Knowledge-Based Systems 39 (2013) 109–123

Contents lists available at SciVerse ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

TEX: An efficient and effective unsupervised Web information extractor

Hassan A. Sleiman [*], Rafael Corchuelo

Universidad de Sevilla, ETSI Informática. Avda. de la Reina Mercedes, s/n, Sevilla E-41012, Spain

ARTICLE INFO

Article history:
Received 23 March 2012
Received in revised form 11 October 2012
Accepted 12 October 2012
Available online 26 October 2012

Keywords:
Information extraction
Semi-structured web documents
Malformed documents
Unsupervised technique
Heuristic-based technique

ABSTRACT

The World Wide Web is an immense information resource. Web information extraction is the task that transforms human friendly Web information into structured information that can be consumed by automated business processes. In this article, we propose an unsupervised information extractor that works on two or more web documents generated by the same server side template. It finds and removes shared token sequences amongst these web documents until finding the relevant information that should be extracted from them. The technique is completely unsupervised and does not require maintenance, it allows working on malformed web documents, and does not require the relevant information to be formatted using repetitive patterns. Our complexity analysis reveals that our proposal is computationally tractable and our empirical study on real-world web documents demonstrates that it performs very fast and has a very high precision and recall.

1. Introduction

The Web is the hugest information repository. Usually, scripts are used to fill in templates with information that is retrieved from server-side databases; the results are formatted using HTML tags and CSS classes. The documents in the Web can be classified into two groups: unstructured documents, whose relevant information are pieces of free text, e.g., blog entries or news articles, and semi-structured documents, whose relevant information are records of attributes that are usually formatted as tables or lists, e.g., an on-line book store. (Note that the criteria to classify which information is relevant depends completely on the context.) Our work focuses on semi-structured documents.

Extracting the relevant information from a semi-structured document to feed an automated business process is not usually an easy task due to the irrelevant information that the template introduces in order to present it in a friendly format [3]. Information extractors are intended to help software engineers in this task [10].

Many information extractors rely on extraction rules. Although they can be handcrafted [15,24,4,42,51,50,20], the costs involved motivated many researchers to work on proposals to learn them automatically. These proposals are either supervised, i.e., they require the user to provide a number of information samples to be extracted [11,44,58,26,32,8,22,9,14,18,30,5,40,21,59], or unsupervised, i.e., they extract as much prospective information as they can and the user then gathers the relevant information from the results [62,12,16,2,28,25,60,39,46,64,67,38,59,57]. Since typical web documents are growing in complexity, a number of authors are also working on techniques whose goal is to identify the region within a web document where relevant information is most likely to be contained [37,7,61,63,27,34,65,66,52,45,35,6]. Sleiman and Corchuelo [55] have recently surveyed and compared the previous techniques.

Information extractors that rely on extraction rules do not usually adapt well to changes to the Web. Note that once a set of extraction rules is handcrafted or learnt, the Web keeps evolving and it is not uncommon that changes may invalidate the existing extraction rules. This motivated some authors to work on proposals to maintain extraction rules (semi-)automatically [31,48,49,41,33,13]. Contrarily, others worked on unsupervised proposals that do not rely on extraction rules, but are based on a number of hypothesis and heuristics that have proven to work well in many cases [1,53,17,23]; changes to a web site do not usually have an impact on these extractors since they analyse every new web document independently from the previous ones.

Our focus is on unsupervised proposals that do not rely on extraction rules. The existing proposals work on one or more input web document and search for repetitive structures that hopefully identify the regions where the relevant information resides. Álvarez et al. [1] use clustering to find a rough region where the relevant information is most likely to be located, i.e., the information region, and then use clustering, tree matching and multi-string alignment to extract prospective information; Simon and Lausen [53] first use a modified version of MDR [36] and then a multi-string alignment algorithm to extract prospective information; Buttler et al. [6] rely on six heuristics to identify the information region and to extract prospective information from it; the proposals by Refs. [17,23] focus on extracting prospective information from lists:

[*] Corresponding author.
E-mail addresses: hassansleiman@us.es (H.A. Sleiman), corchu@us.es (R. Corchuelo).

the former uses a corpus and a scoring function that helps delimit the information in a list and tabulate it, whereas the latter learns a statistical model according to which the information is also delimited and tabulated. Implicitly, the previous proposals assume that the input web documents contain similar information records since they all rely on finding repetitive structures.

A four-page abstract of our proposal was presented elsewhere [56]. In this article, we introduce TEX, which is an unsupervised information extractor that does not rely on extraction rules. Contrarily to the previous proposals, it does not require the input web documents to be translated into DOM trees, i.e., it can work on malformed web documents without correcting them, and does not require the relevant information to be formatted using repetitive structures inside a web document. It works on two or more web documents and compares them in an attempt to discover shared patterns that are not likely to provide any relevant information, but parts of the template used to generate the web documents. (We define a shared pattern of size s between two token sequences $t1$ and $t2$ as a subsequence of s consecutive tokens that occurs at least once in both $t1$ and $t2$.) The idea of identifying shared patterns lies at the heart of proposals like RoadRunner [16], which uses a multi-string alignment algorithm to learn a regular expression that models the template and its variable parts, FiVaTech [28], which relies on tree matching, tree alignment, and mining techniques, or EXALG [2], which uses two statistical techniques to differentiate the role of individual tokens and determine which are equivalent to one another. Contrarily to these proposals, TEX relies on quite a simple multi-string alignment algorithm that has proven to be very effective and efficient in practice. We have computed an upper limit to the worst-case space and time complexity of our algorithm and we have proven that it is computationally tractable (note that there are very few complexity results in this field); furthermore, we have conducted a series of experiments with 2084 web documents from 55 real-world web sites and our results confirm that our proposal can achieve a mean precision as high as 96%, a mean recall as high as 95%, with a mean execution time of 0.81 s. We conducted the same experiments using other well-known techniques in the literature, and our conclusion is that our proposal outperforms them.

The rest of the article is organised as follows: Section 2 presents TEX and describes the sub-algorithms on which it relies; Section 3 analyses its time complexity; Section 4 reports on our experimental results and compares TEX to other techniques in the literature; finally, Section 5 concludes our work.

2. Description of our proposal

We present the algorithm that lies at the heart of TEX in Fig. 1. It works on a collection of web documents, which we denote as TextSet, and a range of integers, which can introduce a bias to our search procedure. Intuitively, a TextSet is a set of Texts, which are sequences of Tokens. TEX is not bound with a particular tokenisation schema; our implementation and our experiments were carried out using a simple tokenisation schema according to which tokens represent either script blocks, style blocks, HTML tags, or #PCDATA, but this is not an intrinsic feature of our proposal. Note that we use Text as a data type that allows to represent both web documents and fragments of web documents, as well as the information that is extracted from them.

```
1: TEX ($ts$: TextSet; $min$, $max$: int): List⟨TextSet⟩
2:     $l$ = extract($ts$, $min$, $max$)
3:     $result$ = filter($l$)
4: return $result$
```

Fig. 1. Algorithm TEX.

The algorithm works in two steps: at line 2, we invoke Algorithm extract, which makes an attempt to extract the information that varies from document to document; in other words, it attempts to discard information that is likely to belong to the template used to generate the input web documents. Algorithm extract works on the collection of input web documents and searches for shared patterns of size $max, max-1, \ldots, min$. If $min > 1$ or max is less than the size of the shortest input document, then the search has a bias that may lead to situations in which Algorithm extract returns information that actually belongs to the template, which is the reason why we invoke a filtering algorithm at line 3.

Fig. 2 presents a running example. We assume that the algorithm is executed on TextSet $TS1$, which is composed of documents $T1$, $T2$, and $T3$; the result is the list of TextSets $L1$, which contains the extracted TextSets $TS4$, $TS7$, $TS11$, $TS12$, $TS9$, and $TS10$.

In the following subsections, we provide additional details on the ancillary algorithms on which TEX relies.

2.1. Algorithm extract

Algorithm extract searches for shared patterns of size max down to min in a TextSet. For instance, assume that it is invoked on the TextSet denoted as $TS1$ in Fig. 3 and that it has to search for shared patterns whose size is in the range 10 down to 1. Note that there are neither shared patterns of size 10, 9, nor 8; the longest shared pattern is <html><head><title>Results</title></Head><body>, whose size is 7 tokens. The algorithm then attempts to expand TextSet $TS1$ into three additional TextSets that contain the prefixes, the separators, and the suffixes into which the shared pattern partitions the Texts in $TS1$. In this example, there are neither prefixes nor separators, since the shared pattern is found at the beginning of the Texts in $TS1$; there are, however, three suffixes that are stored in TextSet $TS2$. The algorithm then discards TextSet $TS1$ and proceeds with the new TextSet $TS2$. The longest shared pattern that is discovered in $TS2$ is <br/></body></html>, which results in a new TextSet that is denoted as $TS3$. The same procedure is applied as many times as necessary until no more shared patterns are discovered.

We present Algorithm extract in Fig. 4. It works on a TextSet ts, a minimum pattern size min and a maximum pattern size max; it returns a list of TextSets that should contain as much prospective information as possible. The main loop at lines 3–15 iterates over all possible sizes from max down to min; for each size, the inner loop at lines 5–13 searches for a shared pattern of that size. Note that variable $result$ acts as a queue in which we initially put the TextSet on which the algorithm has to work, and then the new TextSets into which it is expanded. In each iteration of the inner loop, a TextSet is removed from $result$ and expanded at line 7. Algorithm expand, which is presented in the following section, searches for shared patterns of a given size in a TextSet; if one such pattern is found, then it is used to expand the current TextSet into new TextSets with prefixes, separators, and suffixes, which are added to $result$ so that they can be analysed later in the inner loop; if no shared pattern is found, then the original TextSet is added to a buffer. Once the inner loop finishes, the buffer contains all of the new TextSets that have been produced, and it is transferred to the $result$ variable so that the algorithm can search for new shared patterns of a smaller size, if possible.

Algorithm expand. This algorithm searches for a shared pattern of a given size inside a given TextSet; if such a pattern is found, it then expands the TextSet into a collection of new TextSets with prefixes, separators, and suffixes. We have already illustrated how Algorithm expand works on two simple cases in which the expansion