Web Information Extraction and Retrieval
Programming Assignment 2:
Document Parsing

Marko Prelevikj
63130345
`mp2638@student.uni-lj.si`

Gojko Hajduković
63180431
`gh8590@student.uni-lj.si`

Stefan Ivanišević
63170405
`si0539@student.uni-lj.si`

May 2019

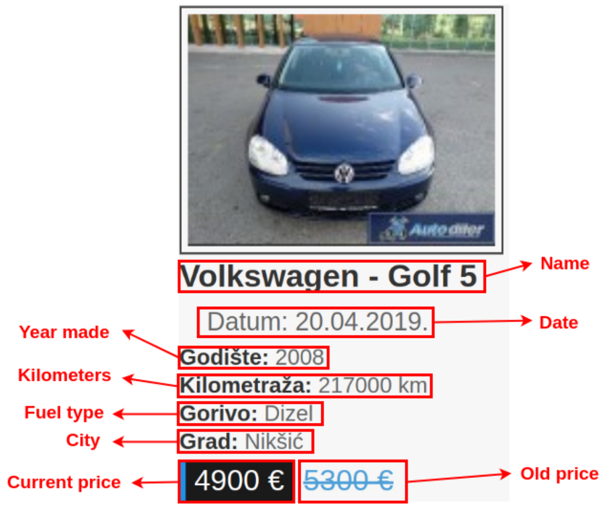
1 Introduction

Nowadays, `World Wide Web` represents an inexhaustible source of information with its size constantly and rapidly enlarging from day to day. Most of the pages that the `WWW` consists of are pages created according to some predefined, structured layout. Extraction of data from the Web pages is very useful and widely used today in many applications for all sorts of purposes from web scraping, web and data mining to all sorts of monitoring such as weather data monitoring, real-estate listing, price comparisons, etc. In this paper we introduce three different approaches in structured data extraction (*Regular expressions, Xpath queries* and *RoadRunner*) from three different types of web pages(*Overstock.com*, *RtvSlo.si* and *Autodiler.me*). Explanation and implementation specifics are provided in following sections.[1]

2 Chosen web page

For our web site of choice, we have picked the audodiler.me web page. From this web site, we have a page representing a car query and another one for trucks. The pages are downloaded and made available for offline reading i.e. extracting within the *inputs* folder. Data (e.g. "Name", "Date", "Year made", etc.) that we are going to extract from the above-mentioned pages is presented in Figure 1.

[1]Data is available on our GitHub repository, alongside with source code and results from our implementations.



Volkswagen - Golf 5

Datum: 20.04.2019.

Godište: 2008

Kilometraža: 217000 km

Gorivo: Dizel

Grad: Nikšić

4900 € ~~5300 €~~

Figure 1: AutoDiler.me Web page sample

3 Regular Expression

A regular expression is a string of text that allows us to create patterns that help match, search, and manipulate text. Regex is very helpful when having to parse a large amount of text. In our example, for extracting predefined data from the HTML pages we have used a regex that are defined bellow.

3.1 Overstock.com

`Overstock` represent a ”listing” type of Web page, having many data records that have the same layout. The regular expression used for extracting data items for a data record is:

```
[\d\-kKt]{5}.*?)<\/b>|
<s>(.*?)<\/s>|
<span class=\"bigred\"><b>(\$\d*,*\d+.\d+)</b></span>|
<span class=\"littleorange\">(\$\d*,*\d+.\d+)*\s\((\d+\%)\)</span>|
<span class=\"normal\">\s*(.*?)\s*<br>
```

Extraction of data is done using a single regular expression using multiple logical *OR*s. We did this to illustrate the power of regular expressions. This approach of extracting data is much shorter but it’s not as readable as the next two approaches we used.

3.2 RtvSlo.si

Regular expressions used for extracting data items from the `RtvSlo` web pages are:

- **Author** - `<div class=\"author-name\">(.*?)<\/div>`
- **Published Time** - `<div class=\"publish-meta\">[\n\s]*(.*?)<br>`

- **Title** - `<h1>(.*)<\/h1>`
- **Subtitle** - `<div class=\"subtitle\">(.*?)<\/div>`
- **Lead** - `<p class=\"lead\">(.*?)<\/p>`
- **Content** - `<article.*?<p.*?>(.+)<\/p>.*<\/article>`

3.3 AutoDiler.me

Regular expressions used for extracting data items for a data record from a `Autodiler` web pages shown in Figure 1 are:

- **Name** - `<h1>.*?>(.*?)<`
- **Image** - `<div class=\"oglas_thumb.*?\">.*?<img src=\"(.*?.jpg)\"`
- **Date** - `Datum: ([\d\.]+)*\.`
- **Year** - `Godi.*?te:</strong> <span>(\d{4})`
- **Kilometers** - `Kilometra.*?a:</strong> <span>(\d+)`
- **Fuel** - `Gorivo:</strong> <span>(\w+)<`
- **City** - `Grad:</strong> <span>(.*?)<`
- **Current price and Old price** - `<div class=\"priceWrapper\"><span class=\" currentPrice\">(\d+ \€)(<`
`><\/div>)*"`

4 Xpath

As one of the approaches in extracting structured data from Web pages we used XML path language, `Xpath`. Xpath represents a query/path language that is mainly used for finding any element on the Web page based on traversing a tree representation of XML document. We have converted HTML document to a XML document in order to make it accessible for `Xpath` usage. *Absolute* and *relative* `Xpath` queries are available and used. Even though, absolute `Xpath` queries are faster, since it is a direct path from the root of a tree to an element,adding or removing an element in the tree makes `Xpath` query fail. In this project we have used relative `Xpath` queries.

4.1 Overstock.com

`Overstock` represent a "listing" type of Web page, having many data records that have a same layout. `Xpath` queries used for extracting data items for a data record are:

- **Title** - `'//a/b[contains(text(),"-kt") or contains(text(),"-Kt") ]/text()'`
- **List price** - `'//td[b[contains(text(),"List Price:")]]/following-sibling::td//text()'`
- **Price** - `'//td[b[text()="Price:"]]/following-sibling::td//text()'`
- **Saving(percent)** - `'//td[b[text()="You Save:"]]/following-sibling::td//text()'`
- **Content** -`'//td[span[@class="normal"]]/span/text()'`

4.2 RtvSlo.si

Xpath queries used for extracting data items from a `RtvSlo` web pages are :

- **Author** - `'//*[@class="author-name"]//text()'`
- **Published Time** - `'//*[@class="publish-meta"]//text()'`
- **Title** - `'//*[contains(@class,"news-container")]//header//h1//text()'`
- **Subtitle** - `'//*[contains(@class,"news-container")]//header//div[@class="subtitle"]//text()'`
- **Lead** - `'//*[contains(@class,"news-container")]//header//p[@class="lead"]//text()'`
- **Content** - `'//div[contains(@class,"article-body")]//p//text()'`

4.3 AutoDiler.me

`Xpath` queries used for extracting data items for a data record from a `Autodiler` pages shown in Figure 1 are:

- **Name** - `'//*[contains(@class,"oglas_thumb")]/div[2]//h1//text()'`
- **Image** - `'//*[contains(@class,"oglas_thumb")]/div[1]//img//@src'`
- **Date** - `'//*[contains(@class,"oglas_thumb")]/div[3]//span[@class="date"]//text()'`
- **Year** - `'//*[contains(text(),"Godište")]/following-sibling::span/text()'`
- **Kilometres** - `'//*[contains(text(),"Kilometr")]/following-sibling::span/text()'`
- **Fuel** - `'//*[contains(text(),"Gorivo")]/following-sibling::span/text()'`
- **City** - `'//*[contains(text(),"Grad")]/following-sibling::span/text()'`
- **Current price** - `'//div[contains(@class,"priceWrapper")]/span[1]/text()'`
- **Old price** - `'//div[contains(@class,"priceWrapper")]/span[2]/text()'`

5 Road Runner

A more automated way of extracting data is using more advanced method, such as *RoadRunner* [1]. The purpose of this method is to generate a *wrapper* which is going to include all the data fields which contain data, and will ease the extraction of the data in the future. *RoadRunner* is an approach which takes a reference page as a wrapper, and compares it to the rest of the pages from a single domain, and tries to build a wrapper.

We have two different approaches on how to implement a *RoadRunner*-like algorithm. One is based on basic string matching, without using a special data structure for recursive moving along the document and it will produce a regex-like output, and another one which uses a tree-like structure provided by `JSoup` [2].

5.1 Version 1

First version that we have tried to build of a RoadRunner like algorithm is based on a RoadRunner description [1] and it is mostly working with lists and matching list elements (`HTML` tokens). We will neglect pseudo-code for this algorithm and just provide the description of the algorithm and problem that we have encounter during this implementation.

5.1.1 Prepocessing and matching

From the two input `HTML` files we remove `<head>` tag and everything in it and also we remove all `<script>`s and all comments from the code. Everything is in lowercase and we transfer them into `.xml` files using `BeautifulSoup` [3] module, which will help us with semi-structured `HTML` files[2].

After preprocessing we have two lists created from two input `HTMLs` (wrapper list and sample list, respectively), with elements that are tokens from the input `HTML` files. This tokens can be HTML tags with attributes or some text, and based on that we continue with matching the corresponding element of the wrapper with the corresponding element of the sample list. While matching list elements there will occur mismatches. We can have two types of mismatch, *string mismatch* and *tag mismatch*.

5.1.2 Algorithm overview

When we discover that a token is a string we change it to `#PCDATA` and continue with next element. This method we also use when we encounter a *string mismatch*, we change both strings to `#PCDATA` and continue with matching next elements. On the other hand, if *tag mismatch* happens we explore two options that can be a cause to this mismatch. First option that we consider when *tag mismatch* happens is that a mismatch is due to the iterators (which we denote as `(iterator)+`. During this process of discovering iterators we are looking for terminal tags and initial tags of an iterator. If we find them, we know that that mismatch is due to the iterator and we proceed with finding iterators. This is done recursively since more mismatches can occur during matching inner tokens of the iterator. Other *tag mismatch* can be optional which we denote as `(iterator)?`. Optional mismatches are easier to find, since they can happen only on wrapper or on sample, not on both of them at the same. We do this by looking for a wrapper tag on a sample page and if we do not find it then we know that a wrapper tag is optional. Similarly, we are looking for sample tag, just instead of finding for it on a sample page we are searching wrapper page.

5.1.3 Problem

The problem that we encountered during the execution of our algorithm is writing the data in the output file. If our algorithm never goes into recursion, everything will work smoothly, but as soon as recursion starts, our way of writing iterators and optional tags in output starts to create problems. Since we are writing mismatches as soon as we find them, and since we are looking for optional tags only when the search for iterators fails, that means that recursion will search for the optional tags last if it finds them then recursion will write them as first in the output.

[2]Since some `HTML` files can have some enclosed tags, this process in our preprocessing steps will help us to avoid further problems with this type of tags.

5.1.4 Output

Output is regex-like expression which consists of a `HTML` tags, `#PCDATA` which are strings, iterators `+` and optional tags `?`. For output files, which can be found in /outputs/road_runner1̆ folder, to be more human readable we prettify it using BeautifulSoup. It can be opened as `HTML` file in any browser, but we recommend that it is opened in text editor for easier readability.

5.2 Version 2

This approach is consisted of a preprocessing step, similar to the one described in Section 5.1 in which we clear out all the attributes from all *HTML* tags, apart from the links `<a></a>`, and images `<img/>`, from the input pages. We do all the preprocessing using `Jsoup`, and as a consequence we get a recursive structure of the *HTML* pages. Next up is the building of the wrapper, which we further discuss in Subsection 5.2.1. The core principles of this approach lie within the comparison of the nodes, and how we expand the wrapper.

5.2.1 Pseudo code

Following the preprocessing, we do the steps:

1. Pick a random page as a reference page, i.e. wrapper, and use the other page to compare the wrapper to.
2. We compare the pages level by level, recursively:
    - compare the current wrapper tag to the other tag (described below)
    - if they are an exact match, we have detected static content and we add it to the wrapper
    - if there are mismatches, we continue the comparison recursively
    - finally, compare the leaves and mark the nodes as data nodes
3. Return the result as a final wrapper.

5.2.2 Comparison function

The comparison is the most important information we get to make our decision on what to do in the recursion next. As a result from it, we get a list denoting whether the nodes are:

- equal tag name, if not - continue
- equal number of children, if yes - it is a potential candidate for an iterator
- shared tag name among children, if yes - it is most likely an iterator
- equal text contained within them, if yes - it is static content
- equal link/image attributes, if not - the link/image content is dynamic

With the comparison function we detect the static and dynamic contents of the page, the multiple items which are equal (referred to as *iterator*), and if there are heterogeneous child nodes (i.e. their tag names do not match) we handle them separately.

5.2.3 Building the wrapper

The task of building the wrapper is performed based on the properties obtained from the comparison function. We have a trivial case when the detected content is an exact match between the pages, in which we add the content to the wrapper, and continue with the processing. Another rather trivial case is when we need reach the leaf tags which have dynamic content and we need to generalize the node.

More troublesome cases are handling the iterators and the heterogeneous lists. In the case of iterators, we need to expand the same *HTML* tag to contain all the properties from every item in the iterator. We do that by comparing the elements and expanding the generic tag element which is the wrapper element. This way we ensure that all of the elements are processed and included in the final wrapper.

When dealing with the heterogeneous lists, we cannot compare nodes which are of different type (different tag names). So, in order to overcome this problem, we are using a method which zips[3] the nodes heuristically: matching the nodes by their node tags greedily, i.e. assuming that the nodes which are of same type are a part of an iterator, and leaving the unmatched ones as singles which expand the wrapper independently.

5.2.4 Node generalization

A node which is detected to have dynamic content, is generalized by removing all the text within it, and its children, and has generalized data such as:

- empty text is generalized to `$data`
- class `optional` is added to the generalized node
- links are generalized to `<a href="$link">$link-data</a>`
- images are generalized to `<img src="$img-link" alt="$img-alt">`

5.2.5 Output

The output is in the form of a generalized *HTML* page which has the generalized nodes as data holders. With this output we are preserving all the static content of the page, and we mark the places where the dynamic content is.

In its raw form, i.e. text, the wrapper is also in a human readable form, as the wrapper can be opened in a web browser of choice. For more clarity on the wrappers' attributes we advise to be opened in a text editor, as the output is prettified using `JSoup`, and it allows the reader a convenient experience while reading the *HTML* code.

5.2.6 Results

As a guideline for our implementation of this approach we used a simple example[4], which was presented at our lectures, which we further extended to cover all of the edge cases which may occur.

[3]creates a list of pairs of nodes which are of same type, and leaves the mismatched elements as pairs with empty elements

[4]included in the input files in our GitHub repository

Additionally, we analyzed the input pages, and concluded that some of them (RtvSlo.si pages) have some inconsistencies within them, which our implementation does not cover, and thus is underperforming. The inconsistency is consisted of some `div` tags being mis-aligned which makes our naive approach crash, as it violates the first and most important assumption that the tags with matching names lying on the same level and position are equal elements.

The rest of the websites are rather consistent, which means that our approach performs rather well on the rest of the pages. We can conclude this by simply comparing the wrapper sizes (number of lines in the `.html` file) to the size of the original pages. As an example, the `AutoDiler.me` websites perform the best, as our approach managed to match most of the content and generalize it really well.

On the other hand, the wrapper for `RtvSlo.si` is bigger than the original pages, which indicates that there is something wrong with our algorithm, as we are trying to minimize, i.e. generalize, the content from pages which we know are similar.

6 Conclusion

In this paper we have introduced three different approaches in structured data extraction from the web pages. We first present two approaches , `XPath` and `Regular Expressions` that are still currently widely used in structured data extraction. Both of the previously explained approaches have their shortcomings in the sense that those are manual approaches that for large companies require a lot of labor resources. As for our third approach we introduce two versions of the `RoadRunner`-like implementation which represents the automated way of generating a wrapper for data extraction.

References

[1] V. Crescenzi, "RoadRunner: Towards Automatic Data Extraction from Large Web Sites," *International Conference on Very Large Data Bases (VLDB)*, 2001. [Online]. Available: http://www.vldb.org/conf/2001/P109.pdf

[2] JSoup, *JSoup documentation.* [Online]. Available: https://jsoup.org/

[3] Beautiful Soup, *Beautiful Soup documentation.* [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/