URL Tree: Efficient Unsupervised Content Extraction from Streams of Web Documents

Borut Sluban
Department of Knowledge Techologies
Jožef Stefan Institute
Jamova 39, Ljubljana, Slovenia
borut.sluban@ijs.si

Miha Grčar
Department of Knowledge Techologies
Jožef Stefan Institute
Jamova 39, Ljubljana, Slovenia
miha.grcar@ijs.si

ABSTRACT

The Web represents the largest, and an increasingly growing, source of information. Extracting meaningful content from Web pages presents a challenging problem, already extensively addressed in the offline setting. In this work, we focus on content extraction from streams of HTML documents. We present an infrastructure that converts continuously acquired HTML documents into a stream of plain text documents. The presented pipeline consists of RSS readers for data acquisition from different Web sites, a duplicate removal component, and a novel content extraction algorithm which is efficient, unsupervised, and language-independent. Our content extraction approach is based on the observation that HTML documents from the same source normally share a common template. The core of the proposed content extraction algorithm is a simple data structure called *URL Tree*. The performance of the algorithm was evaluated in a stream setting on a time-stamped semi-automatically annotated dataset which was made publicly available. We compared the performance of URL Tree with that of several open source content extraction algorithms. The evaluation results show that our stream-based algorithm already starts outperforming the other algorithms after only 10 to 100 documents from a specific domain.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Trees; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering, Retrieval models*; I.7 [**Document and Text Processing**]: Miscellaneous

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

content extraction; boilerplate removal; stream data; Web content; unsupervised learning

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2505654.

1. MOTIVATION AND RELATED WORK

The Web has become the largest source of information and heterogeneous data. The data is available in different representations (texts, graphs, knowledge stores, databases, time series, etc.), languages, and sizes (the concept of *big data* is becoming more and more important), and can have different dynamics (static data, slow or fast-paced streams). Considering only news sites, blogs, and social media, new content is produced continuously at an increasing pace. From this perspective, we can view the Web as a generator of data streams that contain valuable information that is often hard to discover due to the information overload problem. Managing such amounts of unstructured streaming data in terms of information and knowledge discovery requires satisfying, among other things, two major conditions: (i) efficient online processing with near-real time information delivery and (ii) differentiation between the relevant content in HTML documents and the accompanying text called the boilerplate (template and navigation items, recommendations, advertisements, copyright notices, etc.).

Automated content extraction, template identification, or boilerplate removal for a general Web page proves to be a challenging task and has attracted much attention in the scientific literature and industry. Early approaches were mostly based on handcrafted rules and could only be applied to Web pages from a limited number of sources. The main drawback of these approaches was the inability to easily adapt to the changes that occur over time in the HTML structure of Web pages. In contrast, template detection algorithms [1] aim at finding invariant and changing sections of Web pages automatically. In [8], the Document Object Model (DOM) trees of a set of Web pages are analyzed to find the optimal mapping between the tree nodes and thereby identify the content nodes. In [6], the cooccurence of the terms in a set of Web pages is analyzed and the entropy of each text block is computed. The informative text blocks are then identified by thresholding. The approach in [2] excludes template items by detecting similar content with a common layout style during the index building process of a search engine. More recent approaches mainly focus on textual features. *NCleaner* [3] uses a character-level n-gram language model to distinguish between, as they call it, clean and dirty text. The *boilerpipe* algorithm [4] describes text blocks with "shallow text features" and builds a decision tree that is used to classify the text blocks of an arbitrary Web page as content or boilerplate. *jusText* [7] implements a set of rules that characterize text blocks as either "good" or "bad". The algorithm first performs a context-free clas-

sification of text blocks and then refines it with a set of context-aware rules. *Readability*,[1] a popular reading tool, manipulates the DOM tree of a Web page by employing several heuristics based on textual and structural features to remove the boilerplate. More exhaustive overviews of the content extraction field can be found in [7] and [5].

In contrast to the existing methods which are mostly supervised, language-dependent, and/or unaware of the stream setting, we propose an efficient, online, unsupervised, language-independent approach to content extraction from streams of HTML documents.

The main contributions of this work are as follows. First, we describe an infrastructure for content extraction from a continuous stream of documents from different Web sites (see Section 2). Second, we propose a URL normalization procedure for identifying duplicates (see Section 3). Third, we present the URL Tree data structure and the corresponding stream-based content extraction algorithm (see Section 4). Fourth, we compare the performance of the proposed algorithm to 10 different open source algorithms (see Section 5). Finally, we publish the time-stamped semi-automatically annotated dataset that we used for the evaluation purposes. With this, we provide the essential means for further research in this area. Several ideas for further work are presented in Section 6.

2. STREAM SETTING

Most of the content extraction algorithms, incl. the methods presented in Section 1, extract content in an offline setting. By "offline setting" we refer to extracting the relevant text from each HTML document separately, independently of any other acquired documents, except perhaps using a static labeled dataset in the model training phase. Such content extraction algorithms do not take into account that a subset of the previously acquired documents potentially shares the same template and thus the same boilerplate texts that are part of this template. Our approach makes use of the fact that documents are continuously being acquired by RSS readers, which generate sub-streams of documents with the same template. The template can thus be identified as a set of repeatedly observed text segments in a sub-stream and therefore separated from the main content. We base our work on the assumption that it is easier to distinguish between relevant and irrelevant content given the history of documents from the same source. This is analogous to retouching a damaged movie frame as opposed to retouching a damaged photo: the first task is easier because fragments from the preceding frame can be used. The content extraction process on a stream of HTML documents is illustrated in Figure 1.

In our workflow, HTML documents are acquired by *RSS readers*,[2] which periodically check the corresponding Web sites for the most recent RSS documents. An RSS document is essentially an XML containing titles and short descriptions (summaries) of a certain number of the most recent posts. Each RSS item also provides a link (i.e., URL) to the Web page containing the full content and the RSS reader is responsible for downloading the corresponding HTML document. In our case, one RSS reader is acquiring documents from one Web site (e.g., BBC at `http://www.bbc.co.uk`) through one or more RSS feeds provided by the site (e.g., BBC lists its RSS feeds at `http://www.bbc.co.uk/news/10628494`). Each RSS reader keeps the history of acquired documents by representing each document with a hash code computed out of the document's title, description, and publication date as observed in the RSS document. With this, the RSS reader avoids downloading the same document multiple times when processing subsequent RSS documents from the same site.

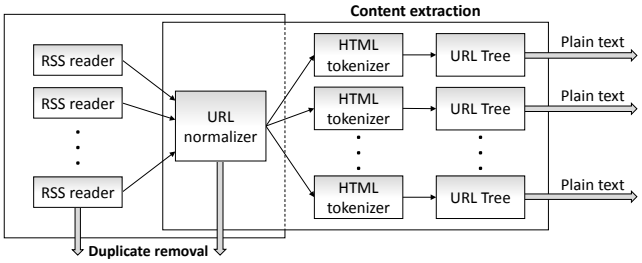

Figure 1: The content extraction process on streams of Web documents.

Multiple copies of the same document that were not identified by this caching mechanism in the RSS readers are filtered out by the *URL normalization* component. This process is more thoroughly described in Section 3.

The *HTML tokenization* component removes HTML markup, CSS styles, and JavaScript code, and extracts text blocks from an HTML document. Text blocks are defined as chunks of text delimited with at least one (opening or closing) HTML tag. Certain inline HTML tags (such as *em*, *strong*, *i*, *font*, *a*, etc.) are ignored when partitioning textual content into text blocks.

The content and boilerplate of an HTML document are identified by employing the proposed *URL Tree* data structure which is the core of our content extraction algorithm. This process is more thoroughly discussed in Section 4.

3. URL NORMALIZATION

In the content extraction pipeline, news content is acquired from different Web sites (e.g., `http://www.reuters.com`, `http://www.bbc.co.uk`, etc.) through their RSS feeds. The RSS feeds provide references to the latest news articles or blog posts (HTML documents) in the form of *Uniform Resource Locators* (URLs). Based on the response URL,[3] a document belongs to a specific domain. By the *domain* of an HTML document, we refer to the combination of the top level domain (TLD) and the second-level domain name of the document's (response) URL, e.g., `news.eu` would be the domain of the HTML document at `http://www.news.eu/politics/europe/article1.html`. Note that documents acquired from the same Web site can in fact originate from different domains.

Although collecting duplicate content from a single Web site is avoided by the caching mechanism in the RSS reader, some HTML documents from the same domain may still be acquired multiple times (within-domain duplicates). One

[1] `http://www.readability.com/developers/api/reader`

[2] In our case, an RSS reader is a software component designed to retrieve content updates from a particular Web site. RSS stands for Really Simple Syndication.

[3] The *response* URL is obtained from a *request* URL after the redirections (if any) have been resolved.

reason is that the same HTML documents are served over multiple RSS feeds covering different topics of interest (e.g., politics and economy). The caching mechanism in the RSS reader, relying on the metadata contained in an RSS XML document (in our case, document title, description, and publication date), often receives different metadata from different RSS feeds even though they reference the same HTML document. Another reason is news aggregators that reference documents from various different sites/domains (e.g., Google News collects news from BBC, CNN, and many other news sites). The same document can thus be acquired through an aggregator (e.g., by the Google News RSS reader) and through its publisher's RSS feed (e.g., by the BBC RSS reader).

As already said, RSS feeds provide references to HTML documents in the form of URLs. This would seem to imply that within-domain duplicates can be easily avoided, simply by observing these URLs. However, it turns out that a URL of an HTML document, in its raw form, is not a unique identifier of that document. Consequently, it cannot be used as a tool to properly detect and avoid duplicates in the data acquisition process, but it is nevertheless a good starting point. Understanding how URLs are composed and how they are used enables us to construct a "nearly unique" document identifier.

A URL is usually composed in the following way:

```
protocol://domainname/path1/.../pathN/file?query
```

For example:

```
http://www.ft.com/world/2013/02/news.html?feed=3
```

In the following, we present a URL normalization process which results in a URL-based nearly-unique identifier of an HTML document, called the *URL key*. Each step of the process decreases the differentiation between documents and thereby increases the ability to avoid collecting duplicated content. The complete URL normalization process is as follows (input: a request URL as given in an RSS XML document):

1. Follow the redirections to convert the request URL into the corresponding response URL.

   Example request URL:

   ```
   http://news.google.com/news/url?sa=t&fd=R&usg=
   AFQjCNHEIIAoeLGPfbkX6IdaQ2xoYptq-w&url=http://
   abcnews.go.com/kabc/story?section%3Dnews/local/
   los_angeles%26id%3D8691010
   ```

   The corresponding response URL:

   ```
   http://abcnews.go.com/kabc/story?section=news/
   local/los_angeles&id=8691010
   ```

2. Convert the title of the corresponding document, as given in the RSS XML document, into a 128-bit hash code and append it as a query parameter to the URL, specifically: _cid_=<hash code>. In addition, normalize the encoding and sort the query parameter names alphabetically in the query part.

   Example (URL after Step 2):

   ```
   http://abcnews.go.com/kabc/story?_cid_=090
   c0a49e55c9734bc2311e08ff006d4&id=8691010
   &section=news/local/los_angeles
   ```

3. Normalize the query with the query normalization rules. For each rule in the rule list, check if it is applicable to the URL. A rule is defined with a regular expression and a list of query parameters (usually just one) that need to be retained. Several examples are given in Table 1. A rule is applicable if the regular expression matches the URL. In that case, the query parameters are filtered according to the rule. If none of the rules apply, the query part is dropped completely.

   Example (URL after Step 3):

   ```
   http://abcnews.go.com/kabc/story?id=8691010
   ```

Table 1: URL normalization rule examples.

Regular expression (trigger)[4]	Query parameter to retain
abcnews\.go\.com	id
www\.fitchratings\.com.*?/detail\.cfm	pr_id
bbs\.chinadaily\.com\.cn/viewthread\.php	tid
www\.hurriyetdailynews\.com/n\.php	n
globeandmail\.golfcanada\.ca	articleId
podcast\.ft\.com/index\.php	pid
www\.aljazeera\.com(\?.*)?$	_cid_
www\.dailymail\.co\.uk/home/index\.html	_cid_
www\.foxnews\.com/on-air.*?/index\.html	_cid_

The resulting URL (after Step 3) represents the corresponding URL key. In Figure 2, we show how URL normalization contributes to the identification of duplicates, solely by comparing URL keys of documents. The presented results were computed on 569,583 documents collected by our data acquisition pipeline described in Section 2. The documents were collected from 31 Web sites over a period of eight weeks (Oct 24 – Dec 19, 2011).

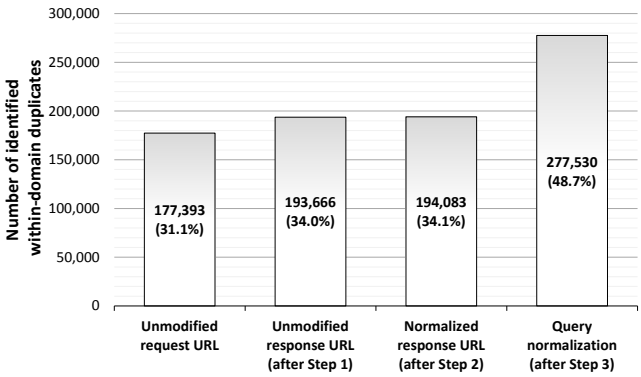

Figure 2: Number of identified within-domain duplicates after each step of the URL normalization process.

The results show that a significant amount of documents (almost 50%) can be filtered out in the data acquisition process if we apply URL normalization. The removal of within-domain duplicates is essential for content extraction with a URL Tree as it prevents perceiving content blocks as boilerplate.

[4]Due to space limitation, the start of each regular expression "http://" is omitted.

4. CONTENT EXTRACTION ALGORITHM

In our approach, we assume that each HTML document with a unique URL key represents a unique document. Our content extraction algorithm is based on the observation that documents from the same domain, whose URL keys differ only in the document identifier, have a lot of boilerplate in common. To determine which text blocks are boilerplate, we count their occurrences in the stream. We store the occurrence counts in the URL Tree structure. When a new document arrives in the stream and passes the duplicate removal filter, its URL key is mapped to a branch in a URL Tree. If the branch does not yet exist in the tree, it is created. The nodes in the branch (i.e. parts of the URL key) hold statistics about the text blocks extracted from the document.

For a newly observed document, each node in the branch is updated as follows: (i) the number of observed documents (n_s) is increased by 1, (ii) each text block is normalized (all non-alphabetic characters are removed from the string, and the string is made all-lowercase) and converted into an MD5 hash code (a set of unique text block hash codes $\mathbf{B} = \{b_i\}$ is used from here on), and (iii) the counter for each of the text blocks in $\mathbf{B}$ (let us denote it with $\mathrm{c}(b_i)$) is increased by 1. The URL Tree construction process is illustrated in Figure 3.

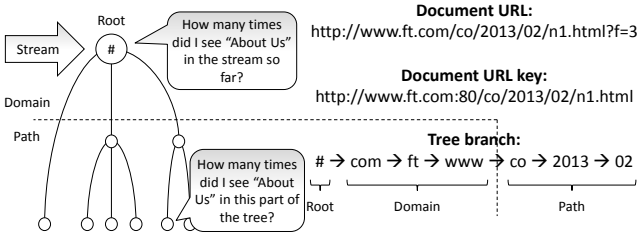

Figure 3: Constructing a URL Tree.

After the URL Tree has 'seen' enough documents at a specific node, that node can be used as a classification model to distinguish between content and boilerplate. The classification process can employ different heuristics. The default heuristic is as follows (input: a document and its URL key):

- The leaf corresponding to the URL key is identified in the URL Tree (note that the leaf always exists because the document is inserted into the tree before its text blocks are classified) and selected for the classification process.
- If the number of observed documents (i.e., support) in the selected node is less than the predefined threshold, i.e., $n_\mathrm{s} < n_\mathrm{min}$, the tree is traversed towards the root until a node satisfying $n_\mathrm{s} \geq n_\mathrm{min}$ is found and selected. If such a node cannot be found, the root node is selected.
- For each text block extracted from the document, the statistics at the selected node are examined to determine whether the text block is content or boilerplate as follows:
  - The text block is normalized (all non-alphabetic characters are removed from the string, and the string is made all-lowercase) and converted into an MD5 hash code. This hash code serves as the key for retrieving the corresponding counter $\mathrm{c}(b_i)$.
  - If the counter $\mathrm{c}(b_i)$ exceeds the predefined threshold c_max, i.e., $\mathrm{c}(b_i) > c_\mathrm{max}$, then the text block is classified as boilerplate. Otherwise, it is classified as content.

5. EVALUATION

Unlike the content extraction algorithms that work in offline settings, our URL Tree-based algorithm is designed to work efficiently on real-time streams of Web documents and to benefit from the continuous inflow of new (evolving) data. In our experiments, we used a time-stamped dataset of Web documents and compared the performance of our URL Tree-based algorithm to 10 different content extraction algorithms developed within four open source projects.

5.1 Dataset

The experiments were performed on a stream of HTML documents acquired from 31 Web sites over the period from Oct 24 to Dec 19, 2011. The initial stream of 569,583 documents was reduced to 292,053 documents after within-domain duplicate removal (URL normalization) as presented in Section 3. This dataset is a part of the data acquired during the European project FIRST.[5]

A total of 56,436 documents, sampled from the beginning (Oct 24, 2011 – Oct 31, 2011), the middle (Nov 10, 2011 – Nov 30, 2011) and the end of the stream (Dec 10, 2011 – Dec 19, 2011), were annotated with manually designed regular expressions tailored for specific Web site templates. The annotated dataset is available for download and preview at `http://first.ijs.si/urltreedataset`.

5.2 Performance measures

We measured the performance of content extraction with basic measures from information retrieval, namely recall (R), precision (P), and their harmonic mean, the F_1-score. We computed these measures for each annotated document as follows:

$$R_i = \frac{|\text{extracted relevant text}|}{|\text{all relevant text}|} \tag{1}$$

$$P_i = \frac{|\text{extracted relevant text}|}{|\text{all extracted text}|} \tag{2}$$

$$F_{1i} = \frac{2 \cdot P_i \cdot R_i}{P_i + R_i} \tag{3}$$

The quantity of "extracted relevant text" in Eq. 1 and 2 was computed with *DiffLib*,[6] a software library that enabled us to compute the size of the intersections between the text labeled as content in the annotated dataset and the different text outputs produced by the employed algorithms. Each text is transformed into a sequence of words/tokens and the length of the longest common subsequence is the length of the extracted relevant text.

In the stream setting, we present the performance of an algorithm as its overall performance until and including the currently processed document j (also referred to as the cumulative moving average):

$$\overline{F_1} = \frac{1}{j} \sum_{i=1}^{j} F_{1i} \; . \tag{4}$$

[5] `http://www.project-first.eu`
[6] `http://difflib.codeplex.com`

5.3 Open source algorithms

We evaluated 10 different algorithms from four open source projects and compared their performance with the proposed URL Tree algorithm. The selected algorithms are listed in Table 2.

Table 2: Open source algorithms.

Algorithm	Description
Boilerpipe[7] [4]	
- DE	The default extractor based on a decision tree and shallow text features.
- AE	An extractor tuned towards news articles.
- ASE	Extracts only whole sentences.
- LCE	Extracts the largest text block.
- NWRE	An extractor based on predefined word-count heuristics.
jusText[8] [7]	Designed to preserve mainly text containing full sentences.
NCleaner[9] [3]	
- default	Uses character-level n-gram models as classifiers.
- non-lexical	Relies on non-lexical text features.
Readability[10]	
- .NET	A .NET (C#) port of Readability.
- Python	A Python port of Readability.

5.4 URL Tree classification heuristics

To extract content with URL Tree, we used four different classification heuristics:

Strict: The default heuristic (discussed in Section 4) with parameters $n_{\min}$ and $c_{\max}$ set to 5 and 1, respectively.

Strict-support-N: Similar to the strict heuristic, except that only the nodes with support n_s greater than N are selected when traversing a URL Tree branch.

Strict-at-domain: The strict heuristic, always applied at the domain node.

Relaxed-at-domain-N: The strict heuristic with an additional rule: If the support in the corresponding domain node is greater than N, then the threshold for content is increased to 2, i.e., text blocks with $c(b_i) > 2$ are declared as boilerplate and blocks with $c(b_i) \leq 2$ as content.

5.5 Results

We conducted two separate experiments. In the first experiment we streamed our dataset of 292,053 HTML documents (see Section 5.1) through the URL Tree content extractor employing the strict classification heuristic. We measured $\overline{F_1}$ at each of the 56,436 annotated documents (i.e., gold standard), for each of the 33 domains separately. From the results presented in Figure 4 we see that the performance of URL Tree, as expected, increased over time. Furthermore, for the majority of the domains, URL Tree achieved good F_1 scores already after 10 to 100 documents from an individual domain. $\overline{F_1}$ scores higher than 0.7 were observed for all the domains, scores higher than 0.84 for three quarters of the domains, and for more than a third of the domains, the $\overline{F_1}$ scores exceeded 0.9. The best scores were achieved for the domains `chosun.com`, `usatoday.com`, and `newyorker.com`, whereas the lowest scores were obtained for the domains `abcnews.go.com`, `cbsnews.com`, and `foxnewsinsider.com`.

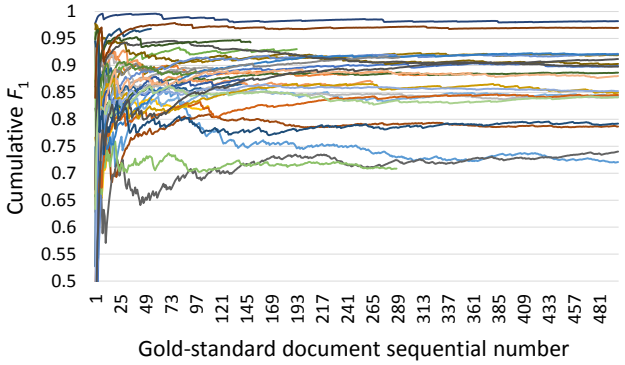

Figure 4: URL Tree content extraction evaluation results for each domain separately.

In the second experiment, we compared the performance of URL Tree with that of the 10 selected open source content extractors (see Section 5.3). We employed the four heuristics discussed in Section 5.4. The first 35,321 documents (which included the first 10,000 gold-standard documents) were used as the tuning set to select the best value for N for the two parametrized heuristics. We varied N from 100 to 1,000 (by 100). According to our observation on the tuning set, N was set to 100 for the strict-support-N heuristic (termed *strict-support-100*) and to 500 for the relaxed-at-domain-N heuristic (termed *relaxed-at-domain-500*). With these settings, the experiment was re-run on the rest of the stream (i.e., the test set).

The results are given in Figure 5. The figure shows the $\overline{F_1}$ scores aggregated over all the domains. The best performing open source algorithms, in our specific experimental setting, were the two Readability implementations ($\overline{F_1}$ of approximately 0.84), followed by the five different flavors of Boilerpipe ($\overline{F_1}$ ranging from 0.76 to 0.81).

We can see that URL Tree outperformed the other algorithms relatively early in the process. The $\overline{F_1}$ score of the strict heuristic reached over 0.85 after about 4,000 documents (which corresponded to around 1,000 gold-standard documents) and remained over this value throughout the rest of the stream. It also proved beneficial to experiment with several different heuristics. The heuristics strict-support-100 and strict-at-domain both outperformed the strict heuristic and exhibited comparable results with $\overline{F_1}$ scores around 0.86. Furthermore, the relaxed-at-domain-500 heuristic clearly outperformed the other three heuristics, with an $\overline{F_1}$ score of about 0.89.

6. DISCUSSION

In this paper, we discussed a URL Tree-based approach to content extraction from streams of HTML documents. The presented approach is efficient (in terms of processing speed), unsupervised, language-independent, and outperforms the other algorithms we evaluated in our RSS data

[7] `https://code.google.com/p/boilerpipe`
[8] `https://code.google.com/p/justext`
[9] `http://sourceforge.net/projects/webascorpus/files/NCleaner/NCleaner-1.0`
[10] `http://code.google.com/p/nreadability`, `https://github.com/buriy/python-readability`

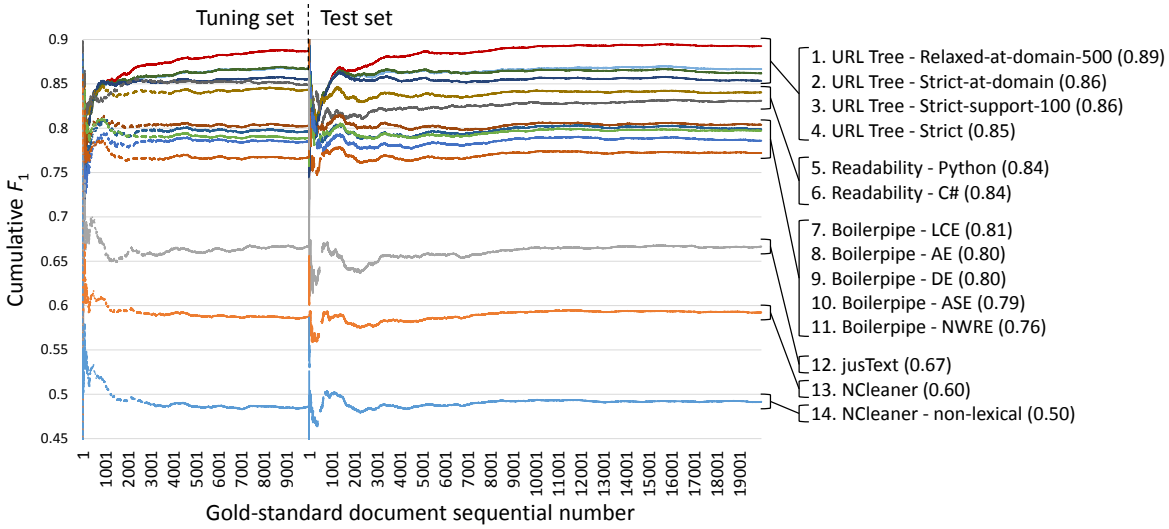

Figure 5: Stream-based content extraction evaluation results. We limit the view to the first 20,000 test documents. The final $\overline{F_1}$ scores (after all the 46,436 test documents) are given in the brackets.

acquisition setting. However, it has some drawbacks as well. One of its weaknesses is that it cannot be applied to small, diverse datasets of HTML pages. It is specifically designed to work in scenarios such as our RSS data acquisition pipeline. It could potentially also be applied to crawling large and dynamic Web sites. In this case the URL normalization rules would need to be changed in order to get the value for the _cid_ parameter from the HTML header rather than the RSS metadata. Another thing to point out is the memory consumption. Our implementation of URL Tree consumes around 6,5 MB of RAM for every 1,000 inserted documents. Even though this aspect was completely ignored in this paper, it needs to be handled in real-life applications.

We have employed a URL Tree-based content extractor in a data acquisition pipeline running continuously since April 2011. So far, we have collected and preprocessed over 15 million unique documents from 175 Web sites providing altogether around 2,500 RSS feeds. Let us also briefly note that the stream of plain text that the pipeline produces is used as an input for a range of analytical components that look for correlations between occurrences of extracted entities and different financial indicators such as credit default swaps and stock market indices. To handle the memory consumption issue, the employed instance of URL Tree holds a maximum of 10,000 documents for each domain. In addition, it removes documents older than 14 days (however, it always keeps at least the 100 most recent documents). This mechanism was defined rather ad-hoc and we plan to investigate the impact of different strategies on the accuracy more thoroughly. Our preliminary results show that removing outdated documents from the tree does not affect the accuracy.

In future work, we plan to explore how URL Tree and several other content extractors perform on different types of boilerplate (advertisements vs. menu items vs. copyright notices etc.). Furthermore, we plan to explore whether it would be better to use a similarity hashing scheme rather than representing text blocks with MD5 hash codes. Last but not least, we plan to explore the possibility of updating the URL normalization rule list in a semi- or fully automatic manner.

7. ACKNOWLEDGMENTS

We would like to thank Marko Brakus for creating the initial annotated dataset. This work was partially funded by the Slovenian Research Agency and by the European Commission in the context of the FP7 projects FIRST and FOC, under the grant agreements no. 257928 and 255987, respectively.

8. REFERENCES

[1] Z. Bar-Yossef and S. Rajagopalan. Template detection via data mining and its applications. In *Proc. of the Int. Conf. on World Wide Web*, pages 580–591, 2002.

[2] L. Chen, S. Ye, and X. Li. Template detection for large scale search engines. In *Proc. of the ACM Symposium on Applied Computing*, pages 1094–1098, 2006.

[3] S. Evert. A lightweight and efficient tool for cleaning web pages. In *Proc. of the Int. Conf. on Language Resources and Evaluation (LREC)*, 2008.

[4] C. Kohlschütter, P. Fankhauser, and W. Nejdl. Boilerplate detection using shallow text features. In *Proc. of the ACM Int. Conf. on Web Search and Data Mining*, pages 441–450, 2010.

[5] T. Kovačič. Evaluating Web Content Extraction Algorithms. Bachelor's Thesis, Faculty of Computer and Information Science, University of Ljubljana, Slovenia, 2012.

[6] S.-H. Lin and J.-M. Ho. Discovering informative content blocks from web documents. In *Proc. of the ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 588–593, 2002.

[7] J. Pomikálek. *Removing Boilerplate and Duplicate Content from Web Corpora*. PhD thesis, Faculty of Informatics, Masaryk University, Brno, Czech Republic, 2011.

[8] K. Vieira, A. S. da Silva, N. Pinto, E. S. de Moura, J. a. M. B. Cavalcanti, and J. Freire. A fast and robust method for web page template detection and removal. In *Proc. of the ACM Int. Conf. on Information and Knowledge Management*, pages 258–267, 2006.