HAL open science

Lifting user generated comments to SIOC

Julien Subercaze, Christophe Gravier

To cite this version:

Julien Subercaze, Christophe Gravier. Lifting user generated comments to SIOC. KECSM-2012 - Knowledge Extraction & Consolidation from Social Media 2012, Nov 2012, Boston, Massachussets, United States. pp.48-62. hal-00990167

HAL Id: hal-00990167

https://hal.science/hal-00990167v1

Submitted on 13 May 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Lifting user generated comments to SIOC

Julien Subercaze and Christophe Gravier

LT2C, Télécom Saint-Étienne, Université Jean Monnet
10 rue Tréfilerie, F-4200 France
{julien.subercaze,christophe.gravier}
@univ-st-etienne.fr
http://portail.univ-st-etienne.fr

Abstract. HTML boilerplate code is acting on webpages as presentation directives for a browser to display data to a human end user. For the machine, our community made tremenduous efforts to provide querying endpoints using consensual schemas, protocols, and principles since the avent of the Linked Data paradigm. These data lifting efforts have been the primary materials for bootstraping the Web of data. Data lifting usually involves an original data structure from which the semantic architect has to produce a mapper to RDF vocabularies. Less efforts are made in order to lift data produced by a Web mining process, due to the difficulty to provide an efficient and scalable solution. Nonetheless, the Web of documents is mainly composed of natural language twisted in HTML boilerplate code, and few data schemas can be mapped into RDF. In this paper, we present CommentsLifter, a system that is able to lift SIOC data from user-generated comments in the Web 2.0.

Keywords: Data extraction, Frequent subtree mining, users comments

1 Introduction

The SIOC ontology [4] has been defined to represent user social interaction on the web. It aims at interconnecting online communities. Nowadays SIOC data are mostly produced by exporters [1]. These exporters are plugins to existing frameworks such as blog platforms (Wordpress, DotClear , ...), content management systems (Drupal) and bulletin boards. Unfortunately, these exporters are not yet default installation plugins for these frameworks. Therefore few administrators enable them, as a consequence the SIOC data production remains the exception.[2]. There exists also numerous closed source platforms that supports online communities, among them are the online newspapers that allows commenting on article, Q&A systems. This subset of the user generated content on the web will never be unlocked using exporters.

In this paper we present CommentsLifter, a web-mining approach that aims at extracting users' comments directly from HTML pages, in order to circumvent

[1] http://sioc-project.org/exporters
[2] http://www.w3.org/wiki/SIOC/EnabledSites

the exporter issue. The comments are identified in webpages by mining frequent induced subtrees from the DOM , and using heuristics allow to discriminate the different field of the comment (username, date, ...). This approach does not require any a priori knowledge of the webpage. We empirically evaluated our approach and obtained very good results.

The paper is structured as following. The next section presents related works on both structuring data into semantic web formats and web mining approaches. Section 3 presents a formalisation of the problem and recalls some theoretical tree mining results. Section 4 details the different steps of CommentsLifter, followed by experimental results. Finally, section 6 concludes.

2 Related works

Converting existing format of data into RDF is a cornerstone in the success of the semantic web. The W3C maintains a list of available RDFizer on its website [3]. Input data can be either structured or unstructured. In the former case, if the semantics of the data can be extracted, then the conversion can take place without human intervention [12], otherwise the user needs to manually specifiy the semantic of the data. Sesame contains an API called SAIL (Storage And Inference Layer) that can be used to wrap existing data format into RDF. The BIO2RDF project [2] uses this API to build bioinformatics knowledge systems. Van Assem presented a method for converting Thesauri to RDF/OWL [1] that has been succesfully applied for biological databases [2]. In order to convert mailing list archive into a RDF format, the authors of [10] developed SWAML, a python script that reads a collection of messages in mailbox and generates a RDF description based on the SIOC ontology. Since the input data are already structured (i.e. emails follow the RFC 4155) the conversion is straightforward. On the other side, there exist several approaches that aims at automatically or semi-automatically adressing the case where user intervention is usually required. Text-To-Onto [17] is a framework to learn ontology from text using text mining techniques as well as its successor Text2Onto [8]. However none of these papers provide a sound evaluation of the quality of learnt ontologies. This is due to the very nature of ontology modeling in which no ground truth can be assessed, there exist as many model as one could imagine for describing the same thing. In [3], Berendt details relationships between web mining and semantic web mining. The different cases (ontology learning, mapping, mining the semantic web, ...) are detailed. From this categorization, the purpose of our research falls into the category of *instance mining*, which focuses on populating instances for existing semantics. For this purpose, learning techniques have been proposed for web scale extraction with a few semantic concepts [9] and presented promising results at the time of the publication. Textrunner [20] also learns extractor to perform web-scale information extraction, presenting good precision but a very low recall. Concerning non learning techniques, automatic modelling of user profiles has

[3] `http://www.w3.org/wiki/ConverterToRdf`

been performed in [11], using term recognition and OpenCalais for named entity recognition.

Several techniques have been developed for web extraction. In [5], the authors simulate how a user visually understands Web layout structure. [15] divided the page based on the type of tags. Many recent research works exploit text density to extract content on pages [14, 19, 13]. This approach presents good results regarding article content extraction. In order to do so, the boilerpipe library[4], based on the work from Kohlschutter [14, 13] is widely used. For a more detailed survey on the different Web data extraction we encourage the reader to refer to [6]. Among other techniques DEPTA [22] (an extension of works done in [16]) presents an hybrid approach of visual and tree analysis. It uses a tag tree alignment algorithm combined with visual information. In a first step DEPTA processes the page using a rendering engine (Internet Explorer) to get the boundaries information of each HTML element. Then the algorithm detects rectangles that are contained in another rectangle, and thus build a tag tree in which the parent relationships indicates a containment in the rendered page. DEPTA then uses a string edit distance to cluster similar nodes into region. Since each data region in a page contains multiple data records, extracted tag trees must be aligned to produce a coherent database table. A tree edit distance (like in [18]) is then defined and used to merge trees. However DEPTA is not able to extract nested comments. We will use a different approach, that only requires DOM parsing technique and that is suitable for analyzing huge amount of pages. Our approach is based on a theoretical tree mining background, presented in the next section.

3 Problem Definition

The purpose of our work is to provide a solution for the leverage of Linked Data using the SIOC schema from user generated comments on webpages, without any a priori knowledge on the webpage. Our main assumption is that comments on a given webpage (even at domain scale) are embedded in the same HTML pattern. This assumption is well fulfilled in the practice since comments are usually stored in a relational database and exposed into HTML after an automatic processing. Therefore our goal is to automatically determine the HTML pattern that is used to expose the comments and then to identify the relevant information in the content to fill SIOC instances.

Basically a comment is a *sioc:Post* contained in a *sioc:Forum* container. We identified the following subset of the core-ontology properties of *sioc:Post* to be relevant for the extraction (we marked with * the mandatory properties and relationships) :

sioc:content* **:** text-only representation of the content
***sioc:topic*:** title of the comment
***dcterms:created*:** creation date

[4] `http://code.google.com/p/boilerpipe/`

and relationships :

sioc:has_creator* : points to a *sioc:UserAccount* which is the resource
sioc:has_container* : indicates a *sioc:Container* object
dcterms:title : text representation of the title
dcterms:created : date of creation of the resource.
sioc:reply_to : links to a *sioc:Post* item
sioc:has_reply : links to *sioc:Post* items

The *sioc:Container* pointed by *sioc:has_container* can be from different types in our case. We consider extraction from user reviews (*sioct:ReviewArea*), posts on forum (*sioc:Forum*), comments from blogs (*sioct:Weblog*), Q&A answers (*sioct:*) or generally for newspaper discussion (*sioc:Thread*). However, distinguishing these differents subclasses of *sioc:Container* would require classification from the webpages we intend to perform extraction on. This is out of the scope of our paper, we will therefore uniformally consider the container as an instance of *sioc:Container*. Similarly there exists subclasses of a *sioc:Post* for each container. As for the container, our algorithm will output *sioc:Post* items.

To summarize, our problem is the following : in order to generate SIOC data from raw HTML, we must identify the different items (*sioc:Post*) and their conversational relationships (*sioc:reply_to* and *sioc:has_reply*). For each item we must identify the user (*sioc:UserAccount*), the content of the post (*sioc:content*) and when possible the date and the title.

3.1 Theory recall : tree mining

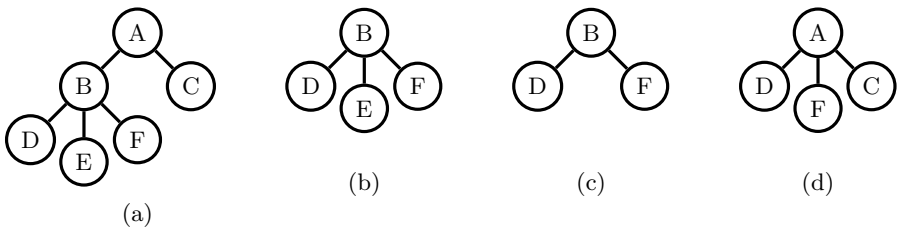

Fig. 1. Different types of subtree from tree (a) : bottom-up subtree (b) ; induced subtree (c) ; embedded subtree (d)

Tree mining deals with the task of finding pattern of interest in single tree (FREQT or AMIOT) or in forests [21]. The patterns to be extracted, as well as the trees they should be extracted from, may differ in nature. Trees are pigeonholed along three criteria: rooted/unrooted, ordered/unordered and labeled/unlabeled. A tree is called rooted if there exists a node that has been designated the root, in which case the tree may be traversed in two directions: towards and away from the root. A tree is said to be ordered if an ordering for

the children of each vertex has been defined. Finally a tree is a labeled tree if each node is given a unique label. For our concern, i.e. tree mining using DOM trees, we will consider the case of rooted labeled ordered trees.
The patterns to be mined are subtrees. There are different types of subtrees for there exists different types of trees. We present in the following the most common types of subtrees (from [7], which provides a very detailed review on frequent subtree mining). For each case, we consider a rooted tree T with vertex set V and edge set E and a subtree T' with respectively V' its vertex set and E' its edge set. The main types of subtrees are as follows.

Bottom-up subtree T' is a *bottom-up subtree* from T iff : $V' \subseteq V$, $E' \subseteq E$; for a vertex $v \in V$, if $v \in V'$ then all descendants of v must be in V' ; the ordering of the siblings must be preserved in the subtree. Intuitively *a bottom-up subtree* T' can be obtained by taking a vertex from V together will all its descendants and the corresponding edges.

Induced subtree T' is an *induced subtree* from T iff : $V' \subseteq V$, $E' \subseteq E$; for a vertex $v \in V$, the left-to-right ordering of the siblings must be preserved in the subtree, i.e. it should be subordering of the corresponding vertices in T. Intuitively a *induced subtree* T' can be obtained by repeatedly removing leafs from T.

Embedded subtree T' is an *embedded subtree* from T iff : $E' \subseteq E$, $(v_1, v_2) \in E'$ where v_1 is the parent of v_2 in T' only if v_1 is ancestor of v_2 in T. Intuitively, as an embedded subtree, T' must not break the ancestor-descendant relationship among the vertices of T.

Figure 1 illustrates these three types of subtrees from a given data tree through three different examples. To better understand the difference between the subtrees, one can say that bottom-up subtrees are *complete* subtrees while inducted subtrees allow to remove nodes horizontally in the subtree and finally embedded subtrees allow both horizontal and vertical removals. We have the following relationship: bottom-up subtree $\subseteq$ induced subtree $\subseteq$ embedded subtree. Following the same order, algorithms' complexity and running time vary with the type of subtree one wants to extract. Therefore it is desirable to well identify which type of subtree we are looking for.

3.2 Frequent Subtree Mining

In the case of product listing extraction, the goal is to extract frequent subtrees that are identical in the page. For this purpose, a bottom-up subtree mining objective is sufficient. For selecting a feature among mined items, for example title and price, a filter on the leaf nodes can be applied. For the case of comments extraction, the patterns can be nested. For instance lets assume that the pattern that we are looking for is $[a[p; br; p; div]]$. If we encounter an answer to a comment, i.e. nested instances in the pattern, the tag tree for the comment and its answer could be as follows : $[a[p; br; p; div; [a[p; br; p; div]]]]$. In the case of a single comment we will encounter our pattern $[a[p; br; p; div]]$. We observe that nodes

can be skipped horizontally along with their descendants, which eliminates the *bottom-up subtree* type. We observed empirically that instances are nested in the way we described previously : the direct parenting relation is preserved. Consequently *induced subtree* is a sufficient type of subtree for our purpose. Embedded subtree mining could also be used, however since the algorithms complexity grow with the complexity of the pattern to mine, *induced subtree* mining is definitely more appropriate.The major advantage of subtree mining over existing works is that it provides a mine once extract many approach. For mining a whole website, one would need to mine the pattern from only one webpage and could later extract data by simple pattern matching on other pages, thus saving large amount of computation time. In the next section, we present SIOCizer, our approach to comments extraction based on frequent subtree mining.

4 CommentsLifter

We now start the description of CommentsLifter. This section presents the different steps of our algorithm. The underlying idea of CommentsLifter is to use simple observations on Web pages structure to reduce the candidate set generation. This allows to minimize the error rate while selecting the winning pattern, and to contribute to the runtime performance optimization objective. To extract comments from a Web page, CommentsLifter uses seven steps: Document preprocessing, Frequent subtrees extraction, Clustering, Merging, Pattern expansion, Winner selection, Field extraction, Data extraction. In the next subsections we detail the process followed in each step of our algorithm.

4.1 Preprocessing

Recommendations issued by the W3C aim at specifying the languages of the World Wide Web, among other various versions of HTML and CSS. While pages following these recommendations produces clean DOM trees, they represent only 4.13 % of the Web pages[5]. The remaining pages are made of wrongly formatted HTML code that is often referred to as "tag soup"[6]. Due to the large portion that these pages represent, Web browser engines are able to handle malformed HTML tags, improperly nested elements, and other common mistakes. In our algorithm, the first step involves to convert any input HTML document (malformed or not) and to output a well-formed document as DOM tree. For this purpose, any dedicated library (Jsoup[7], Jtidy[8]) or browser engine can be used.

[5] http://arstechnica.com/Web/news/2008/10/opera-study-only-4-13-of-the-Web-is-standards-compliant.ars
[6] `http://en.wikipedia.org/wiki/Tag_soup`
[7] `http://jsoup.org/`
[8] `http://jtidy.sourceforge.net/`

4.2 Frequent subtrees extraction

The next step of our algorithm consists in the generation of a first candidate set. For this purpose we extract frequent depth-two trees from the DOM and store them in a cardinality map (an example is given in Table 1). The basic of our approach is to select patterns with a depth of two that we will later expand into largest patterns if possible (empirical results in Section 5 show that the average depth of comment pattern is 4.58). For this purpose, a tag tree is generated from the preprocessed DOM. CommentsLifter traverses the tree in a top down fashion and stores the encountered trees of size two (each node that has children, along with its children). The results are sorted, as presented in Table 1. Candidates with less than two occurrences are discarded, since we assume that there are at least two comments. The same assumption is made for instance by [22]. For each pattern occurrence, the encountered instance is stored in a multimap(in fact we only store the label of the parent node).

Count	13	12	10	8
Tree	div[a;br;i;p]	ul[li;li]	div[p;p]	div[p;p;p]

Table 1. Example of two depth candidates.

At this stage our candidate set is initialized and necessarily contains the instances of comments we are looking for.

4.3 Clustering

Comments in a Web page appears in a continuous manner, it is very unlikely that comments are stored in different branches of the DOM tree. We did not encounter the case of split comments in different subtrees during our experiments. In fact, comments are organized in a tree structure, where the beginning of the comments block is the root of the tree (node with label 0, not depicted on the figure). Since comments are located in the same subtree, we process to a clustering phase of the occurrences for each pattern.

In this step, we aim at clustering co-located instances of the same pattern. In other words, the algorithm builds the couples $(pattern, Instances)$, where each pattern is associated with a set of instances matching it that are located in the same subtree and close to each other. Consequently, one pattern can be associated multiple times with a unique set of instances, whose member is distinct from any other member of another associated set to the pattern. This means that each couple $(pattern, Instances)$ is splitted into different $(pattern, Instances)$ where the instances in $Instances$ do belong to the same subtree in the DOM. The basic of our algorithm is a distance-based clustering.For each given pattern, the algorithm sorts its instances along their depth in the tree. At each depth, we check for each instance if it has a parent in the previously found $(pattern, Instances)$. For the remaining instances, we cluster them using a classical node distance in

trees : $d(a, b) \in \mathbb{N}$ is the length of the shortest path between the nodes a and b. After building the set of $(pattern, Instances)$, we remove elements where the cardinality of instances is equal to one.

For example, running our algorithm on the tree provided by Figure 2 with the pattern $ul[li; li]$ would produce two couples (pattern,instances) that are depicted with dotted and dashed boxes in the same figure.

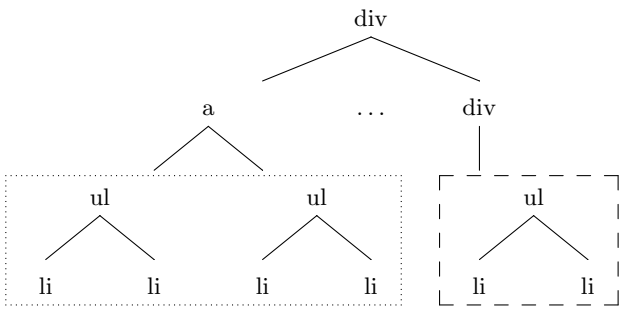

Fig. 2. Example output for section 4.3 with the pattern $ul[li; li]$

At the end of this step, CommentMiner holds the set of couples $(pattern, Instances)$ that matches a pattern to a set of co-located instances in the DOM tree. In the next step, we try to identify mergeable patterns from this data structure.

4.4 Merging

HTML patterns that contains comments often have a depth greater than two (see Section 5 for more details). Consequently our candidate set may at this step contain couple of $(pattern, Instances)$ that in fact do belong to the same global pattern, that we did not discover yet because the pattern expansion process will occur later. Instead of discovering the same pattern from different candidates, we aim at merging these candidates beforehand for optimization.

For this purpose we perform a pairwise comparison between the sets of instances and we process a merge in the case where every element s of set S_1 is topped by an element in the set S_2. An element is topped by another if the latter is a parent of the former. If a set of instances is topped by another set, we discard this set and restart the process until no further updates to the candidate set are performed.

Algorithm 1 presents this merging process. After this step, the candidate set is again drastically pruned since we eliminated all potentially duplicate patterns for the expansion phase. This process, together with the previous clustering process, are the key phases of our algorithm since they discard both duplicate and irrelevant candidates.

Algorithm 1: $MergePatterns$: Merge similar pattern

```
Data: A collection of couple (Pattern,Instances) Input where all patterns are
      different,
minDepth the mininum depth of the instances in the DOM tree
Result: A collection Candidates of (Pattern,Instances) where similar couples
        have been merged
recursion ← false;
Candidates ← ∅;
for i : 0 ... Input.size do
    for j : i ... Input.size do
        //Depth of the highest common element ;
        commonDepth ← depth(HighestCommon(Input[i].Instances,;
        Input[j].Instances));
        //Save useless computation ;
        if commonDepth < minDepth then
            continue ;
        end
        Small ← setWithLessInstances(Input[i], Input[j]);
        Large ← setWithMoreInstances(Input[i], Input[j]);
        if ∀k ∈ Small.Instances, ∃l ∈ Large.Instances, isParentOf(l, k) then
        Candidates ← Input \ Small;
        //Restart the merging process;
        MergePatterns(Candidates);
        //Cut the current call;
        break;;
        else  Candidates ← Candidates ∪ Large ∪ Small
    end
end
return Candidates;
```

4.5 Pattern expansion

Since the candidate set contains simple pattern (i.e. of depth two), we process a pattern expansion to discover the fully matching patterns. Patterns may be expanded in both direction, towards the top and the bottom of the tree.

For each candidate $(pattern, Instances)$, we distinguish two cases. In the first case every instance is at the same depth in the tree, this is the case of product listing extraction that we call *flat case*. This is the case of bottom-up subtree mining (see Figure 1). The second case also called *nested case* is the one where instances belong to the same subtree but at different depths, this is the case of induced subtrees in Figure 1 . Top expansion is straightforward, we check if the type of the parent node (i.e. HTML markup tags) for every instance is the same, in this case we expand the pattern with the new parent node and update the instances consequently. This process is executed until all the instances do not share the same tag as parent node or if the same node (in sense of label in the tree, not HTML markup tag) is the parent of all instances. This process is the same for both *nested* and *flat* cases.

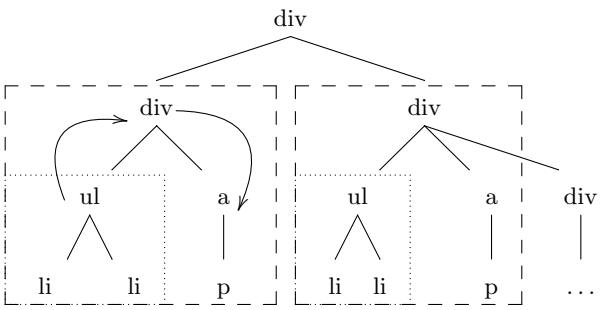

Fig. 3. Pattern expansion process, top expansion until the same node (top div node) is shared by both instances, then bottom expansion for induced subtree (the most right div is skipped)

Once a pattern is expanded in the top direction, the bottom expansion takes places. This bottom expansion in the *flat case*, similarly to the top expansion is simple since *top-down subtrees* are easy to extract. One just needs to traverse the instances trees in a top-down most left direction, looking for nodes that are shared by all instances. Once a node is not present in every instance the algorithm uses backtracking to select the next sibling, and then to continue its process.

However the *nested case* is not trivial since we look for embedded subtrees. Standard algorithms such as AMIOT and FREQT are dedicated to this task, but as we mentioned in section 3.2 they performed poorly on the full Web page, either

the running was excessively long (some minutes) or the program ended with an exception. To make an efficient use of these algorithms, we take advantage of specificities of comments extraction. Apart from the aforementioned issues, the biggest problem we encountered using AMIOT or FREQT was that we were unable to adaptatively select the support (in fact, a percentage) for a Web page. However in our pattern expansion case, we know how much instances are present for each couple $(pattern, Instances)$. Thus, we performed modification in the algorithms (see Implementation in section 5.1 for further details) to discard candidates pattern not on a minimum support base, but on a strict occurrence equality base. Finally, instead of using the complete Web page as the data tree, we construct a tree by adding all the instances in a tree. More precisely since the instances are stored using their top nodes, we build the data tree by adding the subtrees under the instances top nodes as children of the root node. Therefore, our data tree contains on ly the relevant instances. Consequently the set of candidates pattern is drastically reduced, compared with the use of the whole DOM page. To drive AMIOT in the right direction, the candidate pattern set is initialized with the current depth-two pattern, thus avoiding useless candidate generation for the first stage. Figure 3 depicts this expansion process . In this figure we present the instances as they appear in the DOM tree. Our starting pattern is $ul[li; li]$, and its instances are represented within the dotted boxes. Both instances have a *div* node with a different label as parent, consequently the pattern is expanded to the top : $div[ul[li; li]]$. Next, both instances again have a *div* node as parent but in this case this is the same node. The top expansion process finishes. For the bottom expansion, we consider this subtree as the data tree. AMIOT (resp. FREQT) performs a left to right expansion that adds the node a to the pattern: $div[ul[li; li]; a]$. The right most div node is discarded since it does not belong the left instance (the occurrence is one whereas the algorithms expects two). Then AMIOT adds the node p to the pattern that becomes $div[ul[li; li]; a[p]]$. The figure does not show the part under the right most div, in this part we could find for example another instance of $div[ul[li; li]; a[p]]$, resulting in a nested comment.

4.6 Winner selection

Recurring structures competing with the comments pattern in the candidate set are usually menu elements, links to other articles. In [13, 14], Kohlschütter developed a densitometric approach based on simple text features that presents excellent results (F_1-Score : 95 %) for news article boilerplate removal. User generated comments also differentiate from menu elements on their text features. Comments are from different text lengths, the link density is low since comments are not part of a link as a difference with menu items. We developed simple heuristics, based on our observations, to discard irrelevant candidates and to rank the remaining candidates.

Our experimentation showed that instances with a link density greater than 0.5 (Kohlschutter found 0.33 for news article) are always boilerplate. Short comments in very complex HTML boilerplate patterns can produce instances with a

quite high link density. We empirically observed that links are on the username or on its avatar and links to a profile page. Consequently we discard candidates where the average link density is above 0.5.

For the remaining candidates their score is given by the following formula:

$$Score(p, I) = \overline{lgt(I)} \times \sqrt{\frac{1}{n-1} \sum_{j=1}^{n} \overline{lgt(I)} - lgt(I_j)^2} \quad (1)$$

The above formula computes the average text length times its standard deviation. This heuristic promotes candidates where the instances have longer text length with variable length. Finally we did not use heuristics based on a lower bound of words (as in [13, 14]). Once again comments extraction differs from traditional boilerplate removal since some comments are sometimes just one word (e.g. *lol*, *+1*, *first*) or there may be longer than the article they are commenting. Therefore the standard deviation is very useful for eliminating menu items where the text length is often very close among their instances. The $(pattern, Instances)$ couple with the highest score is promoted as the winner. At this step, the algorithm output the tree of instances, i.e. we have the structure of the conversation, but we need to further structure the conversation by identifying the different fields in the pattern.

4.7 Structuring the content

Once the HTML pattern containing comments has been selected, the next step of our algorithm consists in extracting the related SIOC fields as we described in section 3. Two fields are always present in a comment : the username and the content of the comment. From our observations on various website (online press, Q&A, blogs, reviews), two other fields appear often. Firstly the date when the comment has been posted occur in 97.95 % of the cases (See Table 2). Comments less often have a title, but the percentage we measured (24.48 %) remains high enough to be of interest. We voluntarily skipped the extraction of rare fields (vote on comments, account creation date) because of their few occurrences and the fact that they complicate the whole fields identification process. From our observations, we noticed that content and title are always contained into their own HTML markup tags, i.e. it is very unlikely to see the title and the content in the same `div`. However we noticed that username and date often occur between the same markup tag, for example we often encountered comments fields such as `<div>John, May 25, 2012 at 5:00 p.m</div>`. Therefore we apply the following procedures : first, identifying the date field within the pattern. We store the location of the date in the pattern and remove dates in every instances. For instance, the previous example will become`<div>John</div>`. At this point we are sure that the fields we are looking for will be in distinct nodes of the pattern.

Date parsing Date parsing library such as JodaTime[9] and JChronic[10] perform well on extracting date from messy data. These libraries takes a String as input and return a Date object. However they do not offer the feature of returning the original text without the substring containing the date. Therefore we developed our library, using features and heuristics from both of the above mentioned libraries, that offers the date string removal feature.

Fields selection Our heuristic is based again on simple observations. We know that every comment contains at least a username and a content. Date and title being optional. For each leaf of our pattern, we compute the following measures over its instances : percentage of date found, average text length, standard deviation of text length, text entropy and average word count, standard deviation on word count. Using these values, we build two candidate sets, the first one for the date the other one for title, content and username. We distinguish these fields using the fact that title, content and username are unstructured text, however dates have a particular structure (containing year, days, ...).

A node in the pattern is a candidate for the date field if its percentage of date found is over 0.7 (to take into account the fact that date extraction is not perfect), has more than two words and has a coefficient of variation on the word count that is inferior to 0.2. This latter condition requires that the number of words is very close from one instance to another. We did not set value to zero to avoid discarding fields where the date has a variable length, for example `one hour ago` and `yesterday`. From these candidates, we pick the field with the highest entropy in order to discard constant fields that may have been recognized as a date. However if the set is empty, then no date are specified for the comments, practically this happens very rarely.

The second candidate set should contain only nodes that instances own textual data. For this purpose we discard the nodes in the pattern where the variation of text entropy is equal to zero (constant text in every instance). Since we know that the node containing the content must be present within this set, we aim at identifying this field in the first place. Luckily the content is very simple to identify since it contains the most words, has a very variable length and word count. In the practice, selecting the node having the highest word count average is sufficient. Once the content has been removed from this candidate set, we first check its size. If the size is equal to one, the remaining candidate matches the username. In the case where two candidates are present, we have to distinguish between username and title. Username are very short names, usually one or two words, and is then in average shorter than the title. If two fields are present, the shorter is identified as being the username and the longest is then the title.

[9] `http://joda-time.sourceforge.net/`
[10] `https://github.com/samtingleff/jchronic`

	Pages	Ground Truth	True Positive	False Positive
Global	100	2323	2121	153
Flat	77	1837	1674	125
Nested	23	486	447	26

	Precision	Recall	F_1
Global	93,3 %	91,3 %	92,3 %
Flat	93.1 %	91.1 %	92.1 %
Nested	94.5 %	91.97 %	93.22 %

Table 2. Evaluation : steps one to six

	Content	Username	Date	Title
Occurrence (%)	100	100	97.59	24.48
Correct extraction (%)	100	87.75	83.67	81.63

Table 3. Evaluation, step seven : field identification

5 Mining experiments

This section presents the evaluation protocol of CommentsLifter as well as experiment's results. We first detail the experimental setup, which is a bit particuliar for comments extraction due to the large use of AJAX. Finally we present global and detailed results for both flat and nested cases.

5.1 Setup

Many Web pages handle comments using AJAX, consequently downloading raw HTML along manual ground truth construction is not sufficient for building the dataset. To circumvent this issue we developed two components:

Firefox Extension We implemented a Firefox extension that sends the current DOM (after browser-side Javascript processing) to a Web server.

Web Server The server receives the DOM from the browser through a POST request on a servlet, then runs CommentsLifter and presents an evaluation form along with the extracted comments. The user is asked to evaluate the pertinence of the extraction. We distinguish three cases for the result. We used Jena[11] to generate the SIOC output.

5.2 Evaluation

The results obtained from evaluation are given in Table 2 and Table 3. Table 2 presents extraction results of the pattern mining part (steps 1 to 6 of the

[11] `http://jena.apache.org/`

algorithm) whereas Table 3 presents the field identification results (step 7). The sixth first steps of the algorithm presents very good results, with a global F_1 score over 92%. Concerning field identification we first present the occurrence of the different fields over our dataset. Username and content are always present, while the date is not far from being present in every comment. However titles are to be found in one quarter of the comments. The evaluation accords to the heuristics we describe in section 4.7. Content extraction is a straightforward task since it is easy to "measure" differences with other fields, our algorithm perform perfectly at this task. Date parsing is no easy task, however our algorithm still performs well with an identification rate of 83.67 %. However it is to note that while the rest of the process is language agnostic, date parsing libraries are designed to work with western languages (english, german, french, spanish, ...) but may fail with other languages, especially with non latin alphabet.

6 Conclusions and future work

In this paper, we presented CommentsLifter an algorithm that extracts users' comments and output SIOC data. Our algorithm combines mining induced subtrees from the DOM with simple yet robust heuristics to select the pattern containing the comment as well as for identifying the several fields within the pattern. The empirical evaluation presents very good results, for both extraction and field identification. We successfully extracted comments from various types of Web sites, without *a priori* knowledge, such as online newspapers, forum, user reviews, blogs and we were able to reconstruct the conversations.

Further research will focus on refining the category of the extracted container, in order to determine wether the discussion takes into a forum, Q&A, blog, review area.

References

1. Mark Van Assem, Maarten R. Menken, Guus Schreiber, Jan Wielemaker, and Bob Wielinga. A method for converting thesauri to rdf/owl. In *Proc. of the 3rd Intl Semantic Web Conf. (ISWC04), number 3298 in Lecture Notes in Computer Science*, pages 17–31. Springer-Verlag, 2004.
2. F. Belleau, M.A. Nolin, N. Tourigny, P. Rigault, and J. Morissette. Bio2rdf: towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, 41(5):706–716, 2008.
3. Bettina Berendt, Andreas Hotho, and Gerd Stumme. Towards semantic web mining. In *Intl Semantic Web Conf. (ISWC02)*, pages 264–278. Springer, 2002.
4. J. Breslin, A. Harth, U. Bojars, and S. Decker. Towards semantically-interlinked online communities. *The Semantic Web: Research and Applications*, pages 71–83, 2005.
5. D. Cai, S. Yu, J.R. Wen, and W.Y. Ma. Extracting content structure for web pages based on visual representation. In *Proceedings of the 5th Asia-Pacific web conference on Web technologies and applications*, pages 406–417. Springer-Verlag, 2003.

6. C.H. Chang, M. Kayed, R. Girgis, and K.F. Shaalan. A survey of web information extraction systems. *Knowledge and Data Engineering, IEEE Transactions on*, 18(10):1411–1428, 2006.
7. Y. Chi, R.R. Muntz, S. Nijssen, and J.N. Kok. Frequent subtree mining-an overview. *Fundamenta Informaticae*, 66(1-2):161, 2005.
8. P. Cimiano and J. Völker. Text2onto. *Natural Language Processing and Information Systems*, pages 257–271, 2005.
9. M. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. Mitchell, K. Nigam, and S. Slattery. Learning to construct knowledge bases from the world wide web. *Artificial Intelligence*, 118(1):69–113, 2000.
10. S. Fernández, D. Berrueta, and J.E. Labra. Mailing lists meet the semantic web. In *Proc. of the BIS 2007 Workshop on Social Aspects of the Web, Poznan, Poland*, 2007.
11. A. Gentile, V. Lanfranchi, S. Mazumdar, and F. Ciravegna. Extracting semantic user networks from informal communication exchanges. *The Semantic Web–ISWC 2011*, pages 209–224, 2011.
12. D. Huynh, S. Mazzocchi, and D. Karger. Piggy bank: Experience the semantic web inside your web browser. *The Semantic Web–ISWC 2005*, pages 413–430, 2005.
13. C. Kohlschütter, P. Fankhauser, and W. Nejdl. Boilerplate detection using shallow text features. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 441–450. ACM, 2010.
14. C. Kohlschütter and W. Nejdl. A densitometric approach to web page segmentation. In *Proceeding of the 17th ACM conference on Information and knowledge management*, pages 1173–1182. ACM, 2008.
15. S.H. Lin and J.M. Ho. Discovering informative content blocks from web documents. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 588–593. ACM, 2002.
16. B. Liu, R. Grossman, and Y. Zhai. Mining data records in web pages. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–606. ACM, 2003.
17. A. Maedche and S. Staab. Ontology learning for the semantic web. *Intelligent Systems, IEEE*, 16(2):72–79, 2001.
18. D.D.C. Reis, P.B. Golgher, A.S. Silva, and AF Laender. Automatic web news extraction using tree edit distance. In *Proceedings of the 13th international conference on World Wide Web*, pages 502–511. ACM, 2004.
19. F. Sun, D. Song, and L. Liao. Dom based content extraction via text density. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information*, pages 245–254. ACM, 2011.
20. A. Yates, M. Cafarella, M. Banko, O. Etzioni, M. Broadhead, and S. Soderland. Textrunner: open information extraction on the web. In *Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 25–26. Association for Computational Linguistics, 2007.
21. M.J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *Knowledge and Data Engineering, IEEE Transactions on*, 17(8):1021–1035, 2005.
22. Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *Proceedings of the 14th international conference on World Wide Web*, pages 76–85. ACM, 2005.