Viktor Petreski and Lodi Dodevska

Report

Second assignment for Web information extraction and retrieval

Introduction

For this assignment, we implemented content extractors using regular expressions, XPath and automatic web extraction with a roadrunner-like algorithm. We tested our implementation using the provided offline web pages and a third one of our choice.

Sources

Besides the two provided web pages, as a third web page we chose globalscalecompany.com. It is a Wordpress based site for selling weighing scale related products and it is well known to us. The product page contains similar characteristics as the provided pages. Each page has details for a given product, while also providing product's variations as lists.
From each product we extracted its:

- Title
- Lowest price
- Highest price
- Short description
- Category
- Tags
- Variations:
    - Values for their specific attributes
    - List price
    - Sale price

Since each product has distinct attributes for its variations, first we extracted the names of each attribute and then based on that we extracted their values. Furthermore, not all products are on sale, so when they are not, the sale price is set to 0. One variation can not be on sale while the others are not (it's how the website works). The attributes described above are marked on Figure 1.

Fig. 1. Global Scale Co. product page extracted contents

Methods

The methods for content extraction that were used are mentioned in the introduction.
The simplest and most straight-forward method was extraction with XPath. It is intuitive to reach each element, either with full path from the root to the needed element or with shortened version using // operator.

The HTML structure was not altered in any way, unless mentioned otherwise. All the tags were left in place, without removing them before the needed element was extracted. After obtaining the wanted result, when needed, we used regular expressions to filter the unwanted tags and new lines that were in the middle of it.
Each needed element was extracted using one expression or in some cases more than one element.

Regular expressions

The regular expressions were executed using the built in python library *re.* We also added the flags *DOTALL* and *MULTILINE* to catch the new lines in with the any ***. (dot)*** expression. The lazy search/match ***.*?*** was also frequently used to match anything before a given sequence.

Regular expressions for overstock.com

```
titles="<td\s*valign=\"top\">\s*<a\s+href=\"http://www\.overstock\.com/cgi-bin/d2
\.cgi\?PAGE=PROFRAME&amp;PROD_ID=\d+\"><b>(\d{,2}-[K|k][T|t]\.?\s*(?:\S*\s*){,6}\
(?\d*\.?\d*\s\S*\)?)</b></a>", MULTILINE | DOTALL

list_prices =
"(?:nowrap=\"nowrap\"><s>([$€]\s*[0-9.,]+)?</s>|.*?</tbody></table>[\s\n]*</td><t
d\s*valign=\"top\"><span\s*class=\"normal\">)"

prices =
"(?:<b>([$€]\s*[0-9.,]+)?</b>|.*?</tbody></table>[\s\n]*</td><td\s*valign=\"top\"
><span\s*class=\"normal\">)"

savings = "(?:<span\sclass=\"littleorange\">([$€]\s*[0-9.,]+)?\s*(\(\d{,2}%\))?
</span></td></tr>|.*?</tbody></table>[\s\n]*</td><td\s*valign=\"top\"><span\s*cla
ss=\"normal\">)"

contents="valign=\"top\"><span\s*class=\"\w+\">(.*?)<br>", DOTALL | MULTILINE
```

The regular expressions for extracting the prices will work **even when one or multiple prices are missing** (the whole table row or just the text in the row). When a price is missing, in the contents, it is replaced with “N/A”.

Regular expressions for rtvslo.si

```
author_time = "<div class=\"author-name\">(.*?)<\/div>\s*<\/div>\s*<div
class=\"publish-meta\">\s*(.*?)\s*<br>", MULTILINE | DOTALL

title ="<h1>(.*?)<\/h1>\s*<div class=\"subtitle\">", MULTILINE | DOTALL

subtitle = "<div class=\"subtitle\">(.*?)<\/div>", MULTILINE | DOTALL

lead = "<p class=\"lead\">(.*?)\s*<\/p>", MULTILINE | DOTALL
```

```
content = "<figcaption itemprop=\"caption description\">\s*<span
class=\"icon-photo\"><\/span>(.*?)<\/figcaption>.*?<script.*?<p.*?>(.*?)<div",
MULTILINE | DOTALL
```

The author and publish time are extracted with one regex. The first group represents the author and second the publish time. The regular expression for content is slightly longer than the others, since we had to include the description of the images and the text from the paragraphs, and skip everything else at the same time.

Regular expressions for globalscalecompany.com

```
title = "class=\"product_title\s+entry-title\">\s*(.*?)</h1>", MULTILINE | DOTALL

price_from =
"<p\s+class=\"price\"><span\s+class=\"woocommerce-Price-amount\samount\">.*?([$€]
)</span>([0-9.,]+)</span>\s+–\s+"

price_to =
"<span\s+class=\"woocommerce-Price-amount\samount\">.*?([$€])</span>([0-9.,]+)</s
pan></p>"

description = "woocommerce-product-details__short-description\">(.*?)</div>",
MULTILINE | DOTALL

category = "class=\"posted_in\">Category:\s+<a.*?rel=\"tag\">(.*?)</a></span>"

tags = "<a href=\".*?product-tag.*?\"\s+rel=\"tag\">(.*?)</a>"

attribute = "<label>(.*?)\s*</label>"
for attr in attributes:
     varitaiton_attrs = f"<td\s+data-title=\"{re.escape(attr)}\">(.*?)</td>")


separate_list_price =
"<span\sclass=\"item\"><span\sclass=\"price\">(?:<del>)?.*?([$€])</span>([0-9.,]+
)</span>(?:</del>)?"

separate_discount_price = "<ins>.*?([$€])</span>([0-9.,]+)</span></ins>"
```

Each price is extracted using two groups in the regular expressions, since the currency sign is located in another tag and not in front of the price digits. To extract the variation attributes, we looped over the attribute names (extracted in ***attribute***) and got the values. The *attr* variable represents the name of the attribute for which the value that needs to be extracted.

XPath

To extract the contents using XPath we used the *lxml* python library. In cases where we needed to extract multiple values with one XPath, we used the and **(|)** operator and parsed the output afterwards.

XPaths for overstock.com

```
titles =
'//table[2]/tbody/tr/td[5]//table//table/tbody/tr[@bgcolor][{i}]/td[2]/a/b/text()
'
contents =
'//table[2]/tbody/tr/td[5]//table//table/tbody/tr[@bgcolor][{i}]//td[2]/span[@cla
ss="normal"]/text()'

list_price =
'//table[2]/tbody/tr/td[5]//table//table/tbody/tr[@bgcolor][{i}]//s/text()')
price =
'//table[2]/tbody/tr/td[5]//table//table/tbody/tr[@bgcolor][{i}]//span[@class="bi
gred"]/b/text()'

saving =
'//table[2]/tbody/tr/td[5]//table//table/tbody/tr[@bgcolor][{i}]//span[@class="li
ttleorange"]/text()'
```

Each expression is executed in a loop (hence the *i* in each path), since we needed to take into account that some elements may be missing. The title is the main element that is assumed to be always present. The loop stops when a title is not found twice in a row. The paths would have been much shorter if we did not care for missing data.

XPaths for rtvslo.si

```
author = '//div[@class="author-name"]/text()'

publishedTime = '//div[@class="publish-meta"]/text()

title = '//header[@class="article-header"]//h1/text()'

subtitle = '//div[@class="subtitle"]/text()'

lead = '//header[@class="article-header"]//p[@class="lead"]/text()'

content =
'//div[@class="article-body"]//p/text() |
//div[@class="article-body"]//p/strong/text() |
//div[@class="article-body"]//figure/figcaption[@itemprop="caption
description"]/span/text()'
```

The XPath expressions for rtvslo.si were really intuitive to write. We had a minor problem with the content. In order to prevent the extraction of unnecessary lines at the end of the

content we tried to filter the *figure* by “class” but at the end we realised that the current solution is simpler.

XPaths for globalscalecompany.com

```
title = '//h1[@class="product_title entry-title"]/text()'

price_from = '//div[@class="summary entry-summary"]/p/span[1]/span/text() |
//div[@class="summary entry-summary"]/p/span[1]/text()'

price_to = '//div[@class="summary entry-summary"]/p/span[2]/span/text() |
//div[@class="summary entry-summary"]/p/span[2]/text()'

description =
'//div[@class="woocommerce-product-details__short-description"]/p/text()'

category = '//span[@class="posted_in"]/a/text()'

tags = '//span[@class="tagged_as"]/a/text()'

attributes = '//table[@class="table table-hover
variations"]/thead/tr/th[not(@*)]/text()'

for attr in attributes:
       var_attr[attr] = f'//table[@class="table table-hover
variations"]/tbody/tr/td[@data-title="{attr}"]/text()'

separate_list_prices = '//span[@class="price"]//span[1]/span/text() |
//span[@class="price"]//span[1]/text()'

separate_discount_prices = '//ins/span/span/text() | //ins/span/text()'
```

Here, similar to regular expression extraction, we first obtained the names of the attributes (***attributes)*** and then looped over them to get the values for each variation’s attribute. Here we used the negation operator **neg()** with the attribute wildcard **(@*)** to filter out every table header that has any attribute in the tag.

Automatic Web extraction

We implemented a roadrunner-like algorithm for automatic web data extraction. To implement it, we followed the instructions provided in [1]. The results were satisfactory, meaning it correctly extracted most of the data, but missed some of it. We also added detailed documentation, in the code, of each function used in this algorithm.

First we preprocessed the raw HTML to remove the unwanted tags: “script”, ”style”, “meta”, “link”, “map”, “br”, “strong”, “b”, “i”. Also, we removed every attribute in the tags, since they would pose a problem when matching.
Next, we used *BeautifulSoup* to reformat the html so that each tag is in its own line and then split to obtain a list. After that, the consecutive sentences are concatenated to get one line of text instead of multiple entries in the list.

Pseudo code for the algorithm

Algorithm 1 RoadRunner-like implementation

```
1: Preprocess the wrapper and the sample
2: Call RoadRunner
3: procedure RoadRunner(wrapper, sample, idx_wrapper, idx_sample, result)
4:     if the end of the wrapper or the sample is reached then
5:         The recursion is finished, return the final result
6:     end if
7:     Get the current element from the wrapper and the sample
8:     if the elements are the same then
9:         Append the current element to the result
10:        Call RoadRunner from the next positions in the wrapper and in the sample
11:    else if the elements are not the same, but both of them are string then
12:        Append #text to the result
13:        Call RoadRunner from the next positions in the wrapper and in the sample
14:    else
15:        Search for potential loops begins
16:        Find the elements on the previous positions in both wrapper and sample
17:        Set is_optional to False
18:        if the previous tag in the wrapper is closing and the current tag in the wrapper is opening and they are of the same nature then
19:            There is a loop detected in the wrapper
20:            Find the part of the HTML code in the wrapper that makes a square
21:            if the opening tag that needs closing in the sample loop is found then
22:                Call RoadRunner within the found squares
23:                if the previous statement returned some result then
24:                    Call RoadRunner from the current position in the sample and from the position that comes after the loop in the wrapper
25:                    Append the result from the square to the main result
26:                else
27:                    is_optional = True
28:                end if
29:            else
30:                is_optional = True
31:            end if
32:        else if the previous tag in the sample is closing and the current tag in the sample is opening and they are of the same nature then
33:            There is a loop detected in the sample
34:            repeat steps 20-30 for the current element from the sample
35:        else
36:            is_optional = True
37:        end if
38:    end if
39:    if is_optional is True then
40:        if current elements in the wrapper and sample are tags then
41:            There are two tags on the same line, find which one is optional
42:            if the optional tag is in the sample then
43:                Append the optional tag to the result
44:                Call RoadRunner from the current position in the wrapper and from the position that comes after the optional tag in the sample
45:            else the optional tag is in the wrapper
46:                Append the optional tag to the result
47:                Call RoadRunner from the current position in the sample and from the position that comes after the optional tag in the wrapper
48:            end if
49:        else if only the current element in the wrapper is a tag then
50:            Find the match of the tag in the sample
51:            Append the optional elements to the result
52:            Call RoadRunner from the current position in the wrapper and from the position that comes after the optional elements in the sample
53:        else if only the current element in the sample is a tag then
54:            Find the match of the tag in the wrapper
55:            Append the optional elements to the result
56:            Call RoadRunner from the current position in the sample and from the position that comes after the optional elements in the wrapper
57:        else no optional elements were found
58:            Call RoadRunner from the next positions in the wrapper and in the sample
59:        end if
60:    end if
61: end procedure
```

Wrapper

We have noticed that the loops are sometimes misidentified. This happens in the cases where there are different numbers of elements on different pages in the same loop, ex. in the Overstock page. Overstock has a lot of tables, which have a lot of rows and columns, so the end result is enormous, since it cannot skip everything.
In the rtvslo.si there is an element “Zadnje novice”, with five latest news. The text here is identified incorrectly, i.e. it is copied from the Audi page instead of being marked with #text.
The lists, menus and header/footer content are identified correctly in almost every case on every page.

We obtained really good structure for the GlobalScale company page. It is a new, nicely structured page, with proper tags and attributes. Everything was extracted as we expected.

The HTMLs are really big, because we wanted to keep them as legible as possible. That is why we attached three separate files on the github repository, one for each web site.

References

[1] Valter Crescenzi, Giansalvatore Mecca, Paolo Merialdo
RoadRunner: Towards Automatic Data Extraction from Large Web Sites
27th International Conference on Very Large Databases (VLDB 200)