BlogForever Crawler: Techniques and Algorithms to Harvest Modern Weblogs

Olivier Blanvillain
École polytechnique fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
olivier.blanvillain@epfl.ch

Nikos Kasioumis
European Organization for Nuclear Research (CERN)
1211 Geneva 23, Switzerland
nikos.kasioumis@cern.ch

Vangelis Banos
Department of Informatics
Aristotle University of Thessaloniki, Greece
vbanos@gmail.com

ABSTRACT

Blogs are a dynamic communication medium which has been widely established on the web. The BlogForever project has developed an innovative system to harvest, preserve, manage and reuse blog content. This paper presents a key component of the BlogForever platform, the web crawler. More precisely, our work concentrates on techniques to automatically extract content such as articles, authors, dates and comments from blog posts. To achieve this goal, we introduce a simple and robust algorithm to generate extraction rules based on string matching using the blog's web feed in conjunction to blog hypertext. This approach is leading to a scalable blog data extraction process. Furthermore, we show how we integrate a web browser into the web harvesting process in order to support the data extraction from blogs with JavaScript generated content.

General Terms

Algorithms, Software Engineering

Keywords

Web archiving, Web crawling, ...

1. INTRO

1. Introduce the importance for blog preservation
2. Explain the difficulties in harvesting blogs
3. Explain why open source and Invenio
4. Introduce the blog crawler

Challenges consist of:

1. Providing a high degree of automation
2. Dealing with large volumes of data

Comment: general introduction to web archiving and web crawlers

Web archiving is the process of harvesting and gathering web content in order to safely preserve it for posterity. As the volume and importance of the information on the World Wide Web increases, web archiving becomes more and more relevant and its importance becomes clearer.
Web crawlers are an essential part of web archiving, allowing for automated capturing of these resources.
An efficient and accurate web crawler is the first crucial step to successful digital preservation.

Comment: introduction to blogs, their nature and their significance

An estimate xx% of current web content is blogs (reference).
OR There is an estimate of xxxxx blogs online currently.
A blog is a *frequently updated web site consisting of personal observations, excerpts from other sources, etc., typically run by a single person, and usually with hyperlinks to other sites; an online journal or diary.*
Comment: http://www.oed.com/view/Entry/256743#eid11173991
Given the nature of blogs, they are expectedly dynamic and highly volatile web resources.

Blogs are nowadays a popular means of communication and expression of ideas and have been adopted by universities, institutions and scolars (reference?).
Blogs have recenlty been used to disseminate ideas in times of political turmoil or even war. (examples, Egypt, Syria etc)
Their political, scientific and cultural significance is undisputed. (rephrase to avoid need for reference?)

However, to this day there is no standard method or authority that ensures blog archiving and xx% dissapear every day (reference).
Comment: A Tale of a Disappearing Website : http://blogs.loc.gov/digitalpre
tale-of-a-disappearing-website/
Comment: Considerations for the Preservation of Blogs :
http://www.digitalpreservationeurope.eu/publications/briefs/preservartion_b
Therefore, it is crucial that the necessary foundations for blog archiving are put in place.

Comment: present background in blog archiving and constraints

The blogosphere is a massive web resource of unstructured data.
Identifying, harvesting and parsing that data can be a very challenging task.
Challenges and constraints: size of blogosphere, heterogeneous nature of different publishing platforms, ever-changing structure, dynamically created content using modern web technologies.

complexity in parsing blogs from unstructured data to structured information.
programmatic access to blog content unavailable
unpredictable publishing rate –> scalability

Comment: present issues in current solutions
Is this needed? Can write in "Related Work" instead.
Comment: present our main ideas and principles
Web feeds
HTML + XPATH
efficient string similarity
JavaScript

Comment: present our statement and target
improve efficiency, accuracy and general quality to help with archiving and understanding, using (information retrieval).
Comment: present the contents of the paper
In chapter 1, chapter 2....
Notes:

(from wikipedia) The largest web archiving organization based on a bulk crawling approach is the Internet Archive which strives to maintain an archive of the entire Web. National libraries, national archives and various consortia of organizations are also involved in archiving culturally important Web content. Commercial web archiving software and services are also available to organizations who need to archive their own web content for corporate heritage, regulatory, or legal purposes.

Comment: Aims and contributions of this work. Example from my last paper to see the format: http://purl.pt/24107/1/iPres2013_PDF/CLEAR%20a%20credible%20method%20to%20evaluate%20website%20archivability.pdf
Comment: Semi-structured nature of Web pages
Labeled ordered rooted trees
XML Path Language(XPath)

2. RELATED WORK

Web archiving is a challenging and very interesting problem and has been the subject of many studies as well as the goal of several projects. A fraction of these focuses on the specific problem of blog harvesting and archiving. These studies and projects focus on the unique nature of blogs as dynamic web resources with a usually common set of attributes. Each entry in a blog, commonly called a post, has a title, an author, a publication date, some content and possibly some descriptive tags. Blogs are also strongly associated with web feeds, using technologies such as RSS and Atom (expand and reference). Web feeds are structured documents (often XML-based) that advertize and provide access to a website's content. Given their format, web feeds constitute an excellent source for blog archiving. They do, however, present some problems: (a) web feeds contain only the latest posts of a blog (usually 20) and (b) web feeds may only provide a summary or a fraction of the entire content of each blog post. More advanced methods and techniques have to be employed in order to achieve quality blog archiving. In this section we review some studies and projects that introduce and describe such techniques. *Comment: Maybe this is repetitive, some of these things should already have been mentioned in the introduction*

Comment: list the projects and compare them to what we do

Web Object Identification for Web Automation and Meta-Search
http://www.dbai.tuwien.ac.at/proj/tamcrow/download/Kordomatis2013W
In this paper we deal with the problem of finding specific web objects on previously unseen web pages. The problem consists of the sub problems of how to describe the desired objects (e.g., a specific submit button)and how to identify them on new web pages. This general problem is fundamental to many fields of research like Web Data Extraction, Web Form Understanding, Web Automation, etc. Our approach uses features based on the visual perceivable characteristics of web objects for object description and machine learning techniques for object identification. Applied to a meta-search setting, this allows for a very universal solution. Here, the common elements of the considered search forms can be identified on previously unseen search pages, as long as they follow the same design principles.

Self-supervised Automated Wrapper Generation for Weblog Data Extraction
https://github.com/OlivierBlanvillain/blogforever-crawler-publication/raw

Web Data Extraction, Applications and Techniques: A Survey
http://www.emilio.ferrara.name/wp-content/uploads/2011/07/survey-csur.pdf

Archiving Data Objects using Web Feeds
http://hal.archives-ouvertes.fr/docs/00/53/79/62/PDF/iwawienna.pdf

Intelligent and Adaptive Crawling of Web Applications for Web Archiving
http://pierre.senellart.com/publications/faheem2013intelligent.pdf

Zero-cost Labelling with Web Feeds for Weblog Data Extraction
http://www2013.org/companion/p73.pdf

OXPath: A Language for Scalable, Memory-efficient Data Extraction from Web Applications
http://www.vldb.org/pvldb/vol4/p1016-furche.pdf

3. ALGORITHMS

This section explains in detail the algorithm we developed to extract blog posts articles *Comment: should we explain here that article is the actual content (text) of a blog post? Or is it already clear enough?*
TODO: I don't know. I used "blog post content" as "article + title + author + date + comments" in other sentences but it might be good to say somewhere that "blog post article" refers to the "core textual content of a blog post". as well as its variations for authors, dates and comments. Our approach uses blog specific characteristics to build extraction rules applicable to all posts of a blog *Comment: "all posts of a blog" needs a better formulation.*
Our focus is on minimizing the algorithmic complexity while keeping our approach simple and generic.

3.1 Motivation

Extracting metadata and content from HTML documents is a challenging task. Standards and format recommendations have been around for quite some time, strictly specifying how HTML documents should be organised [20]. For instance the `<h1></h1>` tags have to contain the highest-level heading of the page and must not appear more than once per page [19]. More recently, specifications such as microdata [21] define ways to embed semantic information and metadata inside HTML documents, but these still suffer from very low usage: estimated to be used in less than 0.5% of websites [17]. In fact, the majority of websites rely on the generic `<span></span>` and `<div></div>` container elements with custom `id` or `class` attributes to organise the structure of pages [22], and more than 95% of pages do not pass HTML validation [23]. Under such circumstances, relying on HTML structure to extract content from web pages is not viable and other techniques need to be employed.

Having blogs as our target websites, we made the following observations which play a central role in the extraction process[1]:

(1) Blogs provide web feeds: structured and standardized views of the latest posts of a blog,
(2) Posts of the same blog share a similar HTML structure.

Web feeds usually contain about 20 blog posts [14], often less than the total number of posts in blogs. Consequently, in order to effectively archive the entire content of a blog, it is necessary to download and process pages beyond the ones referenced in web feed.

3.2 Content extraction

To extract content from blog posts, we proceed by building extraction rules from the data given in the blog's web feed. The idea is to use a set of *training data*, pairs of HTML pages and target content, which are used to build an extraction rule capable of locating the target content on each HTML page.

Observation (1) allows the crawler to obtain input for the extraction rule builder algorithm: each web feed entry contains a link to the corresponding web page as well as target content such as the article, its title, authors and publication date. Observation (2) guarantees the existence of an appropriate extraction rule, as well as its applicability to all posts of the blog.

Algorithm 1 shows the generic procedure we use to build extraction rules. The idea is quite simple: for each (*page, target*) input, compute, out of all possible extraction rules, the best one with respect to a certain `ScoreFunction`. The rule which is most frequently the *best rule* is then returned.

Algorithm 1: Best Extraction Rule

input : Set *pageZipTarget* of HTML page and target pairs
output : Best extraction rule

```
bestRules ⟵ new list
foreach (page, target) in pageZipTarget do
    score ⟵ new map
    foreach rule in AllRules(page) do
        extracted ⟵ Apply(rule, page)
        score of rule ⟵ ScoreFunction(extracted, target)
    bestRules ⟵ bestRules + rule with highest score
return rule with highest occurrence in bestRules
```

One might notice that each *best rule* computation is independent and operates on a different input pair. This implies that Algorithm 1 is *embarrassingly parallel*: iterations of the outer loop can trivially be executed on multiple threads.

Functions in Algorithm 1 are voluntarily abstract at this point and will be explained in detail in the remaining of this section. Section 3.3 defines `AllRules`, `Apply` and the `ScoreFunction` we use for article extraction. In section 3.4 we analyse the time complexity of Algorithm 1 and give it a linear time reformulation using dynamic programing. Finally, section 3.5 shows how the `ScoreFunction` can be adapted to extract authors, dates and comments.

3.3 Extraction rules and string similarity

In our implementation, rules are queries in the XML Path Language (XPath). Consequently, standard libraries can be used to parse HTML pages and apply extraction rules, providing the `Apply` function used in Algorithm 1. We experimented with 3 types of XPath queries: selection over the HTML `id` attribute, selection over the HTML `class` attribute and selection using the absolute path in the HTML tree. `id` attributes are expected to be unique, and `class` attributes have showed in our experiments to have better consistency than absolute paths over pages of a blog. For these reasons *Comment: there is a misscontinuation here. It's not for these reasons that we return a single rule per node. It's because we only need one rule per node. The rule is expressed in one of the three types mentioned above.*
TODO: The idea is that because there is such a strong hierarchy between the three rules (id > class > path), we only use "the best one". If it was not the case, we could also return 3 rules per node, which would triple the computation time. But I agree that it might not be very clear with "for these reasons"..., the `AllRules` function returns a single rule per

[1]Our experiments on a large dataset of blogs showed that failing tests were either due to a violation of one of these observations, or to an insufficient amount of text in posts.

node:

Function `AllRules`(*page*)

rules ⟵ new set
foreach *node* **in** *page* **do**
  if *node* **as** `id` attribute **then**
    rules ⟵ *rules* + {`"//*[@id='`*node.id*`']"`}
  else if *node* **as** `class` attribute **then**
    rules ⟵ *rules* + {`"//*[@class='`*node.class*`']"`}
  else *rules* ⟵ *rules* + {`AbsolutePathTo(`*node*`)`}
return *rules*

Unsurprisingly, the choice of `ScoreFunction` greatly influences the running time and precision of the extraction process. When targeting articles, extraction rules are scored *Comment: scored has a different meaning here. Graded or sorted or something similar is more correct*
TODO: ranked? evaluated? We should rename ScoreFunction accordingly and propagate the change everywhere. with a string similarity function comparing the extracted strings with the target strings. We chose the Sørensen–Dice coefficient similarity [5], which is, to the best of our knowledge, the only string similarity algorithm fulfilling the following criteria:

1. Has low sensitivity to word ordering,
2. Has low sensitivity to length variations,
3. Runs in linear time.

Properties 1 and 2 are essential when we deal with cases where the blog's web feed only contains an abstract or a subset of the entire post article. Table 1 gives examples to illustrate how these two properties hold for the Sørensen–Dice coefficient similarity but do not for *edit distance* based similarities such as the Levenshtein [12] similarity.

string1	*string2*	Dice	Leven.
`"Scheme Scala"`	`"Scala Scheme"`	90%	50%
`"Rachid"`	`"Richard"`	18%	61%
`"Rachid"`	`"Amy, Rachid and all their friends"`	29%	31%

Table 1: Examples of string similarities

The Sørensen–Dice coefficient similarity algorithm operates by first building sets of pairs of adjacent characters, also knows as *bigrams*, and then applying the *quotient of similarity* formula:

3.4 Time complexity and linear reformulation

The functions `AllRules`, `Apply` and `Similarity` (as `ScoreFunction`) being defined, the definition of Algorithm 1 for article extraction is complet. We now proceed with a time complexity analysis.

First, let's assume that we have at our disposal a linear time HTML parser that constructs an appropriate data structure, indexing HTML nodes on their `id` and `class` attributes,

Function `Similarity`(*string1, string2*)

bigrams1 ⟵ `Bigrams`(*string1*)
bigrams2 ⟵ `Bigrams`(*string2*)
return $2\,|bigrams1 \cap bigrams2| \;/\; (|bigrams1| + |bigrams2|)$

Function `Bigrams`(*string*)

return set of pairs of adjacent characters in *string*

effectively making `Apply` $\in \mathcal{O}(1)$. As stated before, the outer loop splits the input into independent computations and each call to `AllRules` returns (in linear time) at most as many rules as the number of nodes in its *page* argument. Therefore, the body of the inner loop will be executed $\mathcal{O}(n)$ times. Because each extraction rule can return any subtree of the queried page, each call to `Similarity` takes $\mathcal{O}(n)$, leading to an overall quadratic running time.

We now present Algorithm 2, a linear time reformulation of Algorithm 1 for article extraction using dynamic programming. While very intuitive, the original idea of first generating extraction rules and then picking these best rules prevents us from effectively reusing previously computed bigrams. For instance, when evaluating the extraction rule for the HTML root node, Algorithm 1 will obtain the complete string of the page and pass it to the `Similarity` function. At this point, the information on where the string could be split into substrings with already computed bigrams is not accessible, and the bigrams of the page have to be computed by linearly traversing the entire string. To overcome this limitation and implement *memoization* over the bigrams computations, Algorithm 2 uses a post-order traversal of the HTML tree and computes node bigrams from their children bigrams. This way, we avoid serializing HTML subtrees for each bigrams computation and have the guarantee that each character of the HTML page will be read at most once during the bigrams computation.

Algorithm 2: Linear Time Best Content Extraction Rule

input : Set *pageZipTarget* of Html and Text pairs
output : Best extraction rule

bestRules ⟵ new list
foreach (*page, target*) **in** *pageZipTarget* **do**
  score ⟵ new map
  bigrams ⟵ new map
  bigrams **of** *target* ⟵ `Bigrams`(*target*)
  foreach *node* **in** *page* **with** post-order traversal **do**
    bigrams **of** *node* ⟵
      `Bigrams`(*node.text*) $\bigcup$ *bigrams* **of all** *node.childs*
    score **of** *node* ⟵
      $\dfrac{2\,|(bigrams \text{ of } node) \cap (bigrams \text{ of } target)|}{|bigrams \text{ of } node| + |bigrams \text{ of } target|}$
  bestRules ⟵ *bestRules* + `Rule`(node with best *score*)
return rule with highest occurrence in *bestRules*

With bigrams computed in this dynamic programming manner, the overall time to compute all `Bigrams`(*node.text*) is

linear. To conclude the proof that Algorithm 2 runs in linear time we show that all other computations of the inner loop can be done in constant *amortized* time. As the number of edges in a tree is one less than the number of nodes, the *amortized* number of bigrams unions per inner loop iteration tends to one. Each *quotient of similarity* computation requires one bigrams intersection and three bigrams length computations. Over a finite alphabet (we used printable ASCII), bigrams sizes have bounded size and each of these operations takes constant time.

3.5 Variations for authors, dates, comments

Using string similarity as the only score measurement leads to poor performance on author and date extraction, and is not suitable for comment extraction. This subsection presents variations of the previously presented `ScoreFunction` which addresses issues of these other types of content.

The case of authors is problematic because authors' names often appear in multiple places of a page, which results in several rules with maximum `Similarity` score. The heuristic we use to get around this issue consists of adding a new component in the `ScoreFunction` for author extraction rules: the *tree distance* between the evaluated node and the post content node. This new component takes advantage of the positioning of a post's authors node which often is a direct child or shares its parent with the post content node.

Dates are affected by the same duplication issue, as well as the inconsistency of format between web feeds and web pages. Our solution for date extraction extends the `ScoreFunction` for authors by comparing the *extracted* string to multiple *targets*, each being a different string representation of the original date obtained from the web feed. For instance, if the feed indicates that a post was published on `"Thu, 01 Jan 1970 00:00:00"`, our algorithm will search for a rule that returns one of `"Thursday January 1, 1970"`, `"1970-01-01"`, `"43 years ago"` and so on. So far we do not support dates in multiple languages, but adding new target formats based on languages detection would be a simple extension of our date extraction algorithm.

Comments are usually available in separate web feeds, one per blog post. Similarly to blog feeds, comment feeds have a limited number of entries, and when the number of comments on a blog post exceeds this limit, comments have to be extracted from the post web page. To do so we use the following `ScoreFunction`:

- Rules returning less HTML nodes than the number of comments on the feed are filtered out with a zero score,
- Remaining rules are *TODO: scored* with the value of the *maximum weighted matching* in the *complete bipartite graph* $G = (U, V, E)$, where U is the set of HTML nodes returned by the rule, V is the set of target comment fields from the web feed (such as comment authors) and $E(u, v)$ has weight equal to `Similarity(`u`,` v`)`.

Our crawler executes this algorithm on each post with an overflow on its comment feed, thus supporting blogs with multiple commenting engines. The comment content is extracted first, allowing us to narrow down the initial filtering by fixing a target number of comments.

Regarding time complexity, computing the *tree distance* of each node of a graph to a single reference node can be done in linear time, and multiplying the number of targets by a constant factor does not affect the asymptotic computational complexity. *TODO: Scoring* rules for comment extraction requires a more expensive algorithm. However, this is compensated by the fact that the proportion of candidates left, after filtering out rules not returning enough results, is very low in practice. Analogous reformulations to the one done with Algorithm 2 can be straightforwardly applied on each `ScoreFunction` in order to minimize the time spent in `Similarity`.

4. TECHNIQUES

This section gives an overview of the different techniques used in our crawler. We first show how we integrated a headless web browser into the harvesting process to support blogs that use JavaScript to display page content. The overall software architecture is then discussed, introducing the Scrapy framework and the enrichments we implemented for our specific usage. Finally we talk about the design choices we made in view of a large scale deployment.

4.1 JavaScript rendering

JavaScript is a widely used language for client-side scripting. While some applications simply use it for aesthetics (menus, animations...), an increasing number of websites use JavaScript to download and display content. In such cases, traditional HTML based crawlers do not see web pages as they are presented to a human visitor and might therefore be obsolete for data extraction.

In our experiments whilst crawling the blog sphere, we encountered several blogs were data was missed because of the lack of JavaScript interpretation. The most frequent cases where blogs using the Disqus [6] and LiveFyre [13] comment hosting services. For web-masters, these tools are very handy to setup because the entire comments infrastructure is externalized and the setup essentially comes down to including a JavaScript snippet in each target page. Both of these services heavily rely on JavaScript to download and display the comments, even providing functionalities such as real-time updates for edits and newly written comments. Less commonly, some blogs are fully rendered using JavaScript. When loading such websites, the web browser will not receive page contents as HTML files, but will instead have to execute JavaScript code to download and display page contents. The Blogger platform provides as a default template the *Dynamic Views*, which uses this mechanism [9].

To support blogs with JavaScript generated content, we embed a full web browser into the crawler. After considering multiple options, we opted for PhantomJS [15], a headless web browser with great performances and scripting capabilities. The JavaScript rendering is the very first step of web page processing. Therefore, Extracting blog post articles, comments or medias works equally well on blogs with JavaScript generated content and traditional HTML-only blogs.

When the number of comments on a page exceeds a certain threshold, both Disqus and LiveFyre will only load the most recent ones and the stream of comments will end with a *Show More Comments* buttons. As part of the page loading process, we instruct PhantomJS to repeatedly click on these buttons until all comments are loaded. Paths to Disqus and LiveFyre *Show More* buttons were manually obtained. They constitute the only non-generic element of our extraction stack which will require human intervention to maintain and extend to other commenting platforms.

4.2 Architecture

Our crawler is built on top of Scrapy [18], an open source Python framework for web crawling. Scrapy provides an elegant and modular architecture illustrated in Figure 1. Several components can be plugged into Scrapy core infrastructure:

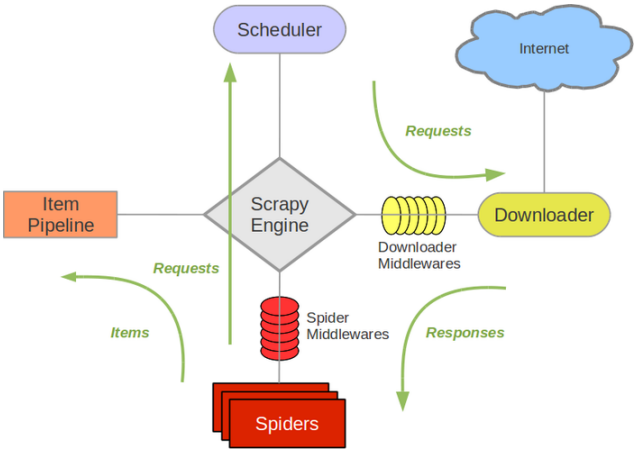


Figure 1: Overview of the crawler architecture. (Credit: Pablo Hoffman, Daniel Graña, [18])

Spiders, *Item Pipelines*, *Downloader Middlewares* and *Spider Middlewares*; each allowing us to implement a different type of functionalities.

Our use case has two types of spiders: *NewCrawl* and *UpdateCrawl*, which implement the logic to respectively crawl a new blog and get updates from a previously crawled blog. After being downloaded and identified as blog posts (details in 4.3), pages are packed into *Items* and sent through the following pipeline of operation:

1. Render JavaScript
2. Extract content
3. Extract comments
4. Download media
5. Propagate resulting records to the back end

This pipeline design provides great modularity. Indeed, disabling JavaScript rendering or plugging in an alternative back-end can be done by editing one single line of code.

4.3 Enriching Scrapy

In order to identify web pages as blog posts, our implementation enriches Scrapy with two components to narrow the extraction process to the subsets of blog pages holding blog posts: *blog post identification* and *download priority heuristic*.

Given an entry point to a website, the default Scrapy behavior looks over all pages of the same domain in a *last-in-first-out* manner. The *blog post identification* function is able to identify from an URL whether or not the corresponding page is a blog post. Internally, for each blog, this function uses a regular expression constructed from web feed entry URLs. This simple approach requires that blogs use the same pattern for all posts (or false negative will occur) which has to be distinct for non-post pages (or false positive will occur). In practice, this assumption holds for all blog platforms we encountered and seems to be a common practice amongst web developers.

In order to efficiently deal with blogs with a large number of non-blog-post pages, this *blog post identification* mechanism

is not sufficient. Indeed, after all pages identified as blog posts are processed, the crawler needs to download non-blog-post pages to search for additional blog posts. To replace the naive *random walk*, *depth first search* or *breadth first search* web site traversals, we use a priority queue where priorities of new URLs are determined by a machine learning system. This mechanism has shown to be mandatory for blogs hosting on a single domain a large number of non-blog web pages, such as a forum or a wiki.

The idea is to give a high priority to URLs which are believed to point to pages with links to blog post. These predictions are done using an active *Distance-Weighted k-Nearest-Neighbor* classifier [7]. Let $L(u)$ be the number of links to blog posts contained in a page with URL u. Whenever a page is downloaded, it's URL u and $L(u)$ are given to the machine learning system as training data. When the crawler encounters a new URL v, it will ask the machine learning system for a estimation of $L(v)$, and use this value as the download priority of v. $L(v)$ is estimated by doing a weighted average on the values of the k URLs most similar to v.

4.4 Scalability

When aiming to work with a large number of inputs, it is crucial to build every layer of a system with scalability in mind [1]. The BlogForever crawler, and in particular the two core procedures *NewCrawl* and *UpdateCrawl*, are designed to be usable as part of an event-driven, scalable and fault resilient distributed system.

To go in this direction, we made the key design choice to have both *NewCrawl* and *UpdateCrawl* as stateless components. From a high level viewpoint, these two components are *purely functional*:

$$NewCrawl : \mathbb{U}\text{RL} \to \mathcal{P}(\mathbb{R}\text{ECORD})$$
$$UpdateCrawl : \mathbb{U}\text{RL} \times \mathbb{D}\text{ATE} \to \mathcal{P}(\mathbb{R}\text{ECORD})$$

Where $\mathbb{U}$RL, $\mathbb{D}$ATE and $\mathbb{R}$ECORD are respectively the set of all URLs, dates and records, and $\mathcal{P}$ designates the powerset operator. By delegating all shared mutable state to the back end system (Invenio's database), web crawler instances can be added, removed, and used interchangeably.

5. EVALUATION

Our evaluation will be articulated in two parts. Firstly, we will compare the article extraction procedure presented in section 3 with three open source projects capable of extracting content and title of web articles. The comparison will show that our blog targeted solution has better performances both in term of success rate and running time. Secondly, a discussion will be held regarding the different solutions available to archive data beyond what is available in HTML source code. Extraction of authors, dates and comments is not part of evaluation because of the lack of publicly available competing projects and reference data sets.

Experiments used *Debian GNU/Linux 7.2*, *Python 2.7* and an *Intel Core i7-3770 3.4 GHz* processor. Timing measurements were made on a single dedicated core with garbage collector disabled. The Git repository of this paper [3] contains the necessary scripts and instructions to reproduce all evaluation experiments presented in this section.

5.1 Extraction success rates

To evaluate post article and title extraction we compared our approach to three open source projects: Readability [16], Boilerpipe [11] and Goose [8], implemented in JavaScript, Java and Scala respectively. These projects are more generic than our blog specific approach in the sense that they are able to identify and extract data directly from HTML source code, and do not make use of web feeds or structural similarities between pages of a same blog (observations (1) and (2)). Table 2 shows extraction success rates for article and title on a test sample of 2300 blog posts obtained from the Spinn3r dataset [4].

Target	Our approach	Readability	Boilerpipe	Goose
Article	93.0%	88.1%	79.3%	79.2%
Title	95.0%	74.0%	N/A	84.9%

Table 2: Extraction success rates

On our test-set Algorithm 1 outperformed the concurrence by 4.9% on article extraction and 10.1% on title extraction. It is important to stress that Readability, Boilerpipe and Goose rely on generic techniques such word density, paragraph clustering and heuristics on HTML tagging conventions, which are designed to work for any type of web pages. Contrariwise, our algorithm is only suitable for pages with associated web feeds, as these provide the reference data which we search for in HTML trees of pages. Therefore, results showed in Table 2 should not be interpreted as general quality evaluation of the different projects, but simply show that our approach is more suitable when working with blogs.

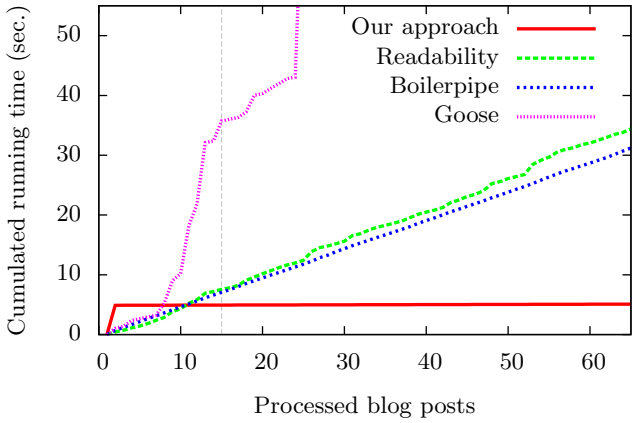
5.2 Article extraction running times

In addition to the quality of the extracted data we also evaluated the running time of the extraction procedures. The main point of interest will be the scalability of extraction procedures as the number of posts in the processed blog increases. This corresponds to the evaluation of a *NewCrawl* task, which is in charge of harvesting all previously published content on a blog.

Experimental data was obtained by taking the arithmetic mean over 10 measurements, which we believe to be significant given that each result has a standard deviations on the order of 2 milliseconds. *Comment: Do I have to justify the arithmetic mean? If yes, this could do:* `http://dl.acm.org/citation.cfm?id=5673`

Figure 2 shows the cumulated time spent on each article extraction procedure (this excludes common tasks such as downloading pages and storing results) as a function of the number of blog posts processed of `http://quantumdiaries.org/`.

Figure 2: Running time of articles extraction



Past the initial computations, the cost to extract an article with our approach are almost null. This is a consequence of having a blog aware, rule based algorithm. If you recall the procedure presented in section 3, the central idea was to first build extraction rules using the information provided by the web feed, and then use these rules on all posts of the blog. On this example, time cost of parsing a new blog post and using extraction rules is in the order of 3 milliseconds, which is not visible on the scale of Figure 2. The three other evaluated solutions do not function this way: each blog post is processed as a new and independent input, leading to a close to linear running times.

In Figure 2, the vertical dashed line at 15 processed blog posts represents a suitable point of comparisons for processing time per blog posts. Indeed, as the test blog's web feed contains 15 blog posts, the time spent on processing the first blog post also includes the processing time of the web feed and it's 15 entries. *Comment: Is this clear?*
That being said, comparing raw performances of different algorithm implemented with different programming languages is not very informative given the high variations of running times observes across languages for identical algorithms [10].

5.3 JavaScript rendering

6. CONCLUSION AND FUTURE WORK

In this paper, we presented the internals of the BlogForever web crawler. Its central article extraction procedure based on extraction rules generation was introduced along with theoretical and empirical evidences validating the approach. Simple adaptation of this procedure allowed to extract different types of contents, including authors, dates and comments. In order to support rapidly evolving web technologies such as JavaScript generated content, the crawler uses a web browser to render pages before processing. We also discussed the overall software architecture, highlighting the design choices made to achieve both modularity and scalability.

Future work could investigate *hybrid* extraction algorithms to try to achieve near 100% success rates. Even if overall our approach obtains better performs, there are few cases where other techniques managed to extract data where we did not. This suggests that combining our approach with others such such as word density, tree edit distance matching or even spacial reasoning could lead to better performances.

Another possible research direction would be the deployment of the BlogForever crawler on a large scale distributed system. This is particularly relevant in the domain of web crawling given that intensive network operations can be a serious bottleneck which benefits from the use of multiple Internet access points. We intend to explore this opportunities in our future work.

To open up further possibilities, the crawler presented in this paper is available under the MIT license from the project's website [2].

7. REFERENCES

[1] The reactive manifesto. http://reactivemanifesto.org/, 2013.

[2] Repository of the blogforever crawler. https://github.com/BlogForever/crawler, 2013.

[3] Repository of this paper. http://goo.gl/fl6Fwh, 2013.

[4] K. Burton, N. Kasch, and I. Soboroff. The ICWSM 2011 spinn3r dataset. In *Fifth Annual Conference on Weblogs and Social Media*, 2011.

[5] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297, July 1945.

[6] Disqus. http://disqus.com/websites/, 2013.

[7] S. A. Dudani. The Distance-Weighted k-Nearest-Neighbor rule. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-6(4):325–327, 1976.

[8] Goose. https://github.com/GravityLabs/goose, 2012.

[9] A. Harasymiv. Blogger dynamic views. http://goo.gl/Ma3G2I, 2011.

[10] R. Hundt. Loop recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days*, 2011.

[11] 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*, WSDM '10, page 441–450, New York, NY, USA, 2010. ACM.

[12] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*, 10(8):707–710, Feb. 1966.

[13] Livefyre. http://web.livefyre.com/, 2013.

[14] M. Oita and P. Senellart. Archiving data objects using web feeds. Sept. 2010.

[15] PhantomJS. http://phantomjs.org/, 2013.

[16] python readability. https://github.com/gfxmonk/python-readability, 2011.

[17] A. Rogers and G. Brewer. Microdata usage statistics. http://trends.builtwith.com/docinfo/Microdata, 2013.

[18] Scrapy. http://scrapy.org/, 2013.

[19] W3C. Use <h1> for top level heading. http://www-mit.w3.org/QA/Tips/Use_h1_for_Title, 2002.

[20] W3C. W3C standards. http://www.w3.org/standards/, 2013.

[21] WHATWG. Microdata - HTML5 draft standard. http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html, 2013.

[22] B. Wilson. Metadata analysis and mining application: Common attributes. http://dev.opera.com/articles/view/mama-common-attributes/, 2008.

[23] B. Wilson. Metadata analysis and mining application: W3C validator research. http://dev.opera.com/articles/view/mama-w3c-validator-research-2/, 2008.