Web Information Extraction and Retrieval
Programming Assignment 2:
Crawler implementation

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.

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* [**?**]. 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` [**?**].

5.1 Version 1

[?] As mentioned above, for this approach we have used

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 module.

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.

5.2 Pseudocode

5.3 Output

Output is regex-like expression which will

5.4

5.5

5.6 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 5.6.1. The core principles of this approach lie within the comparison of the nodes, and how we expand the wrapper.

5.6.1 Pseudocode

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.

5.6.2 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.

5.6.3 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.

A more difficult case is when

5.6.4 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">`

5.6.5 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.

6 Conclusion