

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. ALGORITHMS

This section explains in detail the algorithm we developed to extract blog posts' article and its variations for authors, dates and comments. Our approach uses blog specific characteristics to build extraction rules applicable to all posts of a blog. Our focus is on minimizing the algorithmic complexity while keeping our approach simple and generic.

1.1 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 `` and `<div></div>` container elements with custom `id` or `class` attributes to organise structure 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 use the following observations which play a central role in the extraction process¹:

- (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, often less than the total number of posts in blogs [14]. Consequently, in order to effectively archive old content from blogs, it is necessary to download and process pages beyond the ones referenced in web feeds.

1.2 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 page 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 inputs 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 a *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 details in the remaining of this section. 1.3 defines **AllRules**, **Apply** and the **ScoreFunction** we use for article extraction. In 1.4 we analyse time complexity of Algorithm 1 and give a linear time reformulation using dynamic programming. Finally, 1.5 shows how the **ScoreFunction** can be adapted to extract authors, dates and comments.

1.3 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, thus 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 with the absolute path in the HTML tree. `id` attributes are expected to be unique, and `class` attributes have showed to have better consistency than absolute paths over pages of a blog. For these reasons, the **AllRules** function returns a single rule per 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

¹Our 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.

with a string similarity function comparing the extracted and 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 to 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.

<i>string1</i>	<i>string2</i>	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 known as *bigrams*, and then applying the *quotient of similarity* formula:

Function Similarity (<i>string1</i> , <i>string2</i>)
<i>bigrams1</i> \leftarrow Bigrams (<i>string1</i>)
<i>bigrams2</i> \leftarrow Bigrams (<i>string2</i>)
return $2 bigrams1 \cap bigrams2 / (bigrams1 + bigrams2)$
Function Bigrams (<i>string</i>)
return set of pairs of adjacent characters in <i>string</i>

1.4 Time complexity and linear reformulation

The functions **AllRules**, **Apply** and **Similarity** (as **ScoreFunction**) being defined, the definition of Algorithm 1 for article extraction is completed. 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, 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  $\leftarrow$  new list
foreach (page, target) in pageZipTarget do
    score  $\leftarrow$  new map
    bigrams  $\leftarrow$  new map
    bigrams of target  $\leftarrow$  Bigrams(target)
    foreach node in page with post-order traversal do
        bigrams of node  $\leftarrow$ 
            Bigrams(node.text)  $\cup$  bigrams of all node.childs
        score of node  $\leftarrow$ 
             $\frac{2 |(\textit{bigrams of node}) \cap (\textit{bigrams of target})|}{|\textit{bigrams of node}| + |\textit{bigrams of target}|}$ 
        bestRules  $\leftarrow$  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.

1.5 Variations for authors, dates, comments

Using string similarity as only score measurement leads to poor performances on authors and dates extraction, and is not suitable for comment extraction. This subsection presents variations of the previously presented **ScoreFunction** which address issues of these other contents.

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 in adding a new component in the **ScoreFunction** for authors extraction rules: the *tree distance* between the evaluated node and the post content node. This new component takes advantage of the positioning of post's authors 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 dates 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 base and 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 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 $E(u, v)$ as weight equal to **Similarity**(u, v).

Our crawler executes this algorithm on each post with overflow on its comment feed, thus supporting blogs with multiple commenting engines. Comment contents are extracted first, which allows to narrow down the initial filtering by fixing a target number of comments.

Regarding time complexity, computing the *tree distance* of each nodes 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 affects the asymptotic computational complexity. Scoring rules for comments extraction requires a more expensive algorithm. However, this is compensated by the fact that the proportion of candidates left, after filter out rules not returning enough results, is very low in practice. Analog reformulations to the one done with Algorithm 2 can be applied on each **ScoreFunction** in order to minimize the time spent in **Similarity**.

2. TECHNIQUES

This section gives an overview of 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 will then be discussed, introducing the Scrapy framework and the enrichments we implemented for our specific usage. Finally we will talk about the design choices we made in view of large scale deployment.

2.1 JavaScript rendering

JavaScript is a widely used language for client-side web 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.

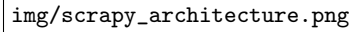
In our experiments whilst crawling the blog sphere, we encountered several blogs where data was missed because of the lack of JavaScript interpretation. The most frequent cases were blogs using the Disqus [6] and LiveFyre [13] comment hosting services. For web-masters, these tools are very handy to setup 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.

2.2 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 in Scrapy core infrastruc-



img/scrapy_architecture.png

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

ture: *Spiders*, *Item Pipelines*, *Downloader Middlewares* and *Spider Middlewares*; each allowing 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 2.3), pages are packed into *Items* and sent through the following pipeline of operation:

1. Render JavaScript
2. Extract content
3. Extract comments
4. Download medias
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.

2.3 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 held 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 post 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.

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, its 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 an 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 .

2.4 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*:

$$\textit{NewCrawl} : \text{URL} \rightarrow \mathcal{P}(\text{RECORD})$$

$$\textit{UpdateCrawl} : \text{URL} \times \text{DATE} \rightarrow \mathcal{P}(\text{RECORD})$$

Where URL, DATE and RECORD 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.

3. EVALUATION

Our evaluation will be articulated in two parts. Firstly, we will compare the article extraction procedure presented in section 1 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 were made on *Debian GNU/Linux 7.2* with *Python 2.7* on an *Intel Core i7-3770 3.4 GHz* processor. Timing measurements used 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.

3.1 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

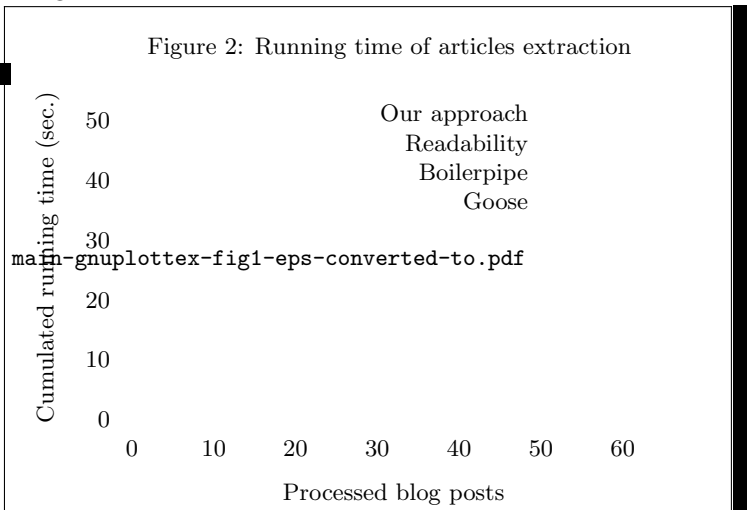
On our test-set Algorithm 1 outperformed the concurrence by 4.9% on article extraction and 10.1% on title extraction. It is important to stress that Readability, Boilerpipe and Goose rely on generic techniques such word density, paragraph clustering and heuristics on HTML tagging conventions, which are 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.

3.2 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/>.



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 1, 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].

3.3 JavaScript rendering

4. 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 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].

5. 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/f6Fwh>, 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] Scrappy. <http://scrappy.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.