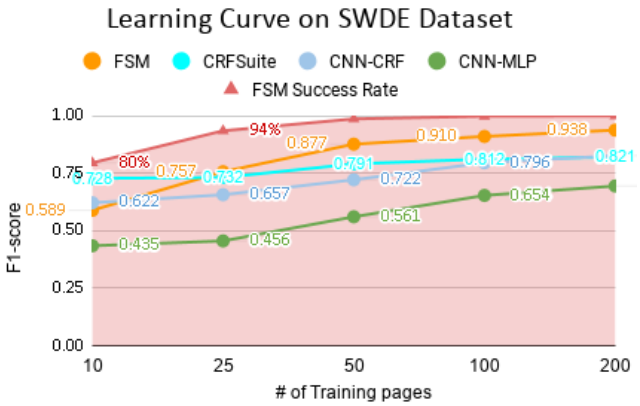

Fig. 8. Learning curve on SWDE dataset. FSM-based approach shows a faster learning curve than three ML-based models.

The biggest difference is the training and testing time. The training time for FSM wrapper construction is much faster than that of ML based wrappers. In terms of testing time, which is our major concern, FSM wrapper with template segmentation takes only 1.22 seconds to extract data from 30 pages, which outperforms the FSM baseline without template segmentation (2.46 seconds). Two CNN-based models, with the help of GPU, are also faster (0.44 or 0.64 seconds) than CRFSuite baseline (6.38 seconds). As we can see, data extraction with the generated wrappers is more efficient than analyzing the testing pages from scratch. Therefore, wrapper generation is necessary. The difference in the second data set is even more obvious.

6.2 SWDE Dataset

For the second dataset, each website has an average of 1000 pages. We randomly selected 10, 25, 50, and 100, and 200 pages for DCADE to conduct unsupervised data extraction and used the output as training data for wrapper generation. The models were tested on the remaining pages. Fig. 8 shows the learning curve of the FSM model with the template segmentation and three ML-based models. The FSM-based wrappers learned much faster than the ML-based wrappers. As we can see, the FSM extraction performance achieves 0.938 F1 with 200 training web pages. However, the performance of CRF-based method is only 0.821F1. Overall, the learning curve of CNN-CRF is better than that of CNN-MLP in this dataset.

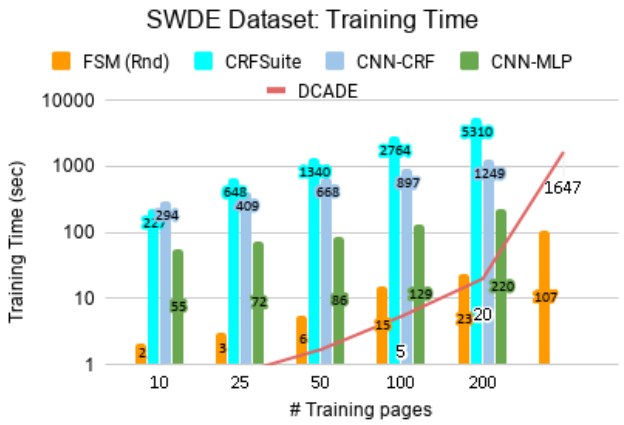

Fig. 9. SWDE dataset: training time.

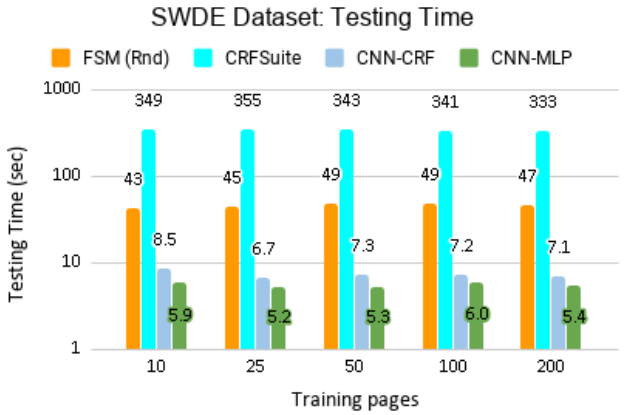

Fig. 10. SWDE dataset: total testing time on remaining pages.

Fig. 9 shows the training time for FSM-based and three ML-based wrapper generation. Since these methods also require DCADE to generate the training data, we also include DCADE running time in Fig. 9. For 100 and 200 training pages, DCADE took 5 and 20 seconds to align the data, respectively. However the time for DCADE to analyze 1,489 training pages drastically increases to 1,647 seconds. The wrapper generation time for FSM and CNN-MLP models remains under hundreds (107 and 220) of seconds. However, the training time for CRF-based methods is much longer (1249 and 5310 seconds).

Fig. 10 shows the testing time for FSM-based and three ML-based wrappers. If we do not generate wrapper and use only DCADE for data extraction, it will cost 1,647 seconds to align all remaining pages. However, with FSM-based wrapper, we only need 43 to 49 seconds (38 milliseconds per page) to extract data from all the testing pages. Besides, CNN-based wrappers only take 7 to 8 seconds (approximately 1 millisecond per page). CRFSuite, although it has good extraction performance, is slower than other wrappers (333 to 355 seconds). In general, we can see the advantages of the wrapper generation over no wrapper generation.

Note that the FSM-based wrappers reject a testing page if an expected template is missing or some template occurs unexpectedly.Therefore, we can compute the success rate to evaluate whether the training examples are comprehensive enough. As shown in Fig. 8, the percentages of testing pages that successfully passed the verification were 80%, 94%, 99%, and 100% given 10, 25, 50, and 100 random training pages, respectively. We will see how to apply this feature to reduce the number of training data with active learning in the next section.

6.3 Active Page Selection

To reduce the number of training pages for wrapper generation, we explored active learning. First, we randomly selected 10 pages for the FSM construction and used the learned model to select pages that that failed the verification process. Next, we added 15 pages to the training set to construct a new FSM-based wrapper and tested it on the remaining pages. The process was repeated until no failed pages were detected. With this training page selection mechanism, the learning curve on the testing data grows rapidly as more training pages are added.

Table 3 compares the wrapper performance of the two training page preparation methods. Obviously, if we select training pages from the failed pages instead of random selection, the generated FSM wrapper will be more comprehensive, thereby increasing the success rate (i.e. the percentage of webpages that can be verified by FSM wrapper). Moreover, the extraction performance of the constructed wrappers will all be improved, no matter which wrapper generation methods is used. For example, when the FSM wrapper is constructed from 10 training pages, the success rate increased from 79.6% to 97.0% and the extraction performance of the CRFSuite wrapper increased from 0.728 to 0.918 F1. When the number of training pages is 50, the extraction performance of the CNN-MLP wrapper can be improved from 0.561 to 0.901 F1, which is quite amazing. Thus, the pages that do not pass the FSM model could provide more radical differences, simulating the so-called active learning paradigm.

Finally, we compared the efficiency of various wrappers. As we can see in the last four columns, the training time for constructing a wrapper from randomly selected training pages is longer than the training time for constructing a wrapper from FSM selected training pages. Meanwhile, the testing time of wrappers constructed from random selection is also longer than the wrapper trained from the pages selected by FSM (38 vs. 33 msec for FSMs, 1.03 vs 0.89 msec for CNN-CRF).

7 Conclusion

The main goal of this work is to improve the efficiency of full-page web data extraction from a large number of testing pages. If no wrapper is constructed, we need to rely on unsupervised data extraction to conduct analysis on testing pages, which takes much longer time than using wrappers. We verified our argument by by comparing the extraction performance with or without wrapper generation through the experiments on 49 small websites and 13 large websites. The contributions of this paper are as follows:

- We clarified the necessity of generating wrappers from unsupervised web data extraction and defined

TABLE 3
Performance Improvement with Selected Training Pages by FSM on SWDE Dataset

Method	10 pages (M10)		25 pages (M25)		50 pages (M50)		M50 Training (sec)		M50 Testing (msec/page)	
	Rand	FSM	Rand	FSM	Rand	FSM	Rand	FSM	Rand	FSM
FSMs (Success Rate)	79.6%	97.0%	93.5%	99.7%	98.7%	100.0%				
FSMs (with Seg)	0.569	0.888	0.751	0.993	0.882	0.997	7.3	6.5	38.7	33.4
CRFSuite	0.728	0.918	0.732	0.966	0.791	0.987	1,340.0	1,209.0	402.0	63.0
CNN-CRF	0.622	0.845	0.657	0.912	0.722	0.948	736.7	646.2	1.03	0.89
CNN-MLP	0.435	0.739	0.456	0.842	0.561	0.901	85.7	48.8	0.74	0.82

the wrapper generation problem of the unsupervised full-page data extraction method.

- We considered two approaches for wrapper generation: (1) schema-guided FSMs, and (2) data-driven ML models. We not only improved the extraction efficiency of FSM-based wrappers through unique and mandatory template segmentation, but also proposed two neural sequence-labeling models to improve the efficiency of CRFSuite.
- We further utilized FSM-based wrapper to select training pages for wrapper generation. Given the same number of training pages, this active learning significantly improve the extraction performance.

For future work, we plan to extend the neural sequence-labeling models to predict whether the input page conforms to the schema, as well as estimate if the wrapper needs to be retrained. We believe that multi-tasking design will enhance the representation of each text node in the input page and improve the performance of neural sequence tagging.

ACKNOWLEDGMENTS

This study is partially supported by Ministry of Science and Technologies, Taiwan under grant MOST-107-2221-E-008-085-MY2.

REFERENCES

[1] C.-N. Hsu and C.-C. Chang, "Finite-state transducers for semi-structured text mining," in *Proceedings of IJCAI-99 Workshop on Text Mining: Foundations, Techniques and Applications*, USA, 1999, pp. 38–49.

[2] I. Muslea, S. Minton, and C. Knoblock, "Stalker: Learning extraction rules for semistructured, web-based information sources," in *Proceedings of AAAI-98 Workshop on AI and Information Integration*. USA: AAAI Press, 1998, pp. 74–81.

[3] C.-H. Chang and S.-C. Lui, "Iepad: information extraction based on pattern discovery," in *Proceedings of the 10th international conference on World Wide Web*. New York: ACM, 2001, pp. 681–688.

[4] B. Liu, R. Grossman, and Y. Zhai, "Mining data records in web pages," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. New York: ACM, 2003, pp. 601–606.

[5] A. Arasu and H. Garcia-Molina, "Extracting structured data from web pages," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York: ACM, 2003, pp. 337–348.

[6] V. Crescenzi and G. Mecca, "Automatic information extraction from large websites," *Journal of the ACM (JACM)*, vol. 51, no. 5, pp. 731–779, Sep. 2004.

[7] M. Kayed and C.-H. Chang, "Fivatech: Page-level web data extraction from template pages," *IEEE transactions on knowledge and data engineering*, vol. 22, no. 2, pp. 249–263, Apr. 2009.

[8] H. A. Sleiman and R. Corchuelo, "Tex: An efficient and effective unsupervised web information extractor," *Knowledge-Based Systems*, vol. 39, pp. 109–123, Feb. 2013.

[9] O. Y. Yuliana and C.-H. Chang, "Dcade: divide and conquer alignment with dynamic encoding for full page data extraction," *Applied Intelligence*, pp. 1–25, Jul. 2019.

[10] N. Kushmerick, "Wrapper verification," *World Wide Web*, vol. 3, no. 2, pp. 79–94, Oct. 2000.

[11] I. F. de Viana, P. J. Abad, J. L. Alvarez, and J. L. Arjona, "Mave: Multilevel wrapper verification system," *IEEE Trans. on Knowl. and Data Eng.*, vol. 28, no. 9, p. 2393–2406, Sep. 2016. [Online]. Available: https://doi.org/10.1109/TKDE.2016.2573302

[12] *Octoparse Help*, 8th ed., Octopus Data Inc., 2021. [Online]. Available: https://helpcenter.octoparse.com/hc/en-us/sections/900000121086-Getting-Started

[13] K. Lerman, S. N. Minton, and C. A. Knoblock, "Wrapper maintenance: A machine learning approach," *Journal of Artificial Intelligence Research*, vol. 18, pp. 149–181, Feb. 2003.

[14] S. Shi, C. Liu, Y. Shen, C. Yuan, and Y. Huang, "Autorm: An effective approach for automatic web data record mining," *Knowledge-Based Systems*, vol. 89, pp. 314–331, Nov. 2015.

[15] A. Omari, B. Kimelfeld, E. Yahav, and S. Shoham, "Lossless separation of web pages into layout code and data," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 1805–1814.

[16] C. Zhao, R. Zhang, and J. Qi, "Web page template and data separation for better maintainability," in *International Conference on Web Information Systems Engineering*, 2018, pp. 439–449.

[17] R. R. Fayzrakhmanov, E. Sallinger, B. Spencer, T. Furche, and G. Gottlob, "Browserless web data extraction: challenges and opportunities," in *Proceedings of the 2018 World Wide Web Conference*. New York: ACM, 2018, pp. 1095–1104.

[18] C.-H. Chang, Y.-L. Lin, K.-C. Lin, and M. Kayed, "Page-level wrapper verification for unsupervised web data extraction," in *International Conference on Web Information Systems Engineering*. Switzerland: Springer, 2013, pp. 454–467.

[19] C.-H. Chang, T.-S. Chen, M.-C. Chen, and J.-L. Ding, "Efficient page-level data extraction via schema induction and verification," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Switzerland: Springer, 2016, pp. 478–490.

[20] N. Okazaki, "Crfsuite: a fast implementation of conditional random fields (crfs)," May 2007. [Online]. Available: http://www.chokkan.org/software/crfsuite/

[21] C. Lockard, P. Shiralkar, and X. L. Dong, "OpenCeres: When open information extraction meets the semi-structured web," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 3047–3056. [Online]. Available: https://aclanthology.org/N19-1309