Programming Assignment 2

Implementing structured data extraction

Jaka Kokošar, Danijel Maraž, and Toni Kocjan

Fakulteta za Računalništvo in Informatiko UL dm9929@student.uni-lj.si, jk0902@student.uni-lj.si, tk3152@student.uni-lj.si

Povzetek The article covers the work done in the scope of the second programming assignment as part of the subject web information extraction and retrieval.

Keywords: Data Extraction Retrieval XPath Regex Roadrunner

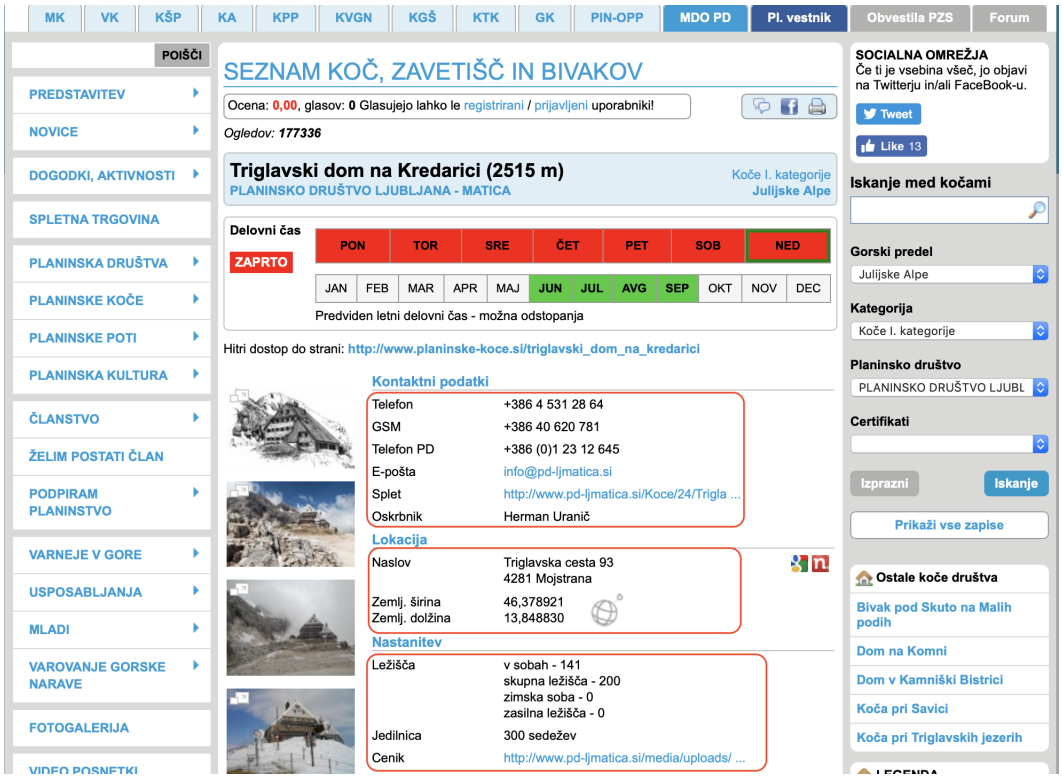
1 Introduction

After having collected the raw data with a crawler the logical next step is to convert it into a more structured format. As web pages do not have a strict shape this poses quite a challenge as any attempt of data extraction must be robust and resistant to various deformities in the html code. The report covers our attempts in implementing processes of structured data extraction in 6 different pages using basic methods such as regular expressions and xpath expressions. Alongside this an attempt was made to implement the RoadRunner algorithm which greatly simplifies the unpredictability aspect of data extraction.

2 Description of chosen web pages and data items

In addition to the mandatory websites, we selected a PZS[1] (*Planinska zveza Slovenije*) site, which is the central register of information about the operation of mountain huts in Slovenia. As shown in Figure 1, we wanted to extract information of accommodations, location and contact details of a given hut. Since there are no other resources or APIs through which we could access this information and use them for any other purpose, we are forced to obtain information directly from this site.

[1] https://www.pzs.si/



Slika 1. Data items we want to extract from pzs.si website

3 Regular expressions implementation

We defined three functions *parse_rtv_content*, *parse_overstock_content*, *parse_pzs_content*. Every function is responsible for a different page type but they are structured similarly. We extract data in three steps:

1. read the input file
2. pattern matching
3. clean the results

We only define one regular expression per page type. We extract all of the data items with a single expression in which we define all of the desired groups. After

we obtain the results we clean the data of leftover html tags and strip newlines, whitespaces and tabulators. Results are available in the project repository[2].

4 XPath implementation

4.1 jewelry

An initial Xpath expression is called on the input html which selects a *tbody* element. The children of this element represent our jewelry and other items on the page. We count the number of children and with a for loop and extra Xpath expression access each data item that interests us for each child. Any children that are malformed or do not contain the items we're interested in (such as price etc.) encounter an exception as the Xpath expression throws an error and are automatically not processed. Afterwards some minor post-processing of the acquired strings is done and the json is created.

Output

Output

4.2 cars

The items Title, SubTitle, Author, PublishedTime and Lead are all collected via seperate XPath expressions on the input html as both pages have the same structure when it comes to them. The main difference between the two pages is the structure of the article body from which we've derived our Content item. We handle this by first extracting an article body tag which contains all the contents we need. Afterwards we give the extracted element as an argument to our function *intr* which recursively iterates through the tag structure and appends all encountered text into a *nonlocal* string. The string is then briefly processed and added to our json output.

output1

output2

[2] https://github.com/LampDM/structured_data_extraction_methods/tree/master/output/regex

4.3 koce

output1

output2

5 RoadRunner like implementation

RoadRunner algorithm is based on a technique called *tree matching*, which automatically generates a common wrapper by exploiting similarities and differences among pages of same class. In principal it works as follow: at every step, matching is performed on two objects: an input page and a reference page. [1]

5.1 Implementation details

We separate our algorithm into two stages:

- tree parsing
- automatic wrapper generation

Tree parsing The result of this stage is a tree-like representation of the HTML document. Each node in the tree contains information about a specific tag in the document, for instance tag's name, it's content, attributes and also it's children nodes.

Parsing is divided into two sub-stages called *Lexer* and *Parser*:

Lexer produces a stream of tokens where each token represents and groups a meaningful subsequnce of characters from the source document (open tag, close tag, identifier, string literal, ...).

Parser consumes stream provided by the lexer and builds the tree. Building process itself is quite straightforward since html does not contain many syntax rules. Altough the main problem of course is that HTML is not strict and that the documents being parsed can contain *incorrect* sequence of tokens and still be successfuly rendered by the browser. For that purpose *Parser* contains a simple error-recovery mechanism.

Wrapper generation Wrapper is also a tree-like structure in which each node represents a result of a matching between a node from input page and a node from reference page.

The algorithm *preorderly* traverses both trees and compares current root nodes. If the nodes match in both tag name and content, they are marked as *matching*. If instead they only match in tag name we also consider them matching, but they are marked as *name-matching*. If they don't match at all we mark them as *non-matching*.

We then recursively compare their children. A problem arises when nodes don't have the same number of children. There are several ways we can solve this problem: a simple solution is to just *zip* both lists together and perform matching. A better solution might be to perform matching multiple times, each time moving the aligment of the smaller list by one, and select a matching with the least miss-matches.

[1] Automatic data extraction