FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jan Joneš

AI-based Structured Web Data Extraction

Software Engineering

Supervisor of the master thesis:	RNDr. Jakub Klímek, Ph.D.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2022

Dedicated to manually crafted web scrapers used to collect datasets for training automatic systems striving to make them obsolete.

I would like to thank everyone that helped in creating this thesis, especially my supervisor, RNDr. Jakub Klímek, Ph.D., for his help with writing of the thesis, and consultant from Apify, Mgr. Matěj Sochor, for his practical advice and writing scrapers to provide me a whole new dataset. Additionally, I would like to thank Bc. et Bc. Jana Řežábková for her useful deep learning insights and my family for their support.

Title: AI-based Structured Web Data Extraction

Author: Bc. Jan Joneš

Department: Software Engineering

Supervisor: RNDr. Jakub Klímek, Ph.D., Department of Software Engineering

Abstract: In this thesis, we explore current approaches for automatic web data extraction, define their limitations, and aim to overcome them. We propose a deep learning model to extract structured data from graph and visual representations of web pages. The model is evaluated on an older dataset from 2011 which we augment with missing visual assets, and a new dataset consisting of modern websites. It achieves results competitive with recent work and outperforms our baseline based on a state-of-the-art model by at least 10 percentage points on the F1 score. We ensure the implementation is reproducible and provide a demo of extraction from live pages.

Keywords: structured web information extraction, web content mining, web scraping, wrapper generation, artificial intelligence

Contents

Introduction

The web is a vast source of information, ranging from news articles to products sold online. There is a lot of useful underlying structured data such as product prices. Use cases for these data include price comparison sites, analysis of changes (e.g., detecting falsely advertised discounts), creating datasets for machine learning tasks, and businesses pricing their products based on prices of their competitors.

However, it is difficult to obtain these data at scale since many sites are intended to be used and interpreted solely by humans, not machines. Traditionally, developers manually program web scrapers to extract structured data from web pages. This approach requires lots of human labor and each resulting scraper is tailored to only one website. Furthermore, websites often change layout and even small website modifications require manually updating scraper code.

Therefore, it is beneficial to create an automatic web scraper which is able to extract useful information without human intervention. The main benefit arises when scraping a multitude of websites—a matter of simply executing the automatic scraper, but traditionally requiring manual creation of one scraper per website.

Prior research addresses this problem usually by defining a set of target attributes (e.g., product price, name, and description) and training a deep learning model on a set of labeled websites, so the model learns specifics of the attributes and is able to find their values also on unseen websites. Most recent state-of-the-art models work only with page source code, not considering visual characteristics of page elements such as their position on screen, text size, or color.

Furthermore, researchers often neglect many important issues. For example, they fail to evaluate their models equivalently to actual inference (e.g., they filter data by gold labels), use old datasets (although rapid progress in web development dramatically changes how web pages look both on screen and in source code), or insufficiently report all training prerequisites, making their results non-reproducible.

The goal of this thesis is to analyze the problem space and approaches taken by prior work, design a model which overcomes existing limitations and evaluate its performance. We focus mainly on product websites, motivated by the use cases stated above. Specifically, main contributions of this thesis are

- comprehensive overview of related work while precisely distinguishing different sets of problems targeted by prior research,
- design of a deep learning model that uses visual features, is competitive with recent models, and is published as open-source code with reproducible parameters, training environment, and even proof-of-concept live inference demonstration,
- visual extractor, a tool that executes pages in a headless browser to obtain their visual characteristics and can enhance old datasets with visual information by downloading missing assets from an external internet archive,
- re-implementation of a state-of-the-art model which does not use visual features, properly re-evaluating it on a new modern dataset, and comparing its results to our visual-based model.

This thesis is organized as follows. Chapter 1 analyzes the web scraping domain and formally defines the problem framework. Chapter 2 dives into statistical methods used by related work to create automatic web scrapers and formally defines theoretical grounds for a machine learning model solving the stated problem. It also lists limitations of prior work and summarizes the most important related research. Chapter 3 describes available datasets, specifies architecture of the proposed model, and provides experimental results including comparison with related work.

Chapter 1

Web scraping problem analysis

In this chapter, we provide an overview of web scraping, define its terminology, describe the problem space, and provide formal grounds for subsequent chapters.

Existing surveys of the field concentrate on model approaches rather than the problem domain [1, 2], or are not very detailed nor up-to-date considering today's fast evolution of the web [3].

1.1 Overview

Web scraping or *web mining* is the process of extracting information from the web. Using taxonomy by Cooley, Mobasher, and Srivastava [4], it can be further divided to

- *content* mining, i.e., extraction of useful information from web pages,
- *structure* mining, i.e., discovering relationships between pages, and
- *usage* mining, i.e., detecting web usage patterns.

In this thesis, we focus on content mining. In the literature, it is also called "web information extraction" [5] or "web attribute extraction" [6].

An example is presented in Figure 1.1. Web pages are given as input and the goal is to provide structured data as output.

Inputs can have different forms, e.g., source code that browsers use to render the page or a screenshot of the rendered page. Outputs should be in a machine-understandable form, e.g., a set of textual key-value pairs.

Consider an example of a website providing users an online catalog of books. It has a web page presenting information about the book "The Time Machine" by H. G. Wells (partial screenshot of the page is in the first row and second column of Figure 1.1). The desired output of content mining for this web page could be a set of key-value pairs listed in Table 1.1.

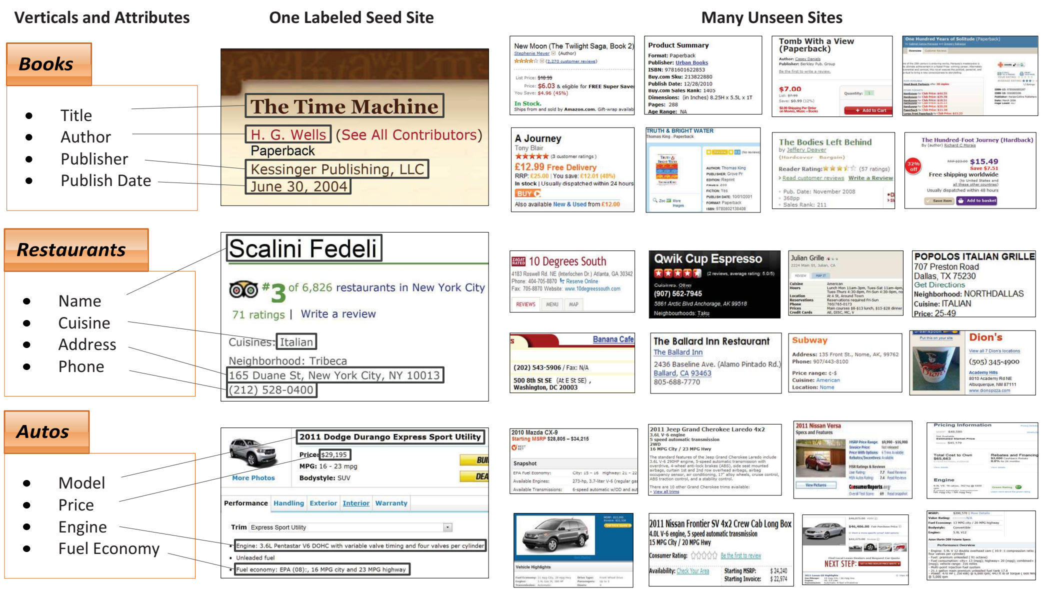


Figure 1.1 Illustration of web content mining as presented by Hao et al. [7]. For each vertical (e.g., books, restaurants, autos), a set of attribute keys is given. The goal is to extract the corresponding value for each key as shown in Table 1.1. The extraction process should work for a range of unseen websites with potentially very different structure.

Key	Value
Title	The Time Machine
Author	H. G. Wells
Publisher	Kessinger Publishing, LLC
Publish Date	June 30, 2004

Table 1.1 A possible output of content mining for a page showing details of a book (corresponding to the top-left corner of Figure 1.1).

1.2 Terminology

Let us introduce web scraping concepts more formally. Different terms are used throughout the literature, so this is our attempt at clarifying the problem domain. When distinct terms are used for the same concept, we provide a few examples of related work each term occurs in.

1.2.1 Abstract model

In this section, we define an abstract model that is used to represent the real world.

Entity An *entity* is one concrete object of interest, e.g., the book presented in the previous section, an e-commerce product, or a movie. It can be called "object" [8], "record" [9], or "entity" [7].

Attribute An entity has a set of *attributes* of interest, e.g., a book has a title, an author, and a publication date. They can be called "fields" [10], "target fields" [11], "attributes" [7], or "labels" [9].

Key and value To be more precise, one must distinguish between attribute *keys* and *values*. Key is the name of an attribute and value is the specific contents for a given entity, e.g., "title" is a key and "The Time Machine" is a value corresponding to that key.

Note that multiple attribute values can correspond to one entity and one attribute key, e.g., a book can have several authors.

Attribute key can be called "semantic label" [9] or just "key" [12]. Often, however, the term "attribute" is used for both keys and values interchangeably [7, 9–11].

Relation Attribute values can be primitive like text or numbers, more complex like images, or they can be references to standalone entities. For example, the author attribute of a book can have just the author's name as a value or the value can somehow identify an entity representing the author. The latter has the advantage of being unambiguous, e.g., in the case where two distinct authors have the same name.

A *relation* is a triple (s, k, o) where s, o are entities and k is an attribute key. Subject s is the entity of interest and object o is its attribute value for the given attribute key k.

1.2.2 Dataset

In this section, we describe how datasets of web pages are usually organized.

Vertical A *vertical* is a category of entities, e.g., books, restaurants, cars, events. It can be called "vertical" [7] or "domain" [9].

Page A *web page* (or simply *page*) [8, 11] is a document accessible over the internet. A *detail* page contains information about one entity. A *list* page contains more entities, sometimes with less information per entity but possibly providing links to the corresponding detail pages.

Template A *template* is a blueprint for dynamically generated web pages, e.g., a set of product detail pages is generated from one template, whereas a set of seller profile pages is generated from a different template.

Website A *website* (or simply *site*) [7] is a collection of interlinked web pages. In the context of information extraction, all pages of a website in a given vertical are considered to be instances of the same template. For example, in the context of the product vertical, the Amazon website is a set of Amazon product detail pages (all conforming to one template).

1.2.3 Document Object Model

In this section, we provide details about the usual representation of a single web page.

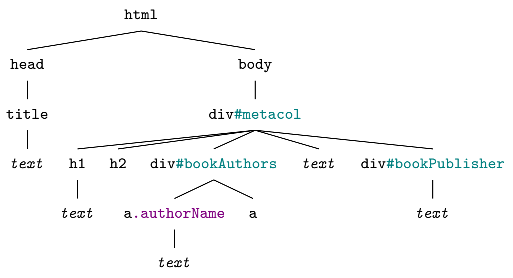
DOM The *Document Object Model (DOM)* [13] is a tree structure parsed from HyperText Markup Language (HTML) source code of a web page by a web browser in order to render the page to the end user. An example of a simplified page HTML source code and the corresponding DOM tree is presented in Figure 1.2.

Note that the DOM tree is initially created from the HTML code (we call this the HTML-induced DOM), but can be then transformed dynamically by JavaScript code defined in the page (either automatically on page load or as a reaction to a user interaction with the page) and augmented with presentation and layout attributes from Cascading Style Sheets (CSS) [14] that further change how the page is rendered.

Node A *node* is an element of the DOM tree as in the standard graph theory terminology. A node has one parent node (except for the root `<html>` node) and an ordered sequence of zero or more children. A node is a *leaf* if and only if (iff) it has no descendants. Otherwise, it is an *inner* node.

```
<html>
<head>
 <title>The Time Machine by H. G. Wells</title>
</head>
<body>
 <div id="metacol" class="last col">
  <h1 id="bookTitle" class="gr-h1 gr-h1--serif">
   The Time Machine
  </h1>
  <h2 id="bookSeries"></h2>
  <div id="bookAuthors">
   <a class="authorName" href="...">H.G. Wells</a>
   <a href="...">(See All Contributors)</a>
  </div>
  Paperback
  <div id="bookPublisher">
   Kessinger Publishing, LLC
  </div>
 </div>
</body>
</html>
```

(a) HTML source code.



(b) Visualization of DOM tree.

Figure 1.2 HTML code of a page and the corresponding DOM tree. This snippet roughly corresponds to the page in the top-left corner of Figure 1.1. HTML code is provided by web servers to web browsers over the network. Web browsers then parse the code to create the DOM tree, a memory representation for manipulation and rendering. For an illustration of correspondence between the DOM and the rendered page, see Figure 1.4.

A node containing an attribute value is called *target node* or labeled node.

Text fragment A *text fragment* is a special kind of leaf DOM node. For example, in HTML `<p>X<br/>Y<br/>Z</p>`, there are three text fragments (`X`, `Y`, and `Z`), which are all children of the inner node `<p>`. It can be called “text field” [5] or “text fragment”[1] [11, 15].

A *text node* is a node that contains some text fragments.

Node properties Each non-text node has an *HTML tag name* (or just *tag name*) that should specify its semantic purpose. The tag name is usually placed in angle brackets, e.g., `<h1>` denotes a top-level heading.

Furthermore, each non-text node can have a set of *HTML attributes* (not to be confused with attributes defined in Section 1.2.1). For example, the first `<div>` node in Figure 1.2a has an attribute `id` with value `metacol`. Some attributes can determine how nodes are rendered in web browsers (e.g., `class` and `style` attributes).

[1]Note that the term “text fragment” has been recently used in the context of HTML for another thing entirely—for specifying a piece of text that a web browser should bring into view. For more details, see `web.dev/text-fragments`.

Whitespace *Whitespace* is any character that is either horizontal or vertical space (including newlines).

Whitespace can be *collapsed*, meaning any consecutive whitespace characters are replaced by a single space. Browsers collapse whitespace before rendering a page unless this behavior is explicitly overridden as defined by the CSS standard [14].

A text fragment is *whitespace* iff it contains only spaces or newline characters. Whitespace text fragments can be generally ignored as they are only byproducts of HTML indentation.

Token A *token* [9, 10] is a word-like unit inside a text fragment. There are multiple ways to split a text fragment into a set of tokens, one possibility is to split by whitespace and punctuation symbols (see Section 2.1.1 for more details).

1.3 Framework

In this section, we sketch the landscape of problems one can attempt to solve in the field of web content mining. Throughout the section, we reference related work tackling each type of problem.

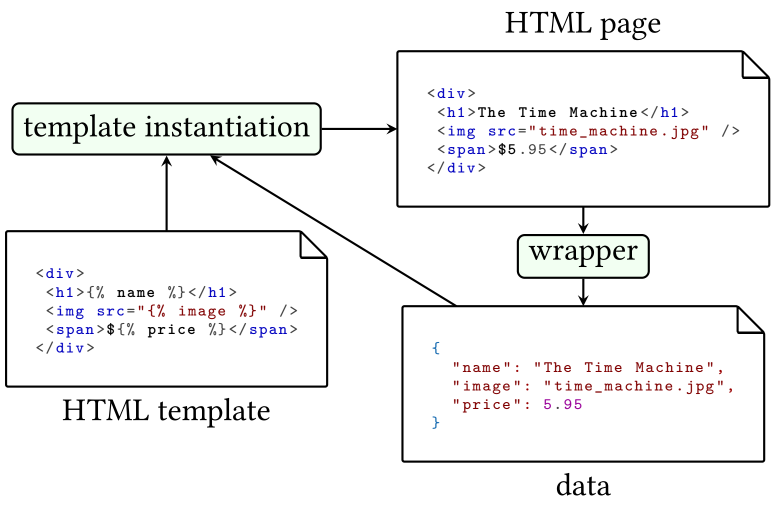
1.3.1 Wrappers

The goal of content mining is to extract useful information from web pages in an automated manner. This is performed via so-called *wrappers* [16] (or “extractors” [17]), transformers from an available semi-structured format (e.g., HTML[2]) into a predefined output format with a useful desired structure.

One wrapper is tailored for one template, i.e., given any page generated by a template, a wrapper of that template extracts structured information contained in the page. In fact, wrappers perform a process inverse to template instantiation (see Figure 1.3):

- Instantiating a template means taking structured data (e.g., a database row representing a product) and producing a format suitable for rendering to end users (e.g., a web page of an e-shop).

- Executing a wrapper means taking the rendered output and trying to reproduce the original structured data.



```
<div>
 <h1>The Time Machine</h1>
 <img src="time_machine.jpg" />
 <span>$5.95</span>
</div>
```

```
<div>
 <h1>{% name %}</h1>
 <img src="{% image %}" />
 <span>${% price %}</span>
</div>
```

```
{
  "name": "The Time Machine",
  "image": "time_machine.jpg",
  "price": 5.95
}
```

Figure 1.3 Wrapper performs an inverse process to template instantiation. Template instantiation (e.g., a web server) takes an HTML template and structured data to produce an HTML page. Wrapper takes the HTML page and attempts to reconstruct the original structured data.

Wrappers are often defined using a set of declarative *rules*. In case of HTML documents, these can be CSS selectors [18] or XPath rules [19]. For each attribute key, a rule should identify precisely the target nodes (i.e., nodes containing the corresponding attribute values) uniquely across pages of the same template.

For example, a wrapper for the page demonstrated in Figure 1.2 could define that a book's title can be found in an `<h1>` node with the HTML attribute `id="bookTitle"` (assuming this holds across other pages of the same template and there are no other nodes with the same characteristics in any DOM of the same template). This would correspond to the CSS selector `h1#bookTitle`.

Wrapper generation

Wrappers can be crafted manually. One can do so by deriving a set of rules after manually inspecting HTML source code of a few pages from the target website. Alternatively, automated tools[3] can aid in this process, e.g., by letting the user merely "point and click" on the target DOM nodes. However, this approach requires a substantial amount of human labor and therefore leads to high costs.

Wrapper induction [7, 16] (or "wrapper learning" [9]) aims to improve on that

[2]Although HTML itself is a fairly structured format, it is designated for rendering by web browsers and subsequent interpretation by humans. Therefore, interesting data are usually buried deep inside and so it is considered semi-structured from the information extraction perspective.

[3]For example, `www.uipath.com` or `www.parsehub.com`.

by automatically generating wrappers from a set of labeled examples. However, it requires manual labeling of sample pages for each new template. Furthermore, whenever templates change, they need to be re-labeled and wrappers regenerated. From the machine learning perspective, this is the easiest problem and it is solved by many [20–23].

Wrapper adaptation attempts to adapt an existing wrapper to another website [9]. *Wrapper maintenance* takes care of updating wrappers when templates change. Kayed and Shaalan [21] also include a wrapper verification component that checks whether existing wrappers need updating.

1.3.2 Input source

In this thesis, we consider information extraction mainly from web pages. However, it might be inspiring to also study other sources of wrapper inputs.

E-mails Gupta, Kondapally, and Guha [24] deal with content mining from e-mail correspondence. They consider e-mails generated by a user filling a template. Since e-mails can also contain HTML code, the input format is fairly similar to web pages.

Documents One large area of research is document understanding [12, 25]. The documents can be e.g., forms, receipts, or invoices. The inputs are scanned documents (i.e., images). Therefore, prior to the extraction itself, optical character recognition (OCR) is usually employed, converting the scans to text. However, there are even end-to-end systems that avoid OCR [26].

Another area of research deals with recovering structured data from tables [27]. Inputs can be plain text, scans, or also HTML. However, tables have a fairly regular structure, making the problem of table recognition vastly different from web information extraction.

Plain text and images Attribute Value Extraction is a task of identifying product attributes given a plain-text product description [28, 29]. Multimodal Attribute Extraction includes also product images as inputs [30, 31]. On the other hand, Image Attribute Extraction takes only product images as inputs and extracts information from that (e.g., color of clothes) [32]. Although the input data are usually extracted from web pages, the extraction process itself is not in the scope of this area of research.

Even if we focus solely on web sources for content mining, several page variations need to be considered.

Granularity Most of the literature deals only with detail pages. However, Raza and Gulwani [22] extract information also from list pages.

Temporality Pages change over time, ranging from small content changes (like price updates) to complete overhauls of website layouts. The latter is challenging in the context of wrapper generation as the defining rules are usually layout-dependent. Omari, Shoham, and Yahav [17] introduce wrappers that can handle structural changes.

1.3.3 Input form

Common information extraction source is a detail web page. However, it remains to define the exact form of the page.

HTML Recall from Section 1.2.3 that each page is defined by HTML source code which induces its DOM tree. Therefore, DOM tree is a natural input for wrapper generation.[4] Often, the HTML-induced DOM is also the only input, even in state-of-the-art systems [6, 10].

Visuals For page rendering in web browsers, HTML source code is not the only ingredient. Layout and visual characteristics (e.g., font size and color of text nodes) can be determined by CSS, usually defined in separate files that the web browser must first download (see Figure 1.4). Number of previous researchers use visual inputs [5, 7, 21, 33]. We call these inputs *visuals*.

Furthermore, JavaScript code that ships with the page can modify the DOM tree before the page is rendered to the user. Applications using *only* JavaScript to render their content—e.g., single-page applications (SPAs)—are becoming abundant in the modern web,[5] although there seems to be no prior research evaluating JavaScript before automatically extracting information from the DOM (existing such systems require manually-defined rules [34, 35]).

Evaluating how a page will render given its DOM is a complicated process even if all the required CSS and JavaScript assets are available, and hence determining visuals practically requires rendering the page in a headless browser,[6] making page processing computationally intensive. This is also the reason why even recent research often uses only the HTML-induced DOM [6, 10].

[4]Some approaches discard the DOM structure and work merely with plain text fragments as is discussed in Chapter 2. However, note that their actual input is still the whole DOM tree.

[5]According to a survey by Statista.com, the most used web framework as of 2021 was React.js, a framework primarily intended to create SPAs.

[6]Headless browsers are regular browsers (after all, the point is to render the page as any user would see it), only they are controlled by automated scripts rather than human users.

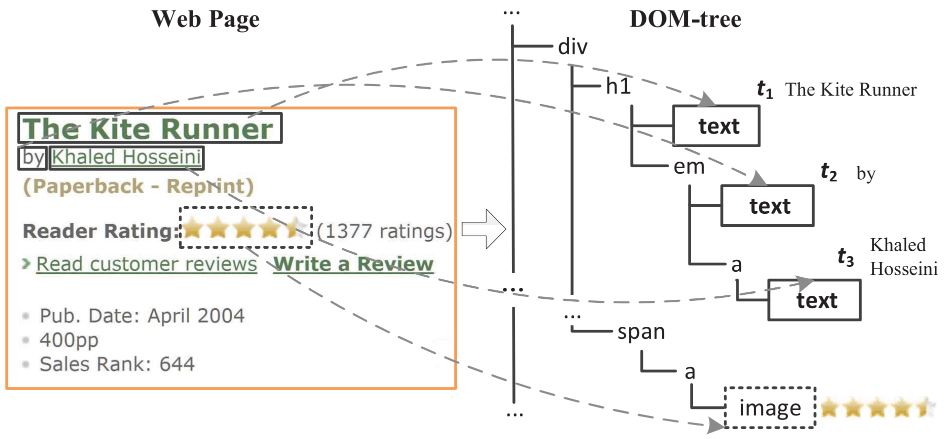


Figure 1.4 Illustration of node visual characteristics as presented by Hao et al. [7]. For example, the nodes t_1, t_2, t_3 have a visual size and position, but also a font size and color. Visuals are not available in the HTML-induced DOM, instead they are loaded by web browsers during rendering, usually from external CSS assets.

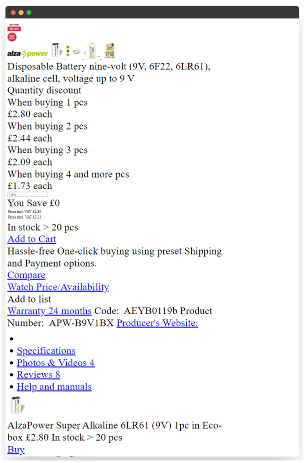
However, visuals arguably add useful information, as illustrated in Figure 1.5. This is especially the case for modern websites as they rely a lot on CSS for layout (and JavaScript in case of SPAs), unlike older websites that heavily used HTML tables for the same purpose [3, 36]. Unfortunately, most recent research is evaluated against old datasets, hence it does not require any visuals to perform well on these static pages [5, 6, 11].

Screenshot Instead of considering specific visual characteristics, one could take a screenshot of the whole page after it has been rendered in the headless browser and use that as an input. This is similar to the document understanding task described in Section 1.3.2.

In web documents, however, there is the additional benefit of having text and other structural information that can be used reliably alongside the screenshot without performing OCR [37, 38].

Screenshots have been also used to determine a web page type in order to select the proper extractor [39].

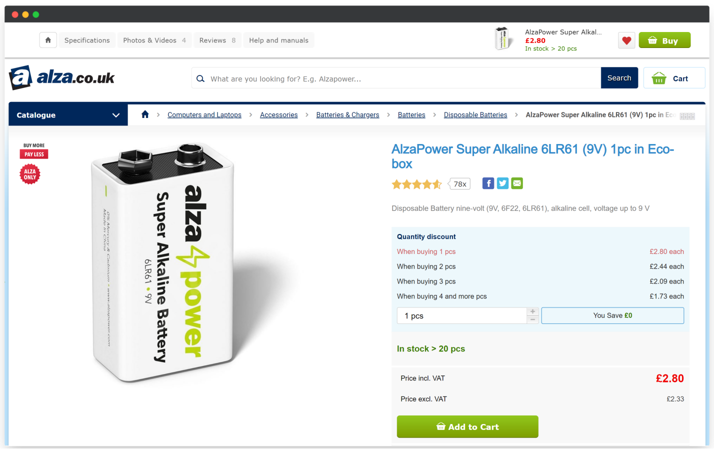
APIs Pages rendered via JavaScript commonly rely on a backend server exposing an application programming interface (API) providing raw data that are then turned into a DOM tree by the JavaScript client-side code. Therefore, access to the API would provide structured data, alleviating the need to extract them.



BUY MORE PAY LESS
ALZA ONLY
alza power
Disposable Battery nine-volt (9V, 6F22, 6LR61), alkaline cell, voltage up to 9 V
Quantity discount
When buying 1 pcs
£2.80 each
When buying 2 pcs
£2.44 each
When buying 3 pcs
£2.09 each
When buying 4 and more pcs
£1.73 each
1 pcs
You Save £0
Price incl. VAT £2.80
Price excl. VAT £2.33
In stock > 20 pcs
Add to Cart
Hassle-free One-click buying using preset Shipping and Payment options.
Compare
Watch Price/Availability
Add to list
Warranty 24 months Code: AEYB0119b Product Number: APW-B9V1BX Producer's Website:

-
- Specifications
- Photos & Videos 4
- Reviews 8
- Help and manuals

AlzaPower Super Alkaline 6LR61 (9V) 1pc in Eco-box £2.80 In stock > 20 pcs
Buy



Specifications | Photos & Videos 4 | Reviews 8 | Help and manuals
AlzaPower Super Alkal...
£2.80
In stock > 20 pcs
Buy
alza.co.uk
What are you looking for? E.g. Alzapower...
Search
Cart
Catalogue
Computers and Laptops > Accessories > Batteries & Chargers > Batteries > Disposable Batteries > AlzaPower Super Alkaline 6LR61 (9V) 1pc in Ec
BUY MORE PAY LESS
ALZA ONLY
AlzaPower Super Alkaline 6LR61 (9V) 1pc in Eco-box
78x
Disposable Battery nine-volt (9V, 6F22, 6LR61), alkaline cell, voltage up to 9 V

Quantity discount	
When buying 1 pcs	£2.80 each
When buying 2 pcs	£2.44 each
When buying 3 pcs	£2.09 each
When buying 4 and more pcs	£1.73 each

1 pcs
You Save £0
In stock > 20 pcs
Price incl. VAT £2.80
Price excl. VAT £2.33
Add to Cart

(a) Without CSS assets.

(b) Full rendering, including CSS assets.

Figure 1.5 The same page with and without CSS.[7] This example illustrates how CSS styles can impact page rendering. Note that the same applies to SPA pages w.r.t. JavaScript (in fact, SPAs might not even render any content unless JavaScript is executed). We can see that it is difficult to extract information from the page (a) even for humans, therefore we would expect automated extraction to benefit from visuals similarly.

There has been an attempt to query available APIs alongside HTML tables [40], although it does not deal with extraction from new websites. Another research intercepts JavaScript Object Notation (JSON) data sent by backend servers while pages are being rendered, although it requires the user to manually define extraction rules [34]. A system automatically inferring JSON schema also exists, however it requires the JSON to be embedded in the page [21].

While this could be a viable approach for some modern websites, it is not universal across the web as it fails for traditional server-rendered websites. Furthermore, not even all modern websites are SPAs due to numerous inefficiencies [41].

Microdata Some pages contain structured information in the form of microdata annotations[8] [42]. While extracting data from these annotations is possible [43], this approach does not generalize to all websites. Many websites do not contain these annotations and others miss many possible annotations, thus leaving some desired information only available in an unstructured form [3].

However, some researchers use microdata annotations to prepare labeled datasets for supervised machine learning [10, 44]. Usually, they extract the

[7]From `www.alza.co.uk`, archived at `https://archive.ph/7xXoG`.
[8]Some common standards include `schema.org` and `microformats.org`.

annotations from Common Crawl,[9] a public dataset of pages.

1.3.4 Outputs

Let us now introduce variations in possible outputs that can be expected from a web content mining system.

Recall from Section 1.3.3 that a natural input for web content mining is the DOM. The corresponding natural outputs are target nodes of the DOM tree selected for a given attribute key. The target node might not be just a plain-text fragment, but can also be a link to another entity, or an image. For example,

- given attribute key "price", the system would identify a text fragment containing the price, e.g., "$21.00";
- given attribute key "product image", the system would identify an `<img>` node containing the main product image; and
- given attribute key "retailer", the system would identify an `<a>` node linking to the retailer's detail page.

Wrappers identifying these nodes can be defined using sets of declarative rules as described in Section 1.3.1. Another possibility is to synthesize imperative code that can then perform the extraction [22].

Although identifying DOM nodes is a common wrapper method, there is no consensus on the exact set of nodes being admitted as output.

Text values Many researchers consider only text fragments as the desired output [6, 7, 15]. In this case, the output can simply be a list of textual key-value pairs, since there is usually no added value in precisely identifying the target nodes.

Note that it is not possible to extract product images using this approach, nor is it possible to extract links to related entities (as neither are text fragments).

Furthermore, it would be challenging to extract information spanning multiple text fragments. Imagine a website rendering prices as

```
<span>$</span><span>21.00</span>,
```

perhaps to have a slightly larger font size for the number. A text-fragment system would have to identify both fragments separately and possibly even group them together (e.g., in case there could be several target prices on the page in different currencies).

[9] `https://commoncrawl.org/`

Text spans Another approach is to produce text spans as outputs. A text span is a continuous range of text which can start and end at any character, even across nodes. This approach can naturally handle not only the above-mentioned output but also text fragments containing several (or parts of) attribute values.

Some researchers deal specifically with this problem, e.g., after selecting a target text fragment, they attempt to further select only interesting parts of text inside [44].

As already mentioned, some methods strip all DOM structure and keep only text fragments. Some of these then naturally produce unconstrained text spans as outputs [10]. Similarly, attribute extraction tasks (see Section 1.3.2) produce text spans, although these might not even exist in the input as text if they are extracted from images [32].

Inner nodes The more general problem is to admit any node to be the wrapper's output. This is more difficult because the amount of considered nodes is substantially larger. Although no existing works seem to take this approach, Wang et al. [10] consider all inner nodes in their model (but then still reduce the final identification to text spans).

1.3.5 Dataset space

Systems for automatic wrapper generation based on statistical models (analyzed further in Chapter 2) need a dataset of labeled pages (i.e., pages and the corresponding target nodes) to train on. These systems can then work on unseen pages. In this section, we categorize these systems based on the set of pages they are allowed to see during training and the target set of unseen pages (the former is the set of pages that is used to generate wrappers for the latter set of pages).

Full supervision As already mentioned, the traditional way of generating wrappers is to train a model on a labeled set of pages that come from the same website as the target set of pages. This "full supervision" approach corresponds to wrapper induction described in Section 1.3.1, it has been extensively studied, and models usually reach over 0.9 F1 score [20–22].

Unsupervised Other researchers propose "unsupervised" models which do not require labeled pages, instead they deduce templates by analyzing co-occurrence of tokens across pages [45–48].

Out-of-domain knowledge More complicated problem is training on pages of a different website than the target website, although the training pages can be from the same vertical as the target pages. If there are training pages also from other verticals, these models are said to use out-of-domain knowledge [5].

Intra-vertical However, most recent research deals with the problem where the training pages are only from the target vertical [5–7, 9, 33]. It is also known as "wrapper adaptation" [15] or "template-independent" extraction [44]. The learning is called k-shot if the training set contains k websites [6].

Cross-vertical Some researchers train their models on a set of pages from a vertical different from the target vertical, although they also add a small amount of pages from the target vertical to the training set (for model fine-tuning). They call it k-shot learning when they train on all websites from vertical v_1 and also k websites from $v_2 \neq v_1$, and test on unseen websites from v_2 [6].

Unseen vertical Lockard et al. [5] attempt to train a model only on websites from a different vertical than the target vertical. This is the most difficult problem since one must also deal with different sets of attribute keys as we discuss in the next section.

When generated, most wrappers can function with just one page as their input. However, some wrappers need a set of more pages as their input [21]. This set must be guaranteed to contain only pages from the same template. Commonly, multiple pages are needed by statistics-based approaches, e.g., those that count node identities [6] or words [7, 49] to determine variable and fixed fragments across pages of one website (see Section 2.1.1).

1.3.6 Label space

In the previous section, we have described possible sets of labeled training pages. In this section, we provide more details about the possible labels.

Predefined keys The most straightforward way is to predefine a set of attribute keys that are then found in each page by the learned model. The set of attribute keys is different for each vertical, e.g., for a product vertical, the set could be "name", "price", "category", but for a restaurant vertical, the set could be "name", "phone", "cuisine".

Most researchers consider only this possibility. Usually, a different model is trained for each vertical (and hence each distinct attribute key set) [6, 7]. Sometimes, a completely different model is trained for each attribute key [28].

Commonly, given one page, one attribute key can match several attribute values, but also zero (e.g., a sold-out product might not have a price).

Zero-shot keys Some researchers also train models capable of recognizing unseen attribute keys (also called zero-shot keys as they are contained in zero training samples). This can be useful to generate wrappers for unseen verticals or to have a richer set of attributes, possibly different even among pages of the same website.

One possibility is to view the labels as relations (or "semantic triples"), i.e., subject, predicate and object. For example, a book (the subject) is written (the predicate) by an author (the object). Commonly, the subject is considered fixed, i.e., it is the main entity of the detail page. Then, only the predicate and the object are labeled in the page. Note that the predicate corresponds to a (dynamic) attribute key and the object to an attribute value.

This is called Open information extraction (OpenIE) by Lockard et al. [5]. In contrast, if the predicate is fixed, it is called Closed information extraction (ClosedIE) which corresponds to the task with a predefined set of attribute keys.

Other researchers view this simply as "new attribute discovery" [9]. Zero-shot keys are also often handled in the related field of document understanding [12] and extracting attributes from plain-text product descriptions [29].

Some models require the unseen attribute keys to be present on the page (e.g., there is the text "price" somewhere in front of the price itself), whereas others can cope with completely missing keys, i.e., pages where only attribute values are present. The former models can discover new keys [5], but the latter must be given the desired keys beforehand [12] or can find attribute values for keys provided dynamically at runtime [10, 29].

Predefined values Note that it is also possible to have just one attribute key and a few predefined attribute values.

For example,

- *noise classification* distinguishes informative nodes from nodes containing noise (e.g., ads or navigation elements) [39, 50] and
- *page classification* categorizes whole pages [51, 52].

In recent works, only Lockard et al. [5] attempted to classify the problem space. They defined three levels, combining dataset space and label space defined above:

L1, corresponding to unseen vertical, zero-shot keys,

L2, corresponding to intra-vertical, predefined keys, and

L3, corresponding to full supervision, predefined keys.

1.4 Task definition

To sum up, the area of automatic web content mining is very diverse, and the exact problem formulations presented by researchers can vary a lot. In previous sections, we have untangled the problem space. In this section, we define the exact problem we deal with in subsequent chapters.

1.4.1 Problem statement

We create a system capable of generating wrappers for detail pages from unseen websites. We focus on the product vertical since it is the only vertical our modern dataset consists of. This corresponds to the intra-vertical system as defined in Section 1.3.5. However, we create a model that should be generic enough to be transferable across verticals, and we evaluate it also against other verticals from an older existing dataset which is also used by most of recent related work. Both of these datasets use one predefined set of attribute keys per vertical. For more details about the datasets, see Section 3.1.

As already mentioned in Section 1.3.3, modern web pages depend on CSS styling and often also JavaScript client-side code for the final rendering. Therefore, to support the modern web, we consider not only the HTML-induced DOM, but also additional visuals like layout and appearance. In order to extract these inputs, we need to render the pages in a headless browser.

Our outputs should be generic enough to encompass both textual values (like price) and more complicated non-textual nodes (like product images). Unlike previous work, we therefore cannot limit ourselves to identifying only text fragments.

However, we feel that supporting arbitrary substrings (i.e., text spans as defined in Section 1.3.4) would bring diminishing returns. From our experience, useful information is contained within whole nodes and also manually created scrapers identify only whole DOM nodes.

Therefore, our wrappers can identify both text fragments and non-textual DOM nodes, although they cannot identify partial substrings inside text fragments.

1.4.2 Formalization

More formally, let $\mathcal{V}$ be a set of verticals, $\mathcal{W}$ a set of websites, $\mathcal{P}$ a set of pages, $\mathcal{N}$ a set of DOM nodes, and $\mathcal{A}$ a set of attribute keys.

For each vertical $v \in \mathcal{V}$, we have a set of pages $\mathcal{P}_v \subseteq \mathcal{P}$. Each page $p \in \mathcal{P}_v$ is defined as a directed tree of DOM nodes (a directed graph without cycles)

$$p \overset{\text{def}}{=} (V_p, E_p),$$

where $V_p \subseteq \mathcal{N}$ are DOM nodes (graph vertices) and $E_p \subseteq V_p \times V_p$ are parent-child relationships between nodes (graph edges).

For each vertical $v \in \mathcal{V}$, we also have a fixed set of attribute keys $\mathcal{A}_v \subseteq \mathcal{A}$. Given a subset of pages $\mathcal{D}_v \subseteq \mathcal{P}_v$, we say it is labeled if we have available a label mapping $\sigma : \mathcal{D}_v \times \mathcal{A}_v \mapsto 2^{\mathcal{N}}$ identifying the target nodes. Note that this mapping is created manually by annotators, and we also assume existence of the true mapping $\sigma_v : \mathcal{P}_v \times \mathcal{A}_v \mapsto 2^{\mathcal{N}}$ identifying the target nodes even for unlabeled pages.

Our goal is to create a wrapper model

$$f : \mathcal{P} \times \mathcal{A} \times \boldsymbol{\Theta} \mapsto \mathcal{N} \cup \{\emptyset\},$$

where $\boldsymbol{\Theta}$ is a set of possible parameter vectors and the predicted nodes belong to the input page, i.e.,

$$\forall v \in \mathcal{V}, \forall p \in \mathcal{P}_v, a \in \mathcal{A}_v, \boldsymbol{\theta} \in \boldsymbol{\Theta} : \ f(p, a, \boldsymbol{\theta}) \in V_p.$$

Note that the model is allowed to predict $\emptyset$ to indicate there is no target node for the given attribute. Ideally, for each vertical $v \in \mathcal{V}$ we want to find such parameters $\boldsymbol{\theta}_v \in \boldsymbol{\Theta}$ that the wrapper finds one of the true target nodes, i.e.,

$$\forall p \in \mathcal{P}_v, a \in \mathcal{A}_v : \ f(p, a, \boldsymbol{\theta}_v) \in \sigma_v(p, a).$$

To measure performance of our models, we use a page-level F1 score that is commonly reported in these cases and hence we can use it also for comparison with related work [6, 7, 11]. For any given attribute key, the page-level F1 score considers any page a "hit" if at least one target node is the model's prediction (if the model predicts multiple nodes, we consider the top-1 prediction which is consistent with prior work). Therefore, we want to minimize the number of false positives and false negatives. For this reason, F1 score is a better choice than for example accuracy.

First, for each attribute key $a \in \mathcal{A}_v$, we define true positives tp_a as the number of pages where the predicted node is labeled

$$\mathrm{tp}_a(\mathcal{D}, \sigma, \boldsymbol{\theta}) \overset{\text{def}}{=} \sum_{p \in \mathcal{D}} \mathbb{I}(f(p, a, \boldsymbol{\theta}) \in \sigma(p, a)),$$

where $\mathcal{D} \subseteq \mathcal{P}_v$ is the evaluated set of pages, and $\mathbb{I}$ is the binary indicator function, returning 1 iff its parameter is true and 0 otherwise. Similarly, we define false positives fp_a as the number of pages where the predicted node is not labeled

$$\mathrm{fp}_a(\mathcal{D}, \sigma, \boldsymbol{\theta}) \overset{\text{def}}{=} \sum_{p \in \mathcal{D}} \mathbb{I}(f(p, a, \boldsymbol{\theta}) \neq \emptyset \wedge f(p, a, \boldsymbol{\theta}) \notin \sigma(p, a)).$$

Lastly, we define false negatives fn_a, as the number of pages where no node is predicted although some are labeled

$$\mathrm{fn}_a(\mathcal{D}, \sigma, \boldsymbol{\theta}) \overset{\text{def}}{=} \sum_{p \in \mathcal{D}} \mathbb{I}(f(p, a, \boldsymbol{\theta}) = \emptyset \wedge \sigma(p, a) \neq \emptyset).$$

From this, an attribute-level F1 score is defined as the harmonic mean of precision or recall, or alternatively

$$F_1^a(\cdot) \overset{\text{def}}{=} \frac{\mathrm{tp}_a(\cdot)}{\mathrm{tp}_a(\cdot) + \frac{1}{2}(\mathrm{fp}_a(\cdot) + \mathrm{fn}_a(\cdot))},$$

and the page-level F1 score is the mean across all attribute keys

$$F_1^v(\cdot) \overset{\text{def}}{=} \frac{1}{|\mathcal{A}_v|} \sum_{a \in \mathcal{A}_v} F_1^a(\cdot),$$

where $(\cdot)$ denotes parameters $(\mathcal{D}, \sigma, \boldsymbol{\theta})$.

Selecting the best model is done by finding parameters $\hat{\boldsymbol{\theta}}_v$ such that the evaluation metric on validation dataset $\mathcal{D}_v^{\text{val}} \subseteq \mathcal{P}_v$ is maximized, i.e.,

$$\hat{\boldsymbol{\theta}}_v = \operatorname*{argmax}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} F_1^v(\mathcal{D}_v^{\text{val}}, \sigma, \boldsymbol{\theta}).$$

This should help in ensuring that the model generalizes well on unseen data, i.e., that it does not overfit to the training dataset [53]. We can further validate this hypothesis by measuring the evaluation metric also on a hold-out test dataset $\mathcal{D}_v^{\text{test}} \subseteq \mathcal{P}_v$ which is not used during model selection.

Note that the set of all labeled pages $\mathcal{D}_v$ is a disjoint union of all the subsets

$$\mathcal{P}_v \supseteq \mathcal{D}_v = \mathcal{D}_v^{\text{train}} \sqcup \mathcal{D}_v^{\text{val}} \sqcup \mathcal{D}_v^{\text{test}}.$$

Since we want to train an intra-vertical model, websites in each subset must be also distinct, i.e.,

$$W(\mathcal{D}_v) = W(\mathcal{D}_v^{\text{train}}) \sqcup W(\mathcal{D}_v^{\text{val}}) \sqcup W(\mathcal{D}_v^{\text{test}}),$$

where $W : 2^{\mathcal{P}} \mapsto 2^{\mathcal{W}}$ maps datasets to website sets

$$W(\mathcal{D}) = \{\, w(p) \mid p \in \mathcal{D} \,\},$$

and $w : \mathcal{P} \mapsto \mathcal{W}$ maps pages to their websites.

Chapter 2

Wrapper generation via statistics

In this chapter, we look into statistical methods that can be used to automatically generate web wrappers. This will serve as a basis when we design our own approach in the next chapter. At the end of this chapter, we also summarize limitations of prior work.

2.1 Feature extraction

Let us start with transformations that turn input data into features processable by statistical methods, e.g., machine learning. Note that some of these transformations might include learnable weights themselves. Nevertheless, we call them feature extraction.

Feature extraction can happen on different levels of data granularity, i.e.,

- individual DOM nodes,
- multiple nodes of a page, and
- multiple pages of a website.

Independently, features can be extracted from different sources, e.g.,

- text,
- DOM (i.e., tag names and attributes), and
- visuals, specifically
  - layout (i.e., position on the rendered page) and
  - presentation (e.g., font size or color).

Sample text	Words	Tokens
`$2,750.00`	[$2]-[,]-[750]-[.]-[00]	[$]-[2]-[,]-[750]-[.]-[00]
`MSRP Price:`	[msrp]-[price]	[msrp]-[price]-[:]

Table 2.1 Tokenization of sample texts. There are two hypothetical tokenizers, their results are shown in the second and the third column, respectively. Notice the subtle but important difference in resulting sets of tokens. For example, token $ (dollar) is separate in the third column which can help the model in recognizing a price. Similarly, separate token `:` (colon) at the end of a text fragment can help in recognizing a label.

2.1.1 Textual features

One of the most important feature extraction sources is text. The goal of textual features is to characterize semantics of text fragments. Features extracted from text are also called "content features" [7, 9, 15] or "language-dependent features" [33].

Character level Useful features can be extracted as low as the character level. For example, simply counting special characters like $ can help in detecting prices.

Hao et al. [7] take this approach when they count three character categories in each text fragment, namely letters, digits, and symbols (except whitespace). They also include length of the whole text fragment, hypothesizing it could be useful to detect special identifiers like ISBN (which always consists of either 10 or 13 digits). Potvin and Villemaire [33] count even whitespace characters, line breaks, and currency symbols.

Another possibility is to take a deep learning approach and embed characters similarly as words are usually embedded (more details about embeddings are provided below). Zhou et al. [6] transform character embeddings via a convolutional neural network (CNN) to compute the final character-level feature.

Word level Commonly, features are extracted from whole tokens (continuous sequences of characters, usually non-whitespace). The first step is tokenizing text fragments into sets of tokens. For example, splitting on whitespace and punctuation gives a list of words; another possibility is to split on more symbols to obtain more tokens as demonstrated in Table 2.1.

To semantically represent the textual tokens, they can be embedded into a space suitable for processing by models working with numerical vectors (e.g., deep neural networks). These embeddings are often precomputed using simple feed-forward neural networks that look at co-occurrences of words in large corpora of texts [54]. Given a dictionary of n words, an *embedding* is a matrix $\mathbb{R}^{n \times d}$ mapping each of the n words into a d-dimensional vector from $\mathbb{R}^d$. Embeddings can also be treated as (pre-initialized) weights and trained alongside a larger neural network in an end-to-end fashion.

Zhou et al. [6] use pre-trained GloVe embeddings [55]. Others train their own embeddings as part of a larger model [15] or even use hashing to assign word indices from a narrow range [37].

Another possibility is to use classical natural language processing (NLP) tools. For example, named entity recognition (NER) can detect proper names and other data types of each token, e.g., dates, zip codes, or identifiers like ISBN and URL [11, 44, 56].

Node level Since text fragments contain varying amounts of tokens, it would be impractical to work directly with token features. Instead, they are aggregated across whole nodes into node-level features. Furthermore, long text fragments are usually truncated, e.g., to 15 tokens [6].

Recurrent neural networks (RNNs) are used by many researchers for this purpose [6, 11, 15]. RNNs are composed of *cells* taking an input $\boldsymbol{x}_i$, a previous state $\boldsymbol{s}_{i-1}$, and producing the next state

$$\boldsymbol{s}_i = f(\boldsymbol{s}_{i-1}, \boldsymbol{x}_i, \boldsymbol{\theta}),$$

where $\boldsymbol{\theta}$ are learnable parameters of the RNN. For a sequence of N tokens, the cell is thus invoked N times.

A simple RNN cell can be just a single-layer feed-forward neural network, i.e., a weighted sum of its inputs composed with an activation function

$$\boldsymbol{s}_i = \tanh(\boldsymbol{U}\boldsymbol{s}_{i-1} + \boldsymbol{V}\boldsymbol{x}_i + \boldsymbol{b}).$$

However, such cells suffer from vanishing or exploding gradients, so more complicated variants are commonly used, e.g., long short-term memory (LSTM) and gated recurrent unit (GRU) cells. These cells have outputs $\boldsymbol{s}_i$ computed separately from hidden states $\boldsymbol{h}_i$.

Generally, an RNN maps the input sequence to another sequence of the same length. The idea is that each vector in the output sequence is enriched with contextual information from the whole sequence. Multiple layers can be stacked such that outputs of one RNN are fed to inputs of the next one. Furthermore, a *bidirectional* RNN can be constructed by feeding the original sequence to one sub-RNN, feeding the reversed sequence to another sub-RNN, and concatenating their outputs into a sequence of length $2N$.

However, we are not interested in mapping sequences to sequences, but rather learning compact representations of whole sequences. To do that, we can ignore all outputs of the RNN except the last one, or aggregate all the outputs via for example the mean function. The latter is the usual approach in related work [6, 11].

Additionally, not all tokens of the input sequence are equally important and hence should not contribute equally to the final representation. A self-attention mechanism [57] can be employed to learn weights $\boldsymbol{\alpha}_i$ from hidden representations $\boldsymbol{u}_i$ and to use them for aggregation into the final sequence vector $\boldsymbol{s}_i$, i.e.,

$$\begin{aligned}\boldsymbol{u}_i &= \tanh(\boldsymbol{W}\boldsymbol{h}_i + \boldsymbol{b})\\ \boldsymbol{\alpha}_i &= \operatorname{softmax}(\boldsymbol{u}_i)\\ \boldsymbol{s}_i &= \sum_j \alpha_{ij} h_{ij},\end{aligned}$$

where the softmax activation function is defined to give probability distribution

$$\operatorname{softmax}(\boldsymbol{x}) \propto e^{\boldsymbol{x}},$$

i.e.,

$$\operatorname{softmax}(\boldsymbol{x})_i \overset{\text{def}}{=} \frac{e^{x_i}}{\sum_j e^{x_j}}.$$

This approach is used by systems that consider only textual features [15, 28].

More recently, the self-attention mechanism has been also used as a basis for neural network architectures called *Transformers*. Pre-trained Transformers are used by some prior work to semantically represent text fragments, most commonly an architecture known as BERT [58]. Lockard et al. [5] average BERT's output over all tokens in a text fragment to get representation of the whole fragment. Zhou et al. [6] experimented with BERT but got worse results than with an LSTM.

Another possibility for encoding semantic information in whole text fragments is to use simple word counting and other statistical techniques known from information retrieval systems. For example, Hao et al. [7] count words and compare the frequencies to distinguish fixed template text (that repeats often) from variable data (with much less repetition). However, to compute these statistics, one usually needs access to the whole corpora of documents [49]. This would most likely not generalize well across different websites, as each template can use a different vocabulary. Therefore, to generate a wrapper for an unseen website, a multitude of its pages have to be the inputs instead of just one as usual (see Section 1.3.5).

2.1.2 DOM features

When extracting information from web pages (and not just plain-text documents) one can also use the added DOM structural information. In this section, we deal only with features that can be extracted from HTML source code (corresponding to the HTML input form defined in Section 1.3.3), and postpone visual features that must be extracted using a headless browser to the next section. DOM features are sometimes called "individual node-level" [49], "template-dependent", or "structural" [33].

Usually, nodes of DOM trees are filtered before feeding them to the model. For example, embedded CSS styles and JavaScript code fragments are removed (after possibly evaluating them in a headless browser), since their containing `<style>` and `<script>` nodes cannot be target nodes. Furthermore, Zhou et al. [6] also remove formatting HTML tags like `<strong>` and `<small>`.

Tag name One semantic feature by definition is the HTML tag name. For example, `<h1>` denotes the main heading of the page and hence it usually contains the name of the target entity. In recent works, tag names are embedded like textual tokens as we have seen in Section 2.1.1, except the fact that tag name embeddings are randomly initialized and trained from scratch [6, 11].

Some works also use specific features like ratio of `<a>` tags (called "link density" or "anchor percentage") [49] or represent tag names as categorical features via the one-hot encoding [33] (where each category $c \in \{1, \dots, C\}$ is encoded as the unit vector $\boldsymbol{e}_c$ [53]).

XPath DOM nodes are connected in a tree. This suggests extracting a feature from the path of tag names of all node's ascendants up to the root node. This path is usually called an XPath since it can be expressed using a simple XPath selector (although the real selector would also have to contain some indices if there were multiple siblings with the same tag name). For example,

```
/html/body/main/div/p/i
```

is an XPath identifying an `<i>` node.

After encoding each tag name in an XPath, a bidirectional LSTM can be used analogously to textual token sequence processing [6, 11].

To represent position, the mere length of the XPath is sometimes used as a feature [33]. Alternatively, a node's depth-first traversal position in the DOM can roughly correspond to node's vertical position on the page [6].

Another option is counting how many times each unique XPath occurs across pages to distinguish *fixed* from *variable* text fragments. Then, target node search can be narrowed to just the set of variable fragments [6]. However, such features need many pages from a website to extract its wrapper (as discussed in Section 1.3.5).

Moreover, variable node detection as defined by its authors [6] is fundamentally flawed, because some target nodes might not be variable by this definition, and thus these would be filtered out and never identified by the model. This stems from the fact that target nodes for one attribute key can be at slightly different positions and hence have different XPaths across pages of the same website. Only by inspecting their source code have we found that the authors actually use gold labels to make sure target nodes are in the set of variable nodes.[1] Of course, this is not possible during inference. Although this filtering could be omitted during inference, their evaluations use it.

2.1.3 Visual features

Considering also CSS and JavaScript assets, the DOM tree might be radically different. CSS adds styling to each node (like color, font size, or layout) and JavaScript can modify or even create the whole DOM tree from scratch. Features extracted from these visual inputs (as defined in Section 1.3.3) are called visual features, "layout" [15], "formatting" [9], "layout-dependent" [33], or "rendered document image level" [49].

Layout One important information unavailable in the HTML-induced DOM tree is the position of each node on the rendered page. It is especially useful when combined with contextual features described in the next section. Together, they can help when detecting target nodes, e.g., if there is a text fragment "Price ($):" next to a text fragment "20.00", the former fragment (also called a *cue* [12]) tells us that the latter is a price of the product, a fact that would be difficult to recognize just from the target text fragment alone.

Although the cue might be close to the target also in the DOM tree, this fact depends on the template implementation. However, when the page is rendered in a web browser, the cue will almost certainly be next to the target in order to be interpretable by the end user. In theory, the two nodes might be at completely unrelated places in the DOM, only brought together on the screen via CSS. In any case, the template implementation can be considered an internal ephemeral detail unlike the visual representation that we hypothesize should be fairly consistent.

Node's position is represented by its *bounding box*, a quadruple consisting of its x, y coordinates, width, and height. The coordinates can be used to detect related nodes [7, 33] or directly as a machine learning feature [37], and the size can be used to detect large sections with the main content [21]. The term bounding box comes from the object detection task [59] where the box (x, y, w, h) is encoded

[1]See function `assure_value_variable` in `extract_xpaths.py` of commit `cdad65e` in the GitHub repository `google-research`.

relatively to a region of interest (which could be the whole page with coordinates $(0, 0, W, H)$ in our case) to get a representation that is more natural for neural network processing:

$$\begin{aligned} t_x &= x/W, & t_y &= y/H, \\ t_w &= \log(w/W), & t_h &= \log(h/H). \end{aligned}$$

Presentation Another group of visual features are *presentational* traits, usually of a text fragment, e.g., font size, family, weight (bold or regular), style (italics or normal), color, alignment (left or center inside its parent). These are mostly categorical features and hence encoded into a one-hot representation. Font size can be represented directly since it is a number [5] and colors usually have each component (red, green, and blue) encoded separately [33].

Beside text fragments, features can be extracted also from images, e.g., a color or texture from a product image of clothes [32]. However, this is usually the focus of attribute extraction tasks, not approaches seeking to identify target nodes (see Section 1.3.4).

Screenshot of the whole page is also an output of page rendering ergo a visual feature (see Section 1.3.3). It is an image and thus usually processed via a CNN [21, 25, 37, 38]. However, since it works with the whole page, it can be also considered as a contextual feature which we discuss next.

2.1.4 Contextual features

So far we have mostly considered individual node-level features. These can be combined to form contextual features. More precisely, using the notation from Section 1.4.2, given a node $n \in V_p$, we denote its individual features $\boldsymbol{x}_n \in \mathbb{R}^m$. Then, we can take its neighborhood $N_n \subseteq V_p$, aggregate its features via a function $a : 2^{\mathbb{R}^m} \mapsto \mathbb{R}^m$ and propagate them for example by concatenating with the individual feature vector into a final feature vector

$$\boldsymbol{x}'_n \overset{\text{def}}{=} [\boldsymbol{x}_n \parallel a(\{\, \boldsymbol{x}_h \mid h \in N_n \,\})],$$

where $[\cdot \parallel \cdot]$ denotes vector concatenation. The precise definitions of neighborhood sets and aggregation functions can differ, and we discuss some possibilities below.

Using only textual input, Liu, Li, and Fan [15] employ an RNN with the attention mechanism on text fragment features from the whole page to propagate contextual information to each node from its surroundings. Analogously, Zhou et al. [6] experiment with using Transformer on node features, although they end up omitting this feature as it worsens their results.

In document understanding (see Section 1.3.2), a model can be made to locate a cue text fragment prior to finding the target value which also helps with zero-shot keys [12].

Some researchers create rules defining the neighbor nodes. For example, Zhou et al. [6] define a set of "friend" nodes plus at most one "partner" node based on their proximity in the DOM tree. They use five friend nodes and concatenate all their features together. Similarly, Hao et al. [7] locate for each fragment its preceding text based on its visual layout proximity.

Graph neural networks

More general approach is using a graph neural network (GNN) to propagate features across edges in a graph of nodes. The problem is analogous—one must define the set of edges and the message-passing function, corresponding to the neighborhood N_n and the aggregation function a, respectively. However, the field of GNNs has been extensively studied, so it might provide us some useful methods [60].

Most GNN methods focus on graphs with high *homophily*, i.e., graphs where linked nodes often belong to the same class or have similar features. For example, in a graph of people, linking friends connects people with similar interests or age. In contrast, DOM-based graphs have high *heterophily*, i.e., linked nodes have dissimilar features. For example, a target price node can consist of digits and a currency symbol, whereas its neighbor node can be the word "Price" and the colon symbol. However, there are methods to help in alleviating the problems, such as encoding node and its neighborhood separately, aggregating information from more distant neighbors, and preserving all intermediate representations [61].

Graph convolution is a common message passing layer. Given an undirected, unweighted graph $G = (V_G, E_G)$ and input features $\boldsymbol{x}_i \in \mathbb{R}^n$ for a node $i \in V_G$, graph convolution computes output features $\boldsymbol{x}'_i \in \mathbb{R}^m$ as the sum over the neighbor features

$$\boldsymbol{x}'_i = \boldsymbol{W}\boldsymbol{x}_i + \sum_{ij \in E_G} \boldsymbol{W}\boldsymbol{x}_j,$$

where $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ is a learnable weight matrix. Note that multiple message passing layers can be stacked, effectively propagating features across longer paths in the graph.

More complex message passing layer is the graph attention (GAT) [62]

$$\boldsymbol{x}'_i = \alpha_{i,i}\boldsymbol{W}\boldsymbol{x}_i + \sum_{ij \in E_G} \alpha_{i,j}\boldsymbol{W}\boldsymbol{x}_j,$$

where $\alpha_{i,j}$ are attention coefficients indicating importance of node j to node i

computed as

$$\alpha_{i,j} = \frac{\exp(\mathrm{LeakyReLU}(\boldsymbol{a}[\boldsymbol{W}\boldsymbol{x}_i \parallel \boldsymbol{W}\boldsymbol{x}_j]))}{\sum_{ik \in E_G \cup \{ii\}} \exp(\mathrm{LeakyReLU}(\boldsymbol{a}[\boldsymbol{W}\boldsymbol{x}_i \parallel \boldsymbol{W}\boldsymbol{x}_k]))},$$

where $\boldsymbol{a}$ are additional learnable weights and

$$\mathrm{LeakyReLU}(\boldsymbol{x}) \overset{\mathrm{def}}{=} \max(0, \boldsymbol{x}) + 0.2 \cdot \min(0, \boldsymbol{x}).$$

Let us present a list of prior researchers using various GNN architectures and their approach to constructing the input graph from DOM.

Lockard et al. [5] use a GAT against a graph consisting of text fragment nodes, horizontal, vertical, and DOM edges. They add horizontal (or vertical) edges between nodes that share horizontal (or vertical) location and there are no other text fragments between them. They add DOM edges between sibling or cousin nodes, i.e., any two nodes that share a parent or a grandparent, respectively. Furthermore, their GAT does not participate in an end-to-end training. Instead, it is pre-trained in a simplified classification task and then used with immutable (also called *frozen*) weights.

Wang et al. [10] use the original HTML-induced DOM tree augmented by sibling edges for a GAT. They train the GAT as part of a bigger model.

In the context of document understanding, Yu et al. [25] use a more complicated GNN consisting of two stages—a graph learning and a graph convolution. In the graph learning stage, attention weights are learned given only a set of nodes, and in the graph convolution stage, features are propagated both from nodes and edges. Specifically, the graph convolution is performed on triples (n_i, e_{ij}, n_j) rather than just nodes, so that edge features are considered, as well. As edge features they select vertical and horizontal distance, aspect ratio of the first node w_i/h_i, width and height of the second node normalized by height of the first node h_j/h_i, w_j/h_i, and sentence length ratio between the two nodes.

2.2 Classification

Having numerical features extracted from semi-structured inputs (DOM) or unstructured inputs (text or images), the next step is to identify the target nodes. Usually, models are designed to classify each extracted node feature vector into a predefined set of classes, i.e., attribute keys in our case plus one class for non-target nodes. This classification of DOM nodes is also known as "node tagging" [11].

However, there are some exceptions, like a model synthesizing program code [22], separate models for each attribute key [28], or a model that also attempts to identify sub-node text spans [44]. This roughly corresponds to output forms defined in Section 1.3.4.

Furthermore, models striving to support zero-shot keys might instead classify pairs of nodes (one for the attribute key and one for the target value) [5] or be given the desired attribute keys as one of the inputs in order to find cues in a separate stage [12] or combine the attribute key with each node via an attention mechanism similar to the one described in Section 2.1.1, also known as *question answering* [10, 29].

Nevertheless, standard node classifiers are most common, including classical methods like support-vector machines [7] and random forests [33], generative Bayesian models [9], and conditional random fields [8]. Recently, deep learning classifiers are often employed, using some number of hidden feed-forward layers and the softmax function as the final activation [6, 15, 33].

Specifically, using the notation from Section 1.4.2, we define the number of target classes $C_v \overset{\text{def}}{=} |\mathcal{A}| + 1$, i.e., one class per each attribute key plus one for non-target nodes. We define a mapping $\gamma_v : \mathcal{A}_v \mapsto C_v$ assigning each attribute key a unique class, leaving class 0 for non-target nodes. Lastly, we restructure the labeling function $\sigma : \mathcal{D}_v \times \mathcal{A}_v \mapsto 2^{\mathcal{N}}$ into $\varphi : \mathcal{D}_v \times \mathcal{N} \mapsto C_v$ defined as

$$\forall p \in \mathcal{D}_v, n \in V_p : \ \varphi(p, n) \overset{\text{def}}{=} \begin{cases} \gamma_v(a) & \text{if } \exists a \in \mathcal{A}_v : \ n \in \sigma(p, a), \\ 0 & \text{otherwise.} \end{cases}$$

Then, the classifier is a function

$$h : \mathbb{R}^m \times \boldsymbol{\Theta} \mapsto \mathbb{R}^{C_v}$$

that given a vertical $v \in \mathcal{V}$, a page $p \in \mathcal{P}_v$, a node $n \in V_p$, and its extracted feature vector $\boldsymbol{x}_n \in \mathbb{R}^m$ produces *logits* that are converted to a probability distribution of the attribute key (label) $y_n \in \{0, \ldots, C_v\}$ of the node via the softmax function

$$p(y_n | \boldsymbol{x}_n, \boldsymbol{\theta}) = \operatorname{softmax}(h(\boldsymbol{x}_n, \boldsymbol{\theta})).$$

The classifier usually consists of a few feed-forward layers, e.g.,

$$\begin{aligned} h_1(\boldsymbol{x}, \boldsymbol{\theta}) &= \operatorname{ReLU}(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) \\ h_2(\boldsymbol{x}, \boldsymbol{\theta}) &= \boldsymbol{W}_2 \boldsymbol{x} + \boldsymbol{b}_2 \\ h(\boldsymbol{x}, \boldsymbol{\theta}) &= h_2(h_1(\boldsymbol{x}, \boldsymbol{\theta}), \boldsymbol{\theta}), \end{aligned}$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{m_1 \times m}$, $\boldsymbol{W}_2 \in \mathbb{R}^{C_v \times m_1}$ are matrices of weights and $\boldsymbol{b}_1 \in \mathbb{R}^{m_1}$, $\boldsymbol{b}_2 \in \mathbb{R}^{C_v}$ are bias weight vectors, all part of the learnable parameter vector $\boldsymbol{\theta}_{\mathbf{t}}$, and the ReLU activation function is defined as

$$\operatorname{ReLU}(\boldsymbol{x}) \overset{\text{def}}{=} \max(0, \boldsymbol{x}).$$

Note that overall parameters $\mathbf{\Theta}$ defined in Section 1.4.2 consist of learnable parameters $\mathbf{\Theta_t}$ that can be trained automatically as described below and hyperparameters $\mathbf{\Theta_h}$ that are chosen manually during model selection, i.e.,

$$\mathbf{\Theta} = \mathbf{\Theta_t} \times \mathbf{\Theta_h}.$$

In fact, we aim to create a universal model that uses the same set of features and layers across verticals, hence we find one $\boldsymbol{\theta}_\mathbf{h} \in \mathbf{\Theta_h}$. However, the model is trained for each vertical $v \in \mathcal{V}$ separately, therefore we find separate $\boldsymbol{\theta}_{\mathbf{t},v} \in \mathbf{\Theta_t}$ for each v. Together,

$$\boldsymbol{\theta}_v \overset{\text{def}}{=} (\boldsymbol{\theta}_\mathbf{h}, \boldsymbol{\theta}_{\mathbf{t},v}).$$

The whole wrapper model f then consists of a feature extraction function $g : \mathcal{P} \times \mathbf{\Theta} \mapsto \mathbb{R}^m$ and the classifier h, predicting the most likely target node for each page $p \in \mathcal{P}_v$ and each attribute key $a \in \mathcal{A}_v$ given parameters $\boldsymbol{\theta}_v \in \mathbf{\Theta}$ learned for a specific vertical $v \in \mathcal{V}$, i.e.,

$$f(p, a, \boldsymbol{\theta}_v) = \begin{cases} n & \text{if } \exists n \in V_p : \ \gamma_v(a) = \operatorname{argmax}_{c \in C_v} p(y_n = c | g(p, \boldsymbol{\theta}_v), \boldsymbol{\theta}_v), \\ \emptyset & \text{otherwise.} \end{cases}$$

During model training, we compute its *misclassification rate* on the training dataset $\mathcal{D}_v^{\text{train}}$ as

$$\mathcal{L}(\mathcal{D}_v^{\text{train}}, \varphi, \boldsymbol{\theta}) \overset{\text{def}}{=} \frac{1}{|\mathcal{D}_v^{\text{train}}|} \sum_{p \in \mathcal{D}_v^{\text{train}}} \frac{1}{|V_p|} \sum_{n \in V_p} \ell(\varphi(p, n), h(g(p, \boldsymbol{\theta}), \boldsymbol{\theta})),$$

where $\ell : C_v \times \mathbb{R}^m \mapsto [0; \infty)$ is a *loss function* measuring difference between predictions and true labels [53]. For multi-class classification, the cross-entropy loss is commonly used [6, 15], defined as

$$\ell(y, \boldsymbol{z}) \overset{\text{def}}{=} -\log(\operatorname{softmax}(\boldsymbol{z})_y),$$

where $\boldsymbol{z}$ is the output of the classifier (a vector of class logits) and the softmax vector is indexed by the true class y in order to obtain its logit.

Given some hyperparameters $\boldsymbol{\theta}_\mathbf{h}$, learnable weights $\boldsymbol{\theta}_\mathbf{t}$ of the model are updated during training with the goal being to minimize the misclassification rate on the training dataset, i.e., to find parameters

$$\hat{\boldsymbol{\theta}}_\mathbf{t} = \underset{\boldsymbol{\theta}_\mathbf{t} \in \mathbf{\Theta_t}}{\operatorname{argmin}} \mathcal{L}(\mathcal{D}_v^{\text{train}}, \varphi, (\boldsymbol{\theta}_\mathbf{t}, \boldsymbol{\theta}_\mathbf{h})).$$

We may use the gradient descent algorithm, updating $\boldsymbol{\theta}_\mathbf{t}$ at each step as

$$\boldsymbol{\theta}_\mathbf{t} \leftarrow \boldsymbol{\theta}_\mathbf{t} - \alpha \nabla_{\boldsymbol{\theta}_\mathbf{t}} \mathcal{L}(\mathcal{D}_v^{\text{train}}, \varphi, (\boldsymbol{\theta}_\mathbf{t}, \boldsymbol{\theta}_\mathbf{h})),$$

where $\alpha \in \mathbb{R}$ is the learning rate and ∇ denotes gradient of the model function over its parameters.

However, evaluating the loss for the whole training dataset at each step would be computationally expensive, hence the computation at each step is restricted to a batch of B random independent samples [53]. Furthermore, more advanced gradient descent algorithms such as Adam that use a dynamic momentum rather than a fixed learning rate are used in prior work [6].

2.3 Existing limitations

Let us now define criteria we set for our wrapper generation system stemming from limitations of existing research and also motivated by the needs of the web scraping company that provided us a new dataset (see Section 3.1.2).

C1 **Modern websites** Our system should handle modern websites that use CSS for layout and SPAs that are rendered via client-side JavaScript code execution. See Section 1.3.3 for more details about this problem.

Unfortunately, most related work uses an old dataset from 2011 [7] that does not contain visual assets. See Section 3.1 for a discussion about datasets.

C2 **Few page extraction** As already defined in Section 1.4, our system should identify target nodes in a page from an unseen website without the need for many pages from the target template. We also demonstrate this capability (see C6 below).

Although it is usually not obvious, previous models often do not work this way. Many use features that would need many pages from the same website during inference time as discussed in Section 1.3.5.

C3 **Generic features** The extracted features should be generic enough to transfer to unseen websites. Especially, they should not be overly complex and over-engineered as such features are arguably not generalizable to many websites. Furthermore, they should apply to more than just text fragments, e.g., also images, since we have defined a generic output in Section 1.4.

As discussed throughout Section 2.1, older works use features that might be vertical-specific, such as the number of dollar symbols in a text fragment. On the other hand, recent work focuses on classifying only text fragments, hence often extracting mainly text-related features.

C4 **Simple model** The model should be relatively simple in terms of trainable parameters. As a result, the model should be trainable in reasonable times without the need for expensive hardware.

Most prior work does not provide details about training times, so this criterion is difficult to judge. However, recent models often have lots of trainable parameters and are trained by large companies with access to powerful hardware.

C5 **Reproducibility** We should train our models in a reproducible way. This means that both the set of parameters of the training code and the set of development environment dependencies (including exact versions) should be published.

Often, source code of prior works is not published and even when it is, executing the code is usually not straightforward since the required software is either not specified at all, or the exact required versions are not obvious, making it challenging to run especially older code that does not work with the latest software libraries out-of-the-box.

C6 **Inference demonstration** Our trained models should be executable in an inference mode. Specifically, we should create a self-contained program (*demo*) that is able to take a live web page input and identify the likely target nodes on demand.

Similarly to C5, this is not common. In fact, we are aware of only one prior work providing a demo [37]. This is useful to demonstrate that the inference can actually be executed on new webpages. For example, Zhou et al. [6] provide source code of their model training pipeline, but all preparation is performed in bulk across the whole dataset, thus it is not clear how actual inference would work.

Below we summarize the most important recent approaches and their limitations. These models are published by top researchers, including research groups from Google, Facebook, and Amazon. More details about them and many other methods have been provided throughout this chapter and the previous one.

- Hierarchical RNN model from 2018 [15]. It works only with text, it is evaluated against an old dataset (violating C1), and it is trained and tested against the same website, i.e., it solves only the full supervision task defined in Section 1.3.5.
- ZeroShotCeres, a model from 2020 [5]. It can handle zero-shot keys (see Section 1.3.6), but it is evaluated only on a subset of the same old dataset.

Although it uses some visual features, they are arguably not very informative since the used dataset contains archived visuals for only few websites as explained in Section 3.1.1. Even in the easier task without zero-shot keys which is similar to ours, it achieves subpar results (see Section 3.4.2).

- SimpDOM [6], a state-of-the-art model at the time we started our research (in 2021). It still uses the old dataset and does not consider visual features at all. Although its source code is available (C5), neither build requirements, nor all training parameters are specified in a reproducible way. Furthermore, they detect variable nodes across many pages in a website (see Section 2.1.2), a feature unavailable during inference against one page (C2).
- WebFormer [10], a state-of-the-art model introduced simultaneously with our thesis (in 2022). It is evaluated on a different problem—all websites from all verticals are present in train and test sets. It is a complex Transformer architecture trained from scratch (apart from pre-trained word embeddings). Although the authors do not provide details about the exact model size and training times, they use very powerful hardware for training (32-core TPU v3). We thus hypothesize it violates C4. Although it extracts features for arbitrary inner nodes unlike previous work (C3), the final classification is restricted to text fragments.

Most other models are evaluated on different problems without any common denominator. Therefore, we compare our work mainly to the contemporary state-of-the-art, SimpDOM, which also targets our use-case, i.e., k-shot intra-vertical learning (see Section 1.3.5).

Chapter 3

Experiments and results

In this chapter, we describe our approach to the problem defined in Section 1.4 using methods presented in Chapter 2. We also provide details about used datasets and technical implementation. Then, we present our results and compare them with related work.

3.1 Datasets

Throughout the literature, different datasets are used, containing websites from several verticals, e.g.,

- products [17, 33, 37, 63, 64],
- books and movies [7, 17],
- events [44, 65],
- news [49, 66],
- personal information [67].

We focus on products (also called *e-commerce* websites). From those product datasets that claim to be available, one was never published [37], another has been apparently lost as we learned trying to obtain it from its authors (as it is fully available only on request) [33], and yet another is hosted as a torrent, but it has no active seeds, so it cannot be downloaded [64].

3.1.1 SWDE

Many researchers use the Structured Web Data Extraction (SWDE) dataset introduced by Hao et al. [7] in 2011. It was freely accessible until recently when its

Vertical	Pages	Attribute keys
Auto	17,923	`model`, `price`, `engine`, `fuel_economy`
Book	20,000	`title`, `author`, `isbn_13`, `publisher`, `publication_date`
Camera	5,258	`model`, `price`, `manufacturer`
Job	20,000	`title`, `company`, `location`, `date_posted`
Movie	20,000	`title`, `director`, `genre`, `mpaa_rating`
NBA Player	4,405	`name`, `team`, `height`, `weight`
Restaurant	20,000	`name`, `address`, `phone`, `cuisine`
University	16,705	`name`, `phone`, `website`, `type`

Table 3.1 SWDE verticals, page counts, and attribute keys. Each vertical consists of 10 websites. Each website consists of 200–2,000 detail pages. More details about one vertical (auto) are provided in Table 3.2.

hosting has been discontinued.[1] As shown in Table 3.1, it consists of 8 verticals, 80 websites, and over 120,000 pages. Three of its verticals (Movie, NBA Player, University) have been also re-annotated for zero-shot attribute key extraction [5].

The SWDE dataset contains for each page its original HTML file and URL. For most pages, this means there are no visuals (as defined in Section 1.3.3), except for some websites that include their CSS inside HTML. We address this problem later in Section 3.3.1.

From the SWDE dataset, we develop our model on the auto vertical since it most resembles the product vertical (for one, it has the important attribute price; although the camera vertical also has that, it contains fewer pages). In Table 3.2, more details about this vertical are provided.

Note that ground-truth attribute values are provided only as plain text. Unfortunately, the mapping from the original HTML text fragments to the plain-text attribute values is not entirely deterministic. Sometimes, special characters are given as encoded HTML entities (e.g., `&nbsp;`), other times, they are present in their decoded Unicode form. Moreover, whitespace is sometimes preserved, and other times collapsed in the plain-text values. Therefore, it is impossible to automatically determine exactly the set of nodes that were labeled by annotators in the original HTML, but by ignoring these inconsistencies we can find a superset of the target nodes (this approach is also taken by others as can be seen from source code of their implementations [6]).

Note that this approach can also label some completely unrelated nodes. For example, if a product costs "$20.00", and there is a list of related products on the page, one of which also costs "$20.00", its price is also labeled, although it should not be a target node.

[1]It can still be accessed via the Internet Archive at `https://web.archive.org/web/20211009004153/https://archive.codeplex.com/?p=swde`.

		price		economy		engine		model	
website	pages	values	nodes	values	nodes	values	nodes	values	nodes
aol	2,000	2,000	2,001	1,845	1,845	0	0	2,000	2,009
autobytel	2,000	1,994	1,994	1,816	1,816	2,000	2,000	3,998	4,023
automotive	1,999	1,999	1,999	1,999	1,999	1,999	1,999	1,999	1,999
autoweb	2,000	2,000	2,001	4,000	4,000	1,998	1,998	4,000	4,000
carquotes	2,000	2,000	2,000	2,000	2,000	2,000	2,000	2,000	2,000
cars	657	647	741	607	607	1,168	1,168	657	683
kbb	2,000	2,000	4,732	2,000	2,000	2,000	2,938	4,000	4,000
motortrend	1,267	1,267	1,267	2,534	2,534	1,267	1,267	1,267	1,267
msn	2,000	3,901	4,007	3,623	4,081	2,000	2,000	2,000	3,177
yahoo	2,000	2,000	2,073	2,000	2,000	2,000	2,000	2,000	6,094
total	17,923	19,808	22,815	22,424	22,882	16,432	17,370	23,921	29,252

Table 3.2 SWDE auto vertical page, attribute value, and target node counts. Most websites have 2,000 pages, except three. Note that attribute values are given as text, and more than one value can be given for one attribute key (the total counts are given in columns "values"). Furthermore, each value can appear multiple times in a page in separate nodes (the total counts are given in columns "nodes"). For example, the website `autoweb` contains the value `model` twice on every page. On the other hand, the website `aol` does not specify the `engine` of its cars.

Furthermore, there are several errors in the dataset. For example, some pages of the website `careerbuilder` in the job vertical have attribute values inside elements that are inside HTML comments, so they would never be displayed to the user. In the website `allmovie` of the movie vertical, rating attribute values contain just the first letter, e.g., an attribute value is `"P"`, but the corresponding HTML text fragment contains `"PG13"`. In the website `amctv` of the same vertical, director attribute values are incomplete, e.g., an attribute value is `"Roy Hill"`, but the corresponding HTML text fragment contains `"George Roy Hill"`.

3.1.2 Apify

We also use a new dataset provided by the web-scraping company Apify.[2] It consists of 10 websites from the product vertical. More details are presented in Table 3.3.

Unlike SWDE, the Apify dataset consists not only of original HTML files, but it also contains all auxiliary files needed to render each page in a browser, e.g., CSS, JavaScript assets, and images.

All pages have been scraped[3] in 2022 and since their assets are also saved,

[2]`https://apify.com/`

[3]In compliance with the EU Directive on Copyright in the Digital Single Market 2019/790.

website	pages	name	price	cat	images	short	long	spec
alza	2,493	2,493	2,478	2,493	2,493	2,493	2,493	2,477
asos	499	499	499	499	499	0	499	0
bestbuy	1,000	1,000	986	998	1,000	0	1,000	1
bloomingdales	448	462	447	448	448	0	1,087	0
conrad	1,500	1,500	1,499	1,500	1,485	1,500	1,500	1,495
etsy	250	250	250	250	250	0	1,000	0
ikea	1,972	1,972	1,972	1,959	1,972	2,261	1,917	0
notino	808	808	702	808	808	808	785	783
radioshack	499	499	499	499	498	0	6,204	4
tesco	1,500	1,499	1,465	1,499	1,499	0	17,205	21,526
total	10,969	10,982	10,797	10,953	10,952	7,062	33,690	26,286

Table 3.3 Apify dataset page and target node counts. There are 10 websites in total. Their names correspond to domains with `.com` or `.co.uk` suffixes (omitted for brevity). Some attribute keys have been shortened—`cat` means category, `short` and `long` mean short and long descriptions, respectively, and `spec` stands for specification table. A screenshot of a sample page from the website `alza` is depicted in Figure 1.5.

they are effectively archived as persistent snapshots. One can then open them in a headless browser in offline mode to extract visuals at any point in the future.

Moreover, nodes are identified using CSS selectors, so no unreliable text-to-text matching is necessary. However, the target nodes can be arbitrary inner DOM nodes. This creates a much harder problem than identifying only text fragments.

3.1.3 Observations

Since the goal is to classify nodes, it should be interesting to know how many nodes there are in a page. As can be seen in Table 3.4, these counts are very different between old websites from the SWDE dataset and modern websites from the Apify dataset.

Note that both datasets consist of only ten websites per vertical. Since one website contains many pages generated from the same template, the set of all pages in one vertical is not a very diverse set of samples. Therefore, we hypothesize a model (especially a deep neural network) trained on such data overfits easily. However, this issue is not discussed in prior research working with the same dataset.

We evaluate our model against both datasets presented in this section. Although the SWDE dataset is old, it is the most common dataset used by related research. Therefore, it can be used as a benchmark for comparing results. We use the new Apify dataset to evaluate performance of our model also on modern websites.

website	nodes
`aol`	1,113
`autobytel`	771
`automotive`	1,496
`autoweb`	644
`carquotes`	464
`cars`	744
`kbb`	937
`motortrend`	1,048
`msn`	1,553
`yahoo`	968
mean	974
standard deviation	349

(a) SWDE dataset, auto vertical.

website	nodes
`alza`	1,428
`asos`	4,410
`bestbuy`	1,845
`bloomingdales`	6,135
`conrad`	2,355
`etsy`	6,323
`ikea`	1,742
`notino`	2,312
`radioshack`	1,647
`tesco`	836
mean	2,903
standard deviation	1,988

(b) Apify dataset, product vertical.

Table 3.4 Median number of nodes across pages in each website. These counts exclude white-space text fragments and invisible or complex nodes (`<script>`, `<style>`, `<head>`, `<noscript>`, `<iframe>`, `<svg>`). Note that modern websites (from the Apify dataset) have much more nodes than older websites (from the SWDE dataset), almost three times on average. Some modern websites use complex templates resulting in over six thousand nodes per page, hence the Apify dataset has a large standard deviation.

3.2 Architecture

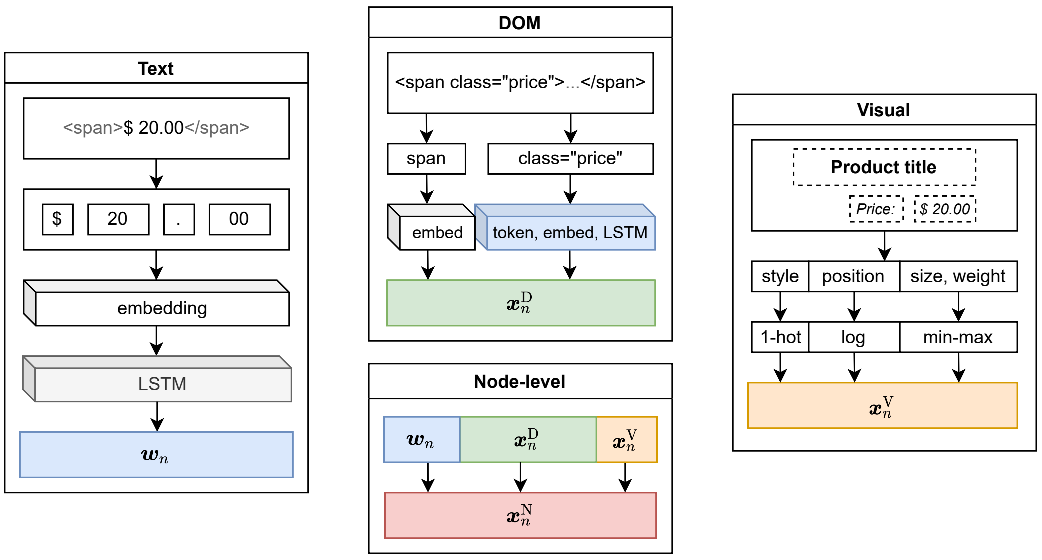
In this section, we describe the architecture of our model. Formally, we present the feature extractor g and classifier h functions defined in Section 2.2. Overall architecture is summarized in Figure 3.1.

Note that the model has configurable hyperparameters $\boldsymbol{\theta}_\mathrm{h}$ which include completely disabling or enabling some parts of the model. We describe the model with a default set of hyperparameters.

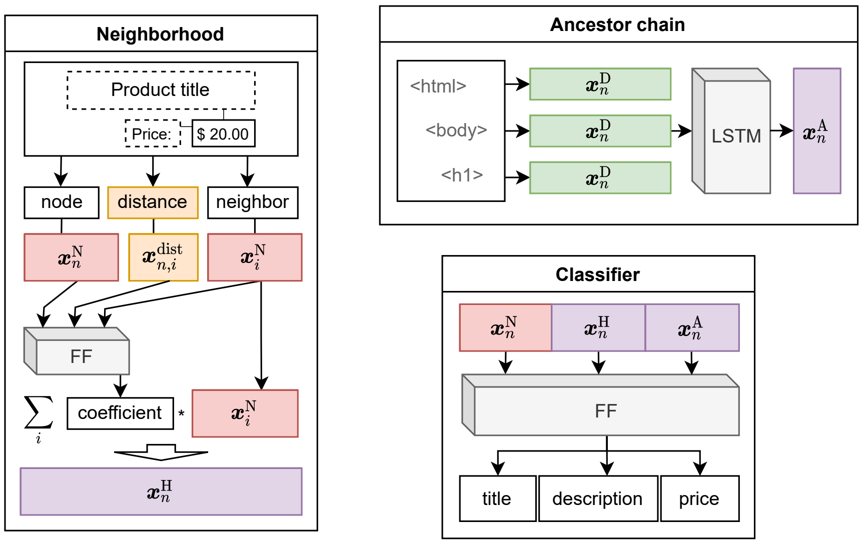
3.2.1 Node-level features

On input, the model accepts only nodes that have visuals, i.e., nodes that are rendered to the end user but not hidden nodes (e.g., only shown by JavaScript upon user interaction) nor special HTML nodes (e.g., `<script>`). More details about extracting these visual inputs are provided in Section 3.3.1.

Textual features Similarly to prior work, we extract features from node's textual content (see Section 2.1.1). Given node $n \in \mathcal{N}$, its text is tokenized to obtain a sequence of tokens t_n.



(a) Node-level features.


Neighborhood
Product title
Price:
$ 20.00
node
distance
neighbor
$\boldsymbol{x}_n^{\mathrm{N}}$
$\boldsymbol{x}_{n,i}^{\mathrm{dist}}$
$\boldsymbol{x}_i^{\mathrm{N}}$
FF
$\sum_i$
coefficient
*
$\boldsymbol{x}_i^{\mathrm{N}}$
$\boldsymbol{x}_n^{\mathrm{H}}$
Ancestor chain
<html>
<body>
<h1>
$\boldsymbol{x}_n^{\mathrm{D}}$
$\boldsymbol{x}_n^{\mathrm{D}}$
$\boldsymbol{x}_n^{\mathrm{D}}$
LSTM
$\boldsymbol{x}_n^{\mathrm{A}}$
Classifier
$\boldsymbol{x}_n^{\mathrm{N}}$
$\boldsymbol{x}_n^{\mathrm{H}}$
$\boldsymbol{x}_n^{\mathrm{A}}$
FF
title
description
price

(b) Graph-level features and the final classifier.

Figure 3.1 Overall architecture of the proposed model. For each node, textual, DOM, and visual features are obtained. Then, visual neighbors and DOM ancestors are aggregated to represent node's context. Finally, each node is classified into a pre-defined set of labels.

If n is not a text fragment, t_n is defined to be an empty sequence. Furthermore, if a node contains a long text, it is usually not very informative. Hence, t_n is restricted to contain only the first $W \in \mathbb{N}$ tokens, where $W = 15$ by default.

The tokenizer splits sentences into words, but also special strings like price (e.g., "\$2") into sequences of their symbols (e.g., "\$" and "2"). This lets the model recognize even character-level semantics where necessary and thus alleviates the need for separate character-level features.

Each token from the sequence t_n is looked up in a predefined dictionary of tokens. This results in token IDs ranging from 0 to M where M is the size of the dictionary and 0 is used for unknown tokens. Token IDs are then embedded via a pre-trained GloVe [55] weight matrix into word vectors of dimensionality $d_w = 100$. Specifically, we use GloVe embeddings pre-trained on a corpus of 2014 Wikipedia and 2011 newspaper articles with 6 billion tokens in total and vocabulary size $M = 400{,}000$. This matrix $\boldsymbol{W} \in \mathbb{R}^{(M+1)\times d_w}$ can be further trained alongside the model as part of its learnable parameters $\boldsymbol{\theta}_{\mathrm{t}}$, however, in our model, the matrix is frozen—this is a regularization technique and also makes the model much smaller as we discuss later.

Note that it is possible to choose a different tokenizer, dictionary, and pre-trained embeddings, but they need to be consistent. Since the embeddings are pre-trained on some set of tokens, a vastly different tokenizer could often produce unknown tokens. We choose the presented defaults according to SimpDOM [6] where they provide good results.

Embedding results in one vector $\boldsymbol{w}_{n,i} \in \mathbb{R}^{d_w}$ per token $t_n(i) \in t_n$. Context information from the whole sequence is propagated using a bidirectional LSTM, resulting in two vectors $\overrightarrow{\boldsymbol{w}}_{n,i}$, $\overleftarrow{\boldsymbol{w}}_{n,i}$ per token (one in each direction). All vectors are aggregated via the mean function to get a representation of the whole sequence

$$\boldsymbol{w}_n = \frac{1}{2|t_n|} \sum_{i=1}^{|t_n|} \left(\overrightarrow{\boldsymbol{w}}_{n,i} + \overleftarrow{\boldsymbol{w}}_{n,i} \right).$$

DOM features As a DOM feature (see Section 2.1.2), we take node's *semantic* HTML tag name. We define it as the tag name of the closest ancestor which wraps only a single child and which is not a `<span>` or a `<div>`. These two tag names represent container nodes without any semantic meaning, but they can be wrapped in a semantic tag such as `<h1>`.

The tag name is embedded into a vector $\boldsymbol{w}_n^{\mathrm{T}} \in \mathbb{R}^{d_t}$ via a randomly initialized weight matrix $\boldsymbol{W}^{\mathrm{T}} \in \mathbb{R}^{(T+1)\times d_t}$ where T is the number of tag names encountered during training and 0 is used for unknown tag names encountered during validation or inference. This is similar to word embedding described above, although the dimensionality here is lower, by default $d_t = 30$.

We also embed HTML node attributes. In particular, values of attributes itemprop, id, name, and class often contain semantic hints if they are present. Attribute itemprop is a form of microdata annotation (see Section 1.3.3). Attributes id and name uniquely identify nodes on a page and are commonly used for form inputs or dynamically manipulated nodes. Attribute class is primarily meant to assign a CSS style to its node and often contains words semantically identifying the purpose of the node.

These values are "humanized", i.e., converted into a sequence of space-separated words from snake_case, camelCase, and more casing styles that are commonly used and then tokenized like normal text (except the sequence length is restricted to generally different W', in our case $W' = 10$). For example, class name yat-market-pricing-bd is converted into tokens yat, market, pricing, bd.

The resulting tokens are embedded using the same matrix $\boldsymbol{W}$ used for text content and aggregated into one vector $\boldsymbol{w}_n^{\mathrm{A}} \in \mathbb{R}^{d_w}$ via the mean function. LSTM is not used since order of attribute values should not matter. However, by default we use only itemprop value, because otherwise the model easily overfits to training data, since it can learn e.g., class names specific to templates present in the training set.

Visual features Unlike most recent works, we also use visual features (see Section 2.1.3). One of them is the position and size of the node on the page when rendered in a web browser. This bounding box is encoded relatively to page size as described in Section 2.1.3.

Categorical visual features include font style, decoration, alignment, etc. These are represented as one-hot-encoded vectors, i.e., the unit vector $\boldsymbol{e}_i \in \mathbb{R}^D$ for category i of D possible categories encountered during training or the zero vector $\boldsymbol{o} \in \mathbb{R}^D$ for unseen values.

Numerical visual features include font size, weight, opacity, letter spacing, line height, etc. These are min-max scaled to the interval $[0, 1]$, i.e., the minimum of training values becomes 0, the maximum becomes 1. Such representation is easier for the model to learn [53].

Colors (text, background, or border) are converted to the HSV (hue-saturation-value) space since it should be more semantic than RGB by definition. More precisely, these colors are represented as vectors with their three components re-scaled to $[0, 1]$.

All visual features (bounding box, categorical, numerical, colors) are concatenated into one vector denoted $\boldsymbol{x}_n^{\mathrm{V}}$. By default, only a few generic features are included to avoid overfitting, namely bounding box, font style, size, weight, and color.

3.2.2 Graph propagation

Given node $n \in \mathcal{N}$, we denote its DOM features

$$\boldsymbol{x}_n^{\mathrm{D}} \overset{\text{def}}{=} [\boldsymbol{w}_n^{\mathrm{T}} \parallel \boldsymbol{w}_n^{\mathrm{A}}],$$

and node-level features

$$\boldsymbol{x}_n^{\mathrm{N}} \overset{\text{def}}{=} [\boldsymbol{x}_n^{\mathrm{D}} \parallel \boldsymbol{w}_n \parallel \boldsymbol{x}_n^{\mathrm{V}}].$$

Based on these, we introduce two page-level contextual features, *visual neighborhood* and *ancestor chain*. See Section 2.1.4 for general overview and notation.

Visual neighborhood Nodes that are in proximity of a target node usually contain cues about the type of the target node (including the fact that it is a target node). Importantly, these cues can be definitely identified only via relative visual position of the nodes as discussed in Section 2.1.3. This fact is our motivation to aggregate features from visual neighbors and provide them as one input to the node classifier h.

We define the visual neighborhood of node $n \in V_p$ in page $p \in \mathcal{P}$ to be the multi-set $\mathcal{H}_n \subseteq V_p$ of $H \in \mathbb{N}$ nodes visually closest to n.

Note that H is a small number, by default 10. However, in the edge case when a node does not have enough visual neighbors, for simplicity, the most distant one is repeated to fill the whole multi-set.

A simple distance metric is the Euclidean distance between nodes' centers. Our nearest neighbor search instead considers all four corners of each node's bounding box to determine the neighborhood. The distance used to determine nearest neighbors is denoted $\mathrm{d} : V_p \times V_p \mapsto \mathbb{R}$ and we define it as

$$\mathrm{d}(i, j) \overset{\text{def}}{=} \min_{(x_1, y_1) \in R_i, (x_2, y_2) \in R_j} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2},$$

where $R_i \subseteq \mathbb{R}^2$ yields all four corners of node i

$$R_i \overset{\text{def}}{=} \{(x_i, y_i), (x_i + w_i, y_i), (x_i, y_i + h_i), (x_i + w_i, y_i + h_i)\},$$

where (x_i, y_i, w_i, h_i) is the bounding box of node i.

Although the neighbors could be ordered by their distance to n and aggregated via an RNN, distance is not characterizing the relationship completely. Another important part is orientation, e.g., whether the neighbor is on the left-hand or the right-hand side of n. Hence, we define the visual distance feature as the combination of the absolute distance and relative differences of (x, y) coordinates

$$\boldsymbol{x}_{n,i}^{\mathrm{dist}} \overset{\text{def}}{=} (\mathrm{d}(n, i), x_n - x_i, y_n - y_i),$$

where (x_n, y_n) are center coordinates of n and (x_i, y_i) are center coordinates of its neighbor $i \in \mathcal{H}_n$.

The neighborhood is combined via an attention mechanism. First, for each neighbor $i \in \mathcal{H}_n$, a coefficient c_i is computed by a feed-forward layer from the node's features, the neighbor's features, and the visual distance between them as

$$c_i \overset{\text{def}}{=} \boldsymbol{W}[\boldsymbol{x}_n^{\text{N}} \parallel \boldsymbol{x}_i^{\text{N}} \parallel \boldsymbol{x}_{n,i}^{\text{dist}}].$$

In these coefficients, the model is expected to learn the importance of each visual neighbor.

Second, coefficients are normalized, so they sum to 1. Third, neighbor features are aggregated via a sum weighted by the normalized coefficients to obtain the neighborhood feature vector

$$\boldsymbol{x}_n^{\text{H}} \overset{\text{def}}{=} \sum_{i=1}^{H} \frac{c_i}{\sqrt{\sum_j c_j^2}} \boldsymbol{x}_i^{\text{N}}.$$

Ancestor chain In the DOM tree, nodes are contained in a hierarchy. Although this hierarchy might not directly correspond to the way the page is rendered, it contains some semantic information since it comes from templates created by developers. For example, headers and footers are constant across pages, and thus they do not contain target nodes. The model could see this information from tag names (e.g., `<header>` and `<footer>`) in a node's ancestor chain.

We define the ancestor chain of node $n \in V_p$ in page $p \in \mathcal{P}$ as the sequence $\mathcal{A}_n \subseteq V_p$ of its ancestors in the DOM tree, sorted from n's parent to the root `<html>` node. Optionally, the chain can be limited to contain only $A \in \mathbb{N}$ nodes.

For the ancestor chain, only DOM features are considered, since ancestors do not have textual content, nor do they have any interesting visuals as non-leaf nodes. These features are aggregated similarly to word vectors using a bidirectional LSTM (however, its output is restricted to dimension 10 in this case) and the mean function over the ancestor chain to produce its feature vector

$$\boldsymbol{x}_n^{\text{A}} \overset{\text{def}}{=} \frac{1}{2|\mathcal{A}_n|} \sum_{a \in \mathcal{A}_n} \left(\overrightarrow{\boldsymbol{x}_a^{\text{D}}} + \overleftarrow{\boldsymbol{x}_a^{\text{D}}} \right).$$

Note that it is possible to have a separate selection of DOM features and a separate tag name embedding matrix for the ancestor chain. In other words, the ancestor chain can have a set of parameters separate from the visual neighborhood. By default, a separate tag name embedding is used.

3.2.3 Classifier

Altogether, the described feature vectors represent a node and its neighborhood. They are concatenated into one feature vector

$$\boldsymbol{x}_n \overset{\text{def}}{=} [\boldsymbol{x}_n^{\mathrm{N}} \parallel \boldsymbol{x}_n^{\mathrm{H}} \parallel \boldsymbol{x}_n^{\mathrm{A}}],$$

and passed to a classification head consisting of a few feed-forward layers followed by ReLU activations, and one final feed-forward layer followed by a softmax output. By default, only two layers with dimensions 100 and 10 are used.

The whole model is trained in an end-to-end fashion for three epochs (i.e., three passes over all training data) using the cross-entropy loss, the Adam optimizer with learning rate 10^{-3}, and batch size 32. See Section 2.2 for general discussion about these topics.

Input selection

The presented model has been developed mostly on the SWDE dataset, because the Apify dataset became available only recently. In the SWDE dataset, only text fragments can be target nodes, so inputs to the model can be filtered to include only text fragments. However, in the Apify dataset, any inner node can be labeled.

In order for our model to be transferable between the datasets and competitive with prior work which uses the SWDE dataset, it also works only with leaf nodes (so no additional features like descendant chain are necessary). However, apart from text fragments, it also supports other node types, like images (`<img>`) which are labeled in the Apify dataset. Labels from inner nodes are propagated to all their descendants for evaluation (see Section 3.4.1).

Furthermore, many websites do not contain short description or specification nodes at all (see Table 3.3) and long description nodes are often labeled inconsistently, e.g., in a few websites, long description consists of many smaller nodes over a non-continuous range with no clear reasoning for which nodes belong to description and which do not. Therefore, we restrict the Apify dataset to have only the other four labels. Since the Apify dataset is still evolving, tackling all attribute keys is planned as future work.

Regularization

As we have discussed in Section 3.1.3, it is likely for the model to overfit to training data. We employ several regularization techniques to avoid that.

During training, a dropout is applied to LSTM and the other aggregation layers, and after each activation in the classification head. Dropout is a standard regularization technique used in neural networks that should prevent overfitting.

It randomly drops a fraction of inputs to a layer, preventing it from learning complex and fragile dependencies and thus ensuring it generalizes better [53]. We set dropout probability to 30%.

Note that in each page, there are only a handful of target nodes. This leads to imbalanced data which is a problem for training. Therefore, during training, non-target nodes are sub-sampled. More precisely, a non-target node is selected as a training sample with probability 30% like in SimpDOM [6].

Other common regularization techniques include batch or layer normalization, weight decay, freezing parts of the network, and early stopping, i.e., training only as long as the validation loss is decreasing [53]. We have experimented with these, but most do not present significant improvements in our case except freezing word embeddings as already mentioned.

Alternate approaches

We also experimented with some completely different architectures. One was based on a Transformer pre-trained for question answering, taking only text as the context (without HTML structure) and the attribute key as the question. Although it showed promising results, more customizations would be needed, requiring us to significantly re-engineer the entire model. The simultaneously published model WebFormer takes this path [10].

Another approach was based on GNNs (see Section 2.1.4). These also propagate features through a graph, similarly to our contextual features. One difference is that GNN models have whole graphs in the batch, equivalent to batch size in thousands in the case when single nodes are classified, causing some training difficulties. Furthermore, libraries for GNNs seem to assume simple graph inputs, so defining custom propagation layers turned out to be more flexible.

Lastly, we have considered using a CNN model for processing whole page screenshots like prior research [37]. However, such models usually have lots of training parameters, which we suspect would make them even more prone to overfitting. Moreover, DOM text and screenshot need to be merged in complex ways, as opposed to simply working with visual features attached to DOM nodes.

3.3 Implementation

Apart from the model described in the preceding section, our implementation includes also a data preparation tool extracting visuals using a headless browser and a simple program demonstrating inference on live pages using a pre-trained model. These three parts are briefly described in this section and their source code is also attached (see Appendix A).

All three parts have their dependencies clearly specified in the code. Furthermore, Docker[4] images are available for both development of the model and the extractor, and for runtime of the inference demo. These images provide self-contained encapsulated environments, making the implementation easily reproducible.[5]

3.3.1 Visual extractor

Recall that datasets contain only HTML files (plus assets in the case of the Apify dataset). As discussed in Section 1.3.3, visual characteristics of DOM nodes are most easily extracted by loading each page in a headless browser.

For this purpose, we introduce *visual extractor*, a command-line tool for extracting visuals from HTML datasets and saving them as JSON files. It is a Node.js application written in TypeScript.

Its fundamental feature is loading pages provided as local HTML files in a headless Chromium-like browser via the Puppeteer controller,[6] traversing the DOM tree of each page, extracting visual characteristics of each node as seen by the browser, and saving them alongside each HTML file as a hierarchical JSON.

By default, JavaScript execution is disabled for performance reasons except for a few websites that use JavaScript to construct their DOM. For such pages, the extractor also saves a snapshot of HTML after JavaScript manipulation, since visuals extracted from the modified DOM are not consistent with the original HTML (and the model needs to load both).

The extractor has many command-line flags that can be used to specify e.g.,

- the set of pages for processing,
- the number of jobs to run in parallel (implemented using a custom recyclable page pool for optimal memory consumption),
- whether to take screenshots or skip already processed pages.

The set of extracted visuals consists of over 20 attributes, mostly based on CSS presentation-level properties [14]. Except for those already mentioned that are used by our model, it also extracts many others that might need more preprocessing to be useful, e.g., size and color of border and shadow, z-index, text overflow, font family.

[4]`https://www.docker.com/`

[5]Note that the training process itself might not be completely deterministic as different hardware can compute weights slightly differently, e.g., according to floating point precision.

[6]`https://github.com/puppeteer/puppeteer`

Note that evaluating pages in the headless browser is not a completely deterministic process. The main culprit is a timeout which must be used to decide when a page is loaded, so the extraction can be performed in a reasonable time. Some pages might fail to load in the specified timeout, especially when extracting visuals in multiple parallel browser tabs. Thankfully, our data loading pipeline has the capability to validate data as described later in Section 3.3.2, hence we can iteratively re-extract visuals from invalid pages.

Wayback Machine

Another important feature of the extractor allows extracting visuals for the SWDE dataset. The dataset is old, so most of its websites have been either revamped or discontinued. In any case, their assets are not accessible, thus they are rendered incorrectly as illustrated in Figure 1.5.

The extractor intercepts requests for assets made by the headless browser and redirects them to their archived versions on Wayback Machine by Internet Archive.[7] For SWDE, it looks for versions from 2011, closest to the locally available (and labeled) HTML files.

Included is also a parallel-safe caching module which ensures that requests to the same asset are made only once. The assets are cached to disk, hence they are persisted even across separate runs. This also significantly improves performance as Wayback Machine responses have a high latency.

Visuals extracted for the verticals from the SWDE dataset that we use (see Section 3.4.1) are made available as open-source at `github.com/jjonescz/swde-visual`. Included are also the original HTML files that are difficult to obtain elsewhere. More verticals will be added over time.

3.3.2 Data loading and training

The model itself is implemented in Python using the PyTorch library.[8] Experiments mentioned in Section 3.2.3 use PyTorch Geometric[9] for GNNs and HuggingFace[10] for Transformers (whose fast tokenizers are used also in the main model).

Our implementation includes an abstract object model for representing different datasets in a uniform way. Web pages are parsed using the Selectolax HTML parser[11] with Lexbor backend engine[12] which can be around 30 times faster than

[7] `https://archive.org/web/`
[8] `https://github.com/pytorch/pytorch`
[9] `https://github.com/pyg-team/pytorch_geometric`
[10] `https://github.com/huggingface/transformers`
[11] `https://github.com/rushter/selectolax`
[12] `https://github.com/lexbor/lexbor`

the popular lxml library[13] used by prior work [6, 10, 11]. This represents a significant improvement in data processing times since a vertical can contain tens of thousands of pages.

Data needed for training or testing can easily take up more than 30 GiB of memory, reaching the capacity of our training hardware. Thus, our implementation includes a one-time pre-processing pipeline which loads all HTML files and visuals from the extractor into a SQLite database.[14] This database is queried as needed without loading all data into memory (as is common practice in Python with Pandas data frames) but in much faster fashion than reading files from disk (since SQLite has caching and some persistent file systems in the cloud where we perform our training can be really slow).

The data loading pipeline also performs comprehensive validation which checks whether all pages are labeled properly, e.g., the target nodes exist, are not empty, and are visible. This validation can be also invoked separately to aid in the dataset creation process.

Hyperparameters of the model are specified in one JSON file. These are used not only to determine model shape, but also which features are used. Hence, no features are pre-computed to support flexible experimentation, although some are cached for best data loading performance.

During model selection, TensorBoard[15] is used to inspect model performance in real time. Each model version has training checkpoints, all hyperparameters, and results stored in its own version directory for reproducibility.

3.3.3 Inference demonstration

Functionality of the model on live examples is demonstrated by a simple Node.js Express[16] server application written in TypeScript. It is a part of the extractor codebase with some modules shared.

Given the URL of a page, it is loaded in the headless browser (on the server side), its visuals, HTML, and screenshot are extracted, sent to a background Python process for feature extraction and inference, and the results are presented in a simple response from the server.

The Python inference process takes some time to load all the pre-trained weights, so it starts asynchronously with the demo server and then resides in memory, communicating with the server through a pipe. This demo is merely a proof-of-concept, a real production model would need more optimizations that are out of scope of this thesis. It is deployed online at `bit.ly/awedemo`.

[13]`https://github.com/lxml/lxml`

[14]`https://sqlite.org/`

[15]`https://github.com/tensorflow/tensorboard`

[16]`https://github.com/expressjs/express`

3.4 Evaluation

In this section, we describe experiments performed to determine performance of the proposed model, state their results and compare them to related work. Furthermore, the model is evaluated in terms of completing the criteria defined in Section 2.3.

3.4.1 Setup

The model specified in Section 3.2 is trained and evaluated separately on verticals from the SWDE dataset and the Apify dataset described in Section 3.1. As already discussed, each vertical contains only a handful of websites, making it impractical to split the dataset into standard train and test sets.

Therefore, prior works use a special form of cross-validation to evaluate performance of their models [6, 11]. More precisely, for a given vertical $v \in \mathcal{V}$, they fix the order of websites. Assume there are $N \in \mathbb{N}$ websites in the vertical v. Then, they take N cyclic permutations over the sequence of N websites and perform k-shot learning on each permutation, i.e., they train on k websites and evaluate on the other $N - k$ websites.

We develop our model on the auto vertical from the SWDE dataset (since it is closest to e-commerce, our main focus) and the product vertical from the Apify dataset. The model is also evaluated on the job vertical from the SWDE dataset. This vertical is chosen as it performed worst in SimpDOM [6], so we hypothesize it is the most difficult one, hence results on it should best represent the model's ability to generalize.

Evaluating on more verticals from the SWDE dataset is out of scope of this thesis as extracting visuals from each vertical is a very time-consuming task. The main reason is the restriction of download speed from Wayback Machine, but also the inherent indeterminism described in Section 3.3.1.

Furthermore, training websites are filtered to have 100 pages each and testing websites to have 250 pages each. This shrinks training times, makes the model overfit less, and should improve evaluation since the dataset is more balanced (as opposed to taking all available pages which would make some websites represented by more data points). These subsets are chosen randomly from all pages, but in a deterministic way, so all experiments use the same subsets.

Evaluation metrics

To evaluate performance of our models, we use the page-level F1 score defined in Section 1.4.2, which is also used by most prior works on the SWDE dataset.

Recall that a page is considered to have an attribute key predicted correctly if the model's most confident prediction is one of the target nodes.

Furthermore, our model predicts only leaf nodes as explained in Section 3.2.3. On the Apify dataset, this requires propagating labels from inner nodes to their descendants. Therefore, in this case, it is enough for the model to predict just one part of the originally labeled node.

This is still a fair metric when comparing with results on the SWDE dataset or with results reported by prior work. However, we also define an *exact match* F1 score which considers a page to have an attribute key predicted correctly only if the model predicts all target nodes in one group (one original target node induces a group of leaf target nodes after propagating the labels).

Baseline model

Our original intention was to compare our approach with the state-of-the-art model SimpDOM [6]. However, we were not able to execute its source code to reproduce its results (see Section 2.3). Thus, we create a *baseline model* with features based on SimpDOM description[17] to ensure it is trained and evaluated equivalently to our model.

Furthermore, only after inspection of its source code, we have learned that it uses gold labels to filter data even during evaluation (see Section 2.1.2). This filtering is disabled during evaluation of the baseline model, although we also report results with this filtering enabled to show that this single modification significantly affects results.

3.4.2 Results and discussion

Training has throughput approx. 22 batches per second (1 minute per vertical) when performed on a cloud-based machine having NVIDIA RTX4000 GPU with 8 GiB RAM and 8 virtual CPUs with 30 GiB RAM. Feature extraction of all data needed for training and evaluation takes 15 minutes and at most 10 GiB RAM. One complete experiment (i.e., ten cross-validation runs on one vertical) takes about 2.5 hours. The baseline model is trained much longer (about 40 minutes per one vertical), because it has substantially more parameters and is trained for 15 epochs rather than 3. Table 3.5 provides details about model sizes.

Overall page-level F1 score of our model is reported and compared with related work in Table 3.6. These are results of 5-shot learning, except for some previous work where these results are not reported. Due to these and many other possible differences in evaluation, our results are reliably comparable only with

[17] In short, these features include tag name embedding, friend cycles, text, XPath LSTM, and variable node filtering. For a full description, see the original paper [6].

model	word parameters	other parameters	checkpoint size
baseline	40.2 M	235,625	461 MiB
ours	0	228,705	156 MiB

Table 3.5 Model trainable parameter count and size on disk. Note that the checkpoint size is in uncompressed form, and it excludes the size of pre-trained GloVe embeddings that are stored separately in a compressed form and take up 129 MiB. Our model has the word embedding matrix frozen, resulting in a model with 175 times fewer parameters than the baseline.

the baseline model. We see that our model has better results than the baseline on all evaluated verticals.

Notice that our model shows viability of using visual features and GNN-like feature propagation, even though the recent model ZeroShotCeres achieves only around 47% F1 using visual features and a GNN.

Figure 3.2 further shows results of our model and the baseline per each attribute key. We can see that visuals improve prediction of all attributes, and especially price on both verticals that contain it. This is encouraging as recognizing price correctly is important for many use-cases.

However, the exact match F1 results suggest that our model is not very strong at predicting arbitrary inner nodes (and neither is the baseline). This is understandable since the model has been developed on the SWDE dataset which consists of only text fragments as explained in Section 3.2.3.

Attributes `category` and `images` are predicted especially poorly in the exact mode. This is likely caused by label propagation described in Section 3.2.3 which requires the model to predict even auxiliary nodes that do not bear much semantic meaning. Such auxiliary nodes are abundant inside `category` and `images`, e.g., text fragments separating category levels (>) or buttons for image gallery manipulation (previous/next).

In Figure 3.3, we summarize the results of our visual extractor. Even though we attempt to obtain assets from Wayback Machine as described in Section 3.3.1, not all websites have their assets archived (or not the exact versions that correspond to HTML files in the SWDE dataset which must be used since other versions would not be consistent with the labeling).

This can cause our model based on visual features to have problems with those websites from the SWDE dataset. However, experiments on these websites indicate that our model can handle even websites without visuals.

Apart from performance, we also strive to overcome qualitative limitations of prior work. These are defined as criteria in Section 2.3. Here we summarize our achievements in this regard.

		SWDE (F1)			Apify	
model	year	auto	job	mean[a]	F1	EM
SSM [68]	2008			74		
Render-Full [7]	2011			89		
ZeroShotCeres [5]	2020			47		
SimpDOM [6]	2021	92	88	94		
baseline + gold[b]		78 ± 8	87 ± 3	83 ± 6	83 ± 6	40 ± 5
baseline		72 ± 9	58 ± 7	65 ± 9	39 ± 6	28 ± 7
ours	2022	82 ± 9	72 ± 5	77 ± 7	73 ± 9	38 ± 6

Table 3.6 Overall results and comparison with prior work. Page-level F1 score is reported, including the exact match (EM) version where applicable. Standard deviation is computed for cross-validation runs (not reported by prior work). For each vertical, a model is trained on 5 websites and tested on the others from the same vertical except ZeroShotCeres and SSM which are trained on all websites but one. Baseline is our re-implementation of SimpDOM, and it illustrates how reported results can differ for various reasons (see Section 3.4.1). Therefore, only the baseline is directly comparable with our results as it is evaluated in the same way.

[a]Mean is over all verticals for SSM, Render-Full, and SimpDOM, over three verticals for ZeroShotCeres, and over two verticals for the baseline and our model.

[b]This is the baseline model evaluated with variable node filtering which uses gold labels even against validation data as in the original SimpDOM [6]. There are still some differences to the originally reported results, but larger difference is to the baseline with proper evaluation.

- Our system can handle modern websites as it has features based on visuals and is able to evaluate even JavaScript in the input pages (C1).

- The proposed model can work on as little as one page during inference (C2). We see that this makes the problem harder—the original SimpDOM uses many pages (and even their labels) to filter nodes it is evaluated on, whereas our re-implementation of SimpDOM does not do this, pushing its results substantially lower.

- We argue that our features are generic since they are not specific to one vertical (C3). This is also demonstrated by the ability of our model to be transferred to unseen verticals. Furthermore, the range of features is wide enough to be useful both for text fragments and non-textual nodes like images.

- As reported above, our model can be trained in a reasonable time on one vertical using commonly available hardware (C4). Otherwise, it is a relatively small model, most of its parameters are pre-trained word embeddings.

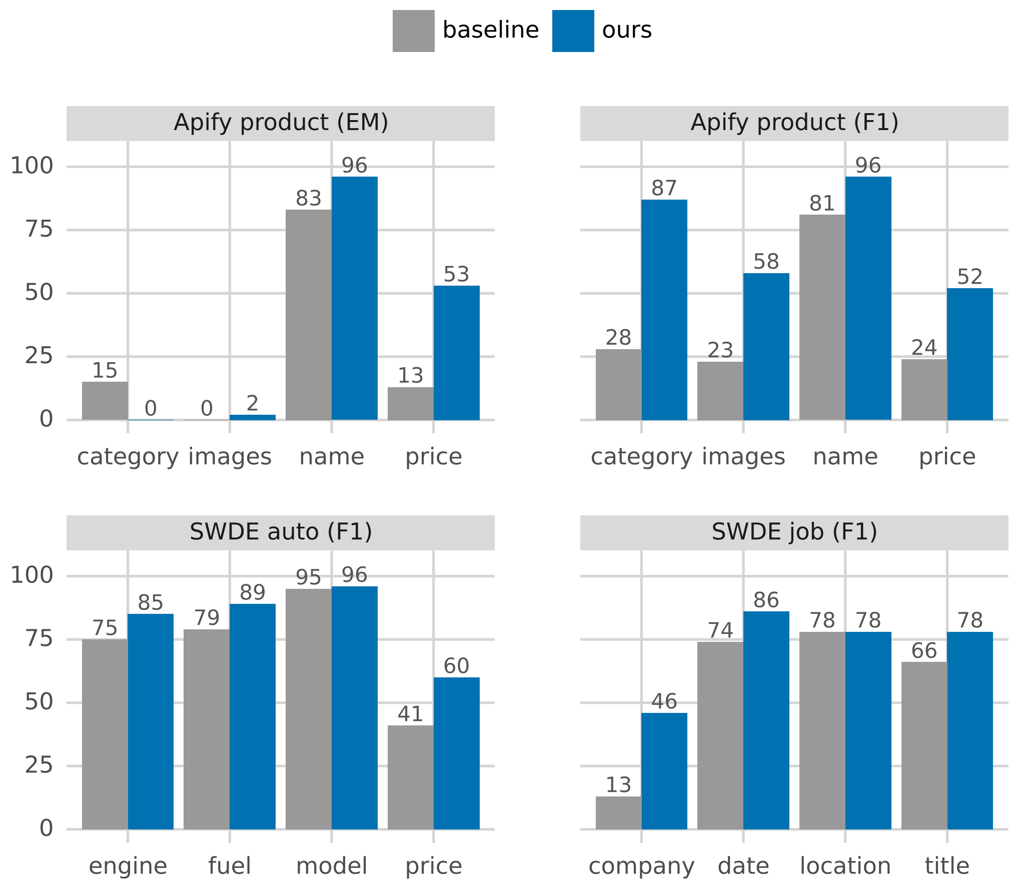


Figure 3.2 Results per attribute key. Notation and experiments are the same as in Table 3.6.

- The model is open-source, including a proof-of-concept demo application (C5, C6). This ensures that training is reproducible, and that inference is indeed possible in a live environment.

Therefore, all criteria are met by our information extraction system.

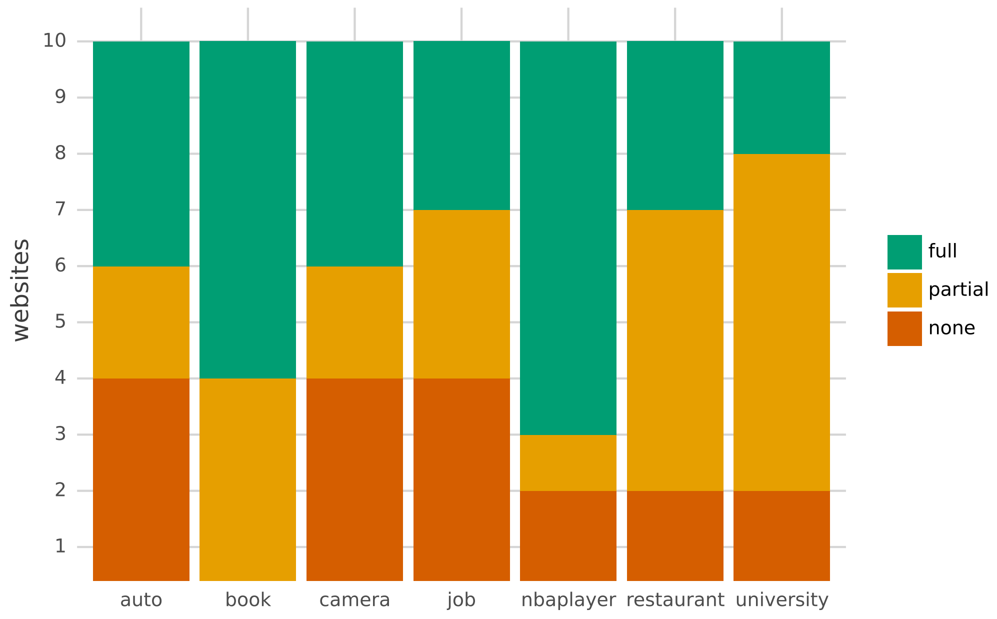


Figure 3.3 Available visuals for each website in the SWDE dataset per vertical. Determined by manual inspection of a few screenshots of each extracted website. "None" means the website has no visuals whatsoever (like Figure 1.5a). "Partial" means there is some layout but also some obvious errors. "Full" means that all visuals are present except for some minor errors.

Conclusion

In this thesis, we presented a comprehensive overview of web scraping theory and rigorously discussed possible problem frameworks and statistical approaches taken by prior work to create automatic web scrapers. We identified limitations of existing approaches and defined our goal, both formally as a function with its performance quantitatively measured, and using a set of qualitative criteria. In short, our aim was to create a system that can handle modern websites, is trained on websites from one vertical, and can perform inference given just one page from an unseen website.

We designed, trained, and evaluated a model which is competitive with results reported by recent work and outperforms our re-implementation of a state-of-the-art model. Furthermore, we fulfilled all defined criteria, overcoming some limitations of previous work. Additionally, we created a sophisticated command-line application that can extract visual features from web pages and even used it to enhance an existing old dataset by querying the Internet Archive for the missing assets.

All source code and experiment parameters are published on GitHub.[18] Moreover, all software dependencies are encapsulated in a Docker image[19] ensuring long-lasting and precise reproducibility. The exact versions used to obtain the results presented in this thesis are described in Appendix A and the corresponding source code is also available as an electronic attachment of this thesis.

We also implemented a demo application[20] to demonstrate that our model has a practical use-case, although full production deployment would require more optimizations that are out of scope of this thesis.

Future work

Nevertheless, the problem of creating an automated web scraper is a difficult one, and we see the main limitation in the lack of diverse datasets. In other words,

[18]`https://github.com/jjoneszz/awe`
[19]`https://hub.docker.com/r/janjones/awe-gradient`
[20]It is deployed online at `bit.ly/awedemo`.

training a deep learning model on ten websites causes the model to overfit fairly quickly. One way to create large datasets is by taking a publicly available set of crawled pages and using their microdata annotations as labels. Another way is to exploit existing scrapers to create labeled datasets. In any case, we would like to emphasize the need to archive each page with all its assets, so it is possible to extract any features in the future.[21]

Furthermore, in this thesis, we dealt with complex data preparation issues like extracting visual features from an old dataset or validating a freshly prepared dataset, we untangled a research area where no comprehensive surveys exist, and even found evaluation errors while trying to reproduce a prior model. Hence, we leave some time-consuming experimentation with machine learning models as future work that we prepared the ground for. For example:

- Since text is one of the most important features, experimenting with more possible tokenizers and pre-trained word embeddings could have a large impact.
- We hypothesize that our generic textual and visual features would generalize well to websites in different languages. In order to validate that, multilingual datasets need to be created.
- It could help to interpret some visual features to extract more useful information like whether a font family is serif or sans-serif. However, this might bring diminishing returns.
- Multimodal learning on page text, visual features, and screenshot could result in a powerful model, although it would likely be a complex one and would need much more training data points.
- Post-processing inference results could significantly improve prediction accuracy, e.g., finding one CSS selector for each target attribute key across a few pages from the same template.
- To complete the automatic scraper, an automatic crawler has to find the (product) detail pages to extract information from. For example, the crawler could use machine learning to classify each page before passing it to the information extractor.

[21]For example, a good choice would be the standardized WARC format (ISO standard 28500:2017). It stores not only file contents but also HTTP headers.

Bibliography

[1] D. Shete, S. Bojewar, and A. Sanghvi. “Survey Paper on Web Content Extraction & Classification”. In: *2021 6th International Conference for Convergence in Technology (I2CT)*. IEEE, Apr. 2021. DOI: 10.1109/i2ct51068.2021.9417947.

[2] M. A. B. M. Azir and K. B. Ahmad. “Wrapper Approaches For Web Data Extraction : A Review”. In: *2017 6th International Conference on Electrical Engineering and Informatics (ICEEI)*. IEEE, Nov. 2017. DOI: 10.1109/iceei.2017.8312458.

[3] T. Weninger et al. “Web Content Extraction - a Meta-Analysis of its Past and Thoughts on its Future”. In: *CoRR* (2015). arXiv: 1508.04066.

[4] R. Cooley, B. Mobasher, and J. Srivastava. “Web Mining: Information and Pattern Discovery on the World Wide Web”. In: *9th International Conference on Tools with Artificial Intelligence, ICTAI ’97, Newport Beach, CA, USA, November 3-8, 1997*. IEEE Computer Society, 1997, pp. 558–567. DOI: 10.1109/TAI.1997.632303.

[5] C. Lockard et al. “ZeroShotCeres: Zero-Shot Relation Extraction from Semi-Structured Webpages”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Ed. by D. Jurafsky et al. Association for Computational Linguistics, 2020, pp. 8105–8117. DOI: 10.18653/v1/2020.acl-main.721.

[6] Y. Zhou et al. “Simplified DOM Trees for Transferable Attribute Extraction from the Web”. In: *CoRR* (2021). arXiv: 2101.02415.

[7] Q. Hao et al. “From One Tree to a Forest: a Unified Solution for Structured Web Data Extraction”. In: *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information - SIGIR '11*. ACM Press, 2011. DOI: 10.1145/2009916.2010020.

[8] J. Zhu et al. “Simultaneous Record Detection and Attribute Labeling in Web Data Extraction”. In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '06*. ACM Press, 2006. DOI: 10.1145/1150402.1150457.

[9] T.-L. Wong and W. Lam. “Learning to Adapt Web Information Extraction Knowledge and Discovering New Attributes via a Bayesian Approach”. In: *IEEE Transactions on Knowledge and Data Engineering* 22.4 (Apr. 2010), pp. 523–536. DOI: 10.1109/tkde.2009.111.

[10] Q. Wang et al. “WebFormer: The Web-page Transformer for Structure Information Extraction”. In: *CoRR* (2022). arXiv: 2202.00217.

[11] B. Y. Lin et al. “FreeDOM: A Transferable Neural Architecture for Structured Information Extraction on Web Documents”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM Press, Aug. 2020. DOI: 10.1145/3394486.3403153.

[12] R. Cao and P. Luo. “Extracting Zero-shot Structured Information from Form-like Documents: Pretraining with Keys and Triggers”. In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 12612–12620. URL: https://ojs.aaai.org/index.php/AAAI/article/view/17494.

[13] J. Robie et al. *Document Object Model (DOM) Level 3 Core Specification*. W3C Recommendation. W3C, Sept. 2021. URL: https://www.w3.org/TR/2021/SPSD-DOM-Level-3-Core-20210928/.

[14] T. A. Jr., E. Etemad, and F. Rivoal. *CSS Snapshot 2021*. W3C Note. W3C, Dec. 2021. URL: https://www.w3.org/TR/2021/NOTE-css-2021-20211231/.

[15] S. Liu, Y. Li, and B. Fan. “Hierarchical RNN for Few-Shot Information Extraction Learning”. In: *Communications in Computer and Information Science*. Springer Singapore, 2018, pp. 227–239. DOI: 10.1007/978-981-13-2206-8_20.

[16] N. Kushmerick, D. S. Weld, and R. B. Doorenbos. “Wrapper Induction for Information Extraction”. In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*. Morgan Kaufmann, 1997, pp. 729–737.

[17] A. Omari, S. Shoham, and E. Yahav. “Synthesis of Forgiving Data Extractors”. In: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM 2017, Cambridge, United Kingdom, February 6-10, 2017*. Ed. by M. de Rijke et al. ACM, 2017, pp. 385–394. DOI: 10.1145/3018661.3018740.

[18] E. Etemad and T. A. Jr. *Selectors Level 4*. W3C Working Draft. W3C, Nov. 2018. URL: https://www.w3.org/TR/2018/WD-selectors-4-20181121/.

[19] R. Whitmer. *Document Object Model (DOM) Level 3 XPath Specification*. Tech. rep. W3C, Nov. 2020. URL: https://www.w3.org/TR/2020/NOTE-DOM-Level-3-XPath-20201103/.

[20] R. Burget. “Information Extraction from the Web by Matching Visual Presentation Patterns”. In: *Knowledge Graphs and Language Technology - ISWC 2016 International Workshops: KEKI and NLP&DBpedia, Kobe, Japan, October 17-21, 2016, Revised Selected Papers*. Ed. by M. van Erp et al. Vol. 10579. Lecture Notes in Computer Science. Springer, 2016, pp. 10–26. DOI: 10.1007/978-3-319-68723-0_2.

[21] M. Kayed and K. Shaalan. “GenDE: A CRF-Based Data Extractor”. In: *J. Web Eng.* 19.3-4 (2020), pp. 371–404. DOI: 10.13052/jwe1540-9589.19342.

[22] M. Raza and S. Gulwani. “Web Data Extraction using Hybrid Program Synthesis: A Combination of Top-down and Bottom-up Inference”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by D. Maier et al. ACM, 2020, pp. 1967–1978. DOI: 10.1145/3318464.3380608.

[23] I. A. A. Sabri and M. Man. “WEIDJ: Development of a new algorithm for semi-structured web data extraction”. In: *TELKOMNIKA (Telecommunication Computing Electronics and Control)* 19.1 (Feb. 2021), p. 317. DOI: 10.12928/telkomnika.v19i1.16205.

[24] R. Gupta, R. Kondapally, and S. Guha. “Large-Scale Information Extraction from Emails with Data Constraints”. In: *Big Data Analytics*. Springer International Publishing, 2019, pp. 124–139. DOI: 10.1007/978-3-030-37188-3_8.

[25] W. Yu et al. “PICK: Processing Key Information Extraction from Documents using Improved Graph Learning-Convolutional Networks”. In: *CoRR* (2020). arXiv: 2004.07464.

[26] G. Kim et al. “Donut: Document Understanding Transformer without OCR”. In: *CoRR* (2021). arXiv: 2111.15664.

[27] R. Zanibbi, D. Blostein, and J. R. Cordy. “A survey of table recognition”. In: *Int. J. Document Anal. Recognit.* 7.1 (2004), pp. 1–16. DOI: 10.1007/s10032-004-0120-9.

[28] G. Zheng et al. “OpenTag: Open Attribute Value Extraction from Product Profiles”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018.* Ed. by Y. Guo and F. Farooq. ACM, 2018, pp. 1049–1058. DOI: 10.1145/3219819.3219839.

[29] Q. Wang et al. “Learning to Extract Attribute Value from Product via Question Answering: A Multi-task Approach”. In: *KDD ’20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020.* Ed. by R. Gupta et al. ACM, 2020, pp. 47–55. DOI: 10.1145/3394486.3403047.

[30] R. L. L. IV, S. Humeau, and S. Singh. “Multimodal Attribute Extraction”. In: *CoRR* (2017). arXiv: 1711.11118.

[31] T. Zhu et al. “Multimodal Joint Attribute Prediction and Value Extraction for E-commerce Product”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020.* Ed. by B. Webber et al. Association for Computational Linguistics, 2020, pp. 2129–2139. DOI: 10.18653/v1/2020.emnlp-main.166.

[32] A. Baloian, N. Murrugarra-Llerena, and J. M. Saavedra. “Scalable Visual Attribute Extraction through Hidden Layers of a Residual ConvNet”. In: *CoRR* (2021). arXiv: 2104.00161.

[33] B. Potvin and R. Villemaire. “Robust Web Data Extraction Based on Unsupervised Visual Validation”. In: *Intelligent Information and Database Systems - 11th Asian Conference, ACIIDS 2019, Yogyakarta, Indonesia, April 8-11, 2019, Proceedings, Part I.* Ed. by N. T. Nguyen et al. Vol. 11431. Lecture Notes in Computer Science. Springer, 2019, pp. 77–89. DOI: 10.1007/978-3-030-14799-0_7.

[34] W. Nadee and K. Prutsachainimmit. “Towards data extraction of dynamic content from JavaScript Web applications”. In: *2018 International Conference on Information Networking, ICOIN 2018, Chiang Mai, Thailand, January 10-12, 2018.* IEEE, 2018, pp. 750–754. DOI: 10.1109/ICOIN.2018.8343218.

[35] P. Fejfar. “Interactive web crawling and data extraction”. MA thesis. Charles University, Faculty of Mathematics and Physics, 2018. URL: https://hdl.handle.net/20.500.11956/103482.

[36] M. Stoeva. “Evolution of Website Layout Techniques”. In: *Computer Technologies and Applications - Anniversary International Scientific Conference, September 15-17, 2021, Pamporovo, Bulgaria.* 2021. URL: `https://www.researchgate.net/publication/354675809_Evolution_of_Website_Layout_Techniques`.

[37] T. Gogar, O. Hubacek, and J. Sedivy. “Deep Neural Networks for Web Page Information Extraction”. In: *Artificial Intelligence Applications and Innovations - 12th IFIP WG 12.5 International Conference and Workshops, AIAI 2016, Thessaloniki, Greece, September 16-18, 2016, Proceedings.* Ed. by L. S. Iliadis and I. Maglogiannis. Vol. 475. IFIP Advances in Information and Communication Technology. Springer, 2016, pp. 154–163. DOI: `10.1007/978-3-319-44944-9_14`.

[38] Z. Cai et al. “A Vision Recognition Based Method for Web Data Extraction”. In: *Advanced Science and Technology Letters.* Science & Engineering Research Support soCiety, Feb. 2017. DOI: `10.14257/astl.2017.143.40`.

[39] B. Nguyen-Hoang et al. “Genre-Oriented Web Content Extraction with Deep Convolutional Neural Networks and Statistical Methods”. In: *Proceedings of the 32nd Pacific Asia Conference on Language, Information and Computation, PACLIC 2018, Hong Kong, December 1-3, 2018.* Ed. by S. Politzer-Ahles et al. Association for Computational Linguistics, 2018. URL: `https://aclanthology.org/Y18-1055/`.

[40] K. Bereta, G. Papadakis, and M. Koubarakis. “Ontop4theWeb: SPARQLing the Web On-the-fly”. In: *15th IEEE International Conference on Semantic Computing, ICSC 2021, Laguna Hills, CA, USA, January 27-29, 2021.* IEEE, 2021, pp. 268–271. DOI: `10.1109/ICSC50631.2021.00053`.

[41] T. Shahin. “Comparison between SPA and MPA: Competition to get the best ranking on SEO”. BS Thesis. Blekinge Institute of Technology, 2017. URL: `https://urn.kb.se/resolve?urn=urn:nbn:se:bth-15183`.

[42] C. Nevile, D. Brickley, and I. Hickson. *HTML Microdata.* WD not longer in development. W3C, Jan. 2021. URL: `https://www.w3.org/TR/2021/NOTE-microdata-20210128/`.

[43] P. Petrovski, V. Bryl, and C. Bizer. “Integrating product data from websites offering microdata markup”. In: *23rd International World Wide Web Conference, WWW ’14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume.* Ed. by C. Chung et al. ACM, 2014, pp. 1299–1304. DOI: `10.1145/2567948.2579704`.

[44] J. Wiedmann. “Joint Learning of Structural and Textual Features for Web Scale Event Extraction”. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. Ed. by S. P. Singh and S. Markovitch. AAAI Press, 2017, pp. 5056–5057. URL: `https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14520`.

[45] 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, San Diego, California, USA, June 9-12, 2003*. Ed. by A. Y. Halevy, Z. G. Ives, and A. Doan. ACM, 2003, pp. 337–348. DOI: `10.1145/872757.872799`.

[46] S. Dill et al. “SemTag and seeker: bootstrapping the semantic web via automated semantic annotation”. In: *Proceedings of the Twelfth International World Wide Web Conference, WWW 2003, Budapest, Hungary, May 20-24, 2003*. Ed. by G. Hencsey et al. ACM, 2003, pp. 178–186. DOI: `10.1145/775152.775178`.

[47] Y. Zhai and B. Liu. “Web data extraction based on partial tree alignment”. In: *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*. Ed. by A. Ellis and T. Hagino. ACM, 2005, pp. 76–85. DOI: `10.1145/1060745.1060761`.

[48] M. Bronzi et al. “Extraction and Integration of Partially Overlapping Web Sources”. In: *Proc. VLDB Endow.* 6.10 (2013), pp. 805–816. DOI: `10.14778/2536206.2536209`.

[49] C. Kohlschütter, P. Fankhauser, and W. Nejdl. “Boilerplate detection using shallow text features”. In: *Proceedings of the Third International Conference on Web Search and Web Data Mining, WSDM 2010, New York, NY, USA, February 4-6, 2010*. Ed. by B. D. Davison et al. ACM, 2010, pp. 441–450. DOI: `10.1145/1718487.1718542`.

[50] S. Gurav et al. “Web Content Extraction Using Machine Learning”. In: *International Research Journal of Engineering and Technology (IRJET)* (2018). URL: `https://www.irjet.net/archives/V5/i4/IRJET-V5I41004.pdf`.

[51] W. Petprasit and S. Jaiyen. “E-commerce web page classification based on automatic content extraction”. In: *2015 12th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, July 2015. DOI: `10.1109/jcsse.2015.7219773`.

[52] T. Guo and B. Cui. “Web Page Classification Based on Graph Neural Network”. In: *Innovative Mobile and Internet Services in Ubiquitous Computing - Proceedings of the 15th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2021), Asan, Korea, 1-3 July, 2021.* Ed. by L. Barolli, K. Yim, and H. Chen. Vol. 279. Lecture Notes in Networks and Systems. Springer, 2021, pp. 188–198. DOI: 10.1007/978-3-030-79728-7_19.

[53] K. P. Murphy. *Probabilistic Machine Learning: An introduction.* MIT Press, 2022. URL: https://probml.ai/.

[54] T. Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings.* Ed. by Y. Bengio and Y. LeCun. 2013. arXiv: 1301.3781.

[55] J. Pennington, R. Socher, and C. D. Manning. “GloVe: Global Vectors for Word Representation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL.* Ed. by A. Moschitti, B. Pang, and W. Daelemans. ACL, 2014, pp. 1532–1543. DOI: 10.3115/v1/d14-1162.

[56] B. P. Majumder et al. “Representation Learning for Information Extraction from Form-like Documents”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020.* Ed. by D. Jurafsky et al. Association for Computational Linguistics, 2020, pp. 6495–6504. DOI: 10.18653/v1/2020.acl-main.580.

[57] D. Bahdanau, K. Cho, and Y. Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.* Ed. by Y. Bengio and Y. LeCun. 2015. arXiv: 1409.0473.

[58] J. Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers).* Ed. by J. Burstein, C. Doran, and T. Solorio. Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/v1/n19-1423.

[59] R. B. Girshick. “Fast R-CNN”. In: *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015.* IEEE Computer Society, 2015, pp. 1440–1448. DOI: 10.1109/ICCV.2015.169.

[60] W. L. Hamilton. *Graph Representation Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2020. DOI: `10.2200/S01045ED1V01Y202009AIM046`.

[61] J. Zhu et al. "Beyond Homophily in Graph Neural Networks: Current Limitations and Effective Designs". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual.* Ed. by H. Larochelle et al. 2020. URL: `https://proceedings.neurips.cc/paper/2020/hash/58ae23d878a47004366189884c2f8440-Abstract.html`.

[62] P. Veličković et al. "Graph Attention Networks". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.* OpenReview.net, 2018. URL: `https://openreview.net/forum?id=rJXMpikCZ`.

[63] S. J. Nasti, M. Asger, and M. A. Butt. "Automatic Extraction of Product Information from Multiple e-Commerce Web Sites". In: *Lecture Notes in Electrical Engineering*. Springer International Publishing, Nov. 2019, pp. 739–747. DOI: `10.1007/978-3-030-29407-6_53`.

[64] J. Lloret-Gazo. "A browserless architecture for extracting web prices". In: *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020.* Ed. by C. Hung et al. ACM, 2020, pp. 2193–2200. DOI: `10.1145/3341105.3373850`.

[65] J. Wiedmann. "Machine learning approaches for event web data extraction". PhD thesis. University of Oxford, UK, 2018. URL: `https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.780612`.

[66] A. K. Yatskov, M. I. Varlamov, and D. Y. Turdakov. "Extraction of Data from Mass Media Web Sites". In: *Program. Comput. Softw.* 44.5 (2018), pp. 344–352. DOI: `10.1134/S0361768818050092`.

[67] Y. Wei et al. "A novel approach for Web page modeling in personal information extraction". In: *World Wide Web* 22.2 (2019), pp. 603–620. DOI: `10.1007/s11280-018-0631-9`.

[68] A. Carlson and C. Schafer. "Bootstrapping Information Extraction from Semi-structured Web Pages". In: *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML/PKDD 2008, Antwerp, Belgium, September 15-19, 2008, Proceedings, Part I.* Ed. by W. Daelemans, B. Goethals, and K. Morik. Vol. 5211. Lecture Notes in Computer Science. Springer, 2008, pp. 195–210. DOI: `10.1007/978-3-540-87479-9_31`.

List of Figures

List of Tables

List of Abbreviations

API application programming interface. 14, 15

CNN convolutional neural network. 24, 29, 48

CSS Cascading Style Sheets. 8, 10, 11, 13, 14, 20, 27, 28, 34, 38–40, 44, 49, 60

DOM Document Object Model. 8, 9, 11, 13, 14, 16, 17, 20, 21, 23, 26–28, 30, 31, 40, 42, 43, 45, 46, 48, 49

GAT graph attention. 30, 31

GNN graph neural network. 30, 31, 48, 50, 54

GRU gated recurrent unit. 25

HTML HyperText Markup Language. 8–15, 20, 27, 28, 31, 38, 39, 41, 43, 44, 48–51, 54

iff if and only if. 8, 10, 22

JSON JavaScript Object Notation. 15, 49

LSTM long short-term memory. 25–27, 43, 44, 46, 47, 53

NER named entity recognition. 25

NLP natural language processing. 25

OCR optical character recognition. 12, 14

RNN recurrent neural network. 25, 29, 35, 45

SPA single-page application. 13–15, 34

SWDE Structured Web Data Extraction. 37–40, 47, 50, 52–55, 75

Appendix A

Attachment

The electronic attachment of this thesis contains source code of our model, the baseline model, the visual extractor, and the demo, plus exact hyperparameters used to perform the reported experiments. Directory structure of the attachment is described in `README.md` which is also the entry point for its documentation.

Equivalent content of this attachment is also available in an online open-source repository hosted at `github.com/jjonescz/awe`. More precisely, tag `v1.0` and commit `12c8739` correspond to the attached source code. Release assets of that tag contain pre-trained weights of the model used in the live demo and also all hyperparameters used to perform cross-validation experiments presented in this thesis.

A Docker image of the demo application is available in an online registry at `hub.docker.com/r/janjones/awe-demo`. More precisely, the image tagged `1651138947` corresponds to the attached source code and contains the pre-trained model released as `v1.0`.

Similarly, a Docker image of the development environment with all software prerequisites pre-installed (but without source code or data) is available at `hub.docker.com/r/janjones/awe-gradient`, tag `1650739890`.

The verticals used for training from the SWDE dataset extended with visual attributes obtained by our extractor are available at `github.com/jjonescz/swde-visual`. The version used in this thesis corresponds to commit `d88ba05`.