Adaptive Post Recognition

Combine Feeds and Linked HTML pages to Generate Blog Templates

Philipp Berger*, Patrick Hennig*, Dominic Petrick†, Marcel Pursche† and Christoph Meinel‡
Hasso-Plattner-Institute
University of Potsdam, Germany
*{philipp.berger, patrick.hennig}@hpi.uni-potsdam.de
†{dominic.petrick, marcel.pursche}@student.hpi.uni-potsdam.de
‡office-meinel@hpi.uni-potsdam.de

***Abstract*—Blogs, news portal and discussion forums are of high interest for today's social interaction research. But the automatic information extraction from the raw html page of those media channels is still a well-known problem.**
We introduce a novel approach to infer website templates based on the syndication format of blogs and news portals, called feeds. In comparison to related approaches that infer templates by clustering generic pages, we do not rely on a manual annotated training set. Instead, we use the feeds and their linked articles as training set to identify characteristic XPaths. Those paths identify the exact article content and article properties like title, author and publishing date.
Further, we can use those paths to detect article pages that are no longer linked from feeds. We show the precision gain by comparing the article content extraction with an alternative approach e.g. boilerplate.

I. Introduction

The number of blogs in the World Wide Web drastically increased over the past years [1], [2], making them a valuable source of information. As the amount of information exponentially grows [3], current research focuses on the automatic extraction of new knowledge. Thus, the calculation of trends, communities, and influencers based on text data gets feasible.

Nevertheless, an important precondition for all kinds of data mining is the successful data collection and integration. Therefore, various crawling techniques have evolved. Beside the regular crawling of web pages, specialized crawlers concentrate on the crawling of blog feeds to collect valuable information [4], [5]. These feeds conform to standardized syndication formats, like ATOM and RSS, used by blogs and news portals to publish articles via a machine-readable channel [6]. Users can subscribe to these feeds and stay up-to-date with their favorite websites.

The first problem is that most feeds only contain the 10 newest posts of a blog hiding the total set of posts from a common feed crawler. Therefore, we infer XPath expressions from the seed list of posts obtained by the blog feed. These paths enable us to decide whether a randomly crawled page of a blog is a post. Further, we can automatically extract all semantic attributes that normally only occur in the blog feed (see Figure 1). This makes it possible to semantically crawl a whole blog without having a feed that contains all posts.

The second problem is that most blogs and news portals tend to only publish excerpts of their content in the feeds to direct users to their own pages. This enables the websites to show more advertisements and other additional information to users. Although this can be useful for a user, it hinders feed crawlers to collect the whole article content and deteriorates the data mining results as well. Current research proposes algorithms to remove boilerplate code from HTML websites and thereby only extract the actual article content out of the crawled pages. These algorithms try to infer the underlying templates of HTML pages to identify the main content areas.

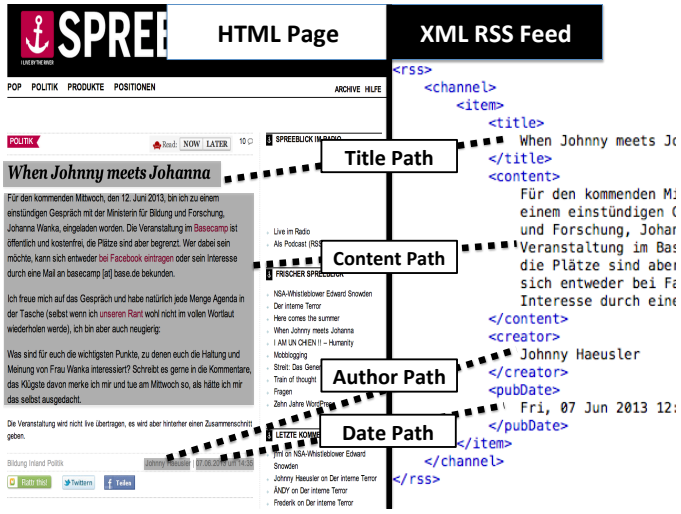

Figure 1: By matching XML and HTML elements, we find characteristics paths that can be used to extract information from raw pages.

We argue that the usage of the extracted XPath expressions to identify the information areas of blogs and news portals improves the extraction precision.

The main contributions of this paper are as follows. First, we propose a method to generate a blog template via XPath-like expressions using the XML feed and utilize this template to decide whether a given webpage of a host is a blog post. Although a post is no longer reference by the blog feed, we show how to use this template to extract property values out of the identified blog post HTML page.

This paper is structured as follows: Section II names

related approaches and groundwork, which is the basis for our approach. The detailed description of our approach and the underlying extraction concepts are presented in Section III, which also dives into the article detection procedure. In Section IV, we evaluate our approach on real world data and discuss challenges. Next, we point out ideas for further improvement of our approach (see Section V). Finally, we conclude our work and summarize our results.

II. Related Work

The crawling of weblogs for data mining is a well-known problem in the community. For example, Berger et al. [4] present strategies and concepts for implementing a distributed blog crawler.

Nevertheless, our goal is to identify the article properties, like title, content and author, of an article page. The following related work concentrates on the problem of removing boilerplate code from a HTML document, e.g. identifying the actual interesting content. One can differentiate two approaches for boilerplate removal: page-level methods and site-level methods.

Page-level methods take only one HTML document as input. Finn et al. [7] use heuristic methods based on the HTML tag density to identify the main content. Marek et al. [8] presented a supervised machine-learning algorithm based on language models. Likewise, Kohlschuetter et al. [9] approach is based on a decision tree by using only two features: word count and link density.

As contrast, site-level methods investigate the structure of all pages of one website to infer the underlying generation templates. Bar-Yossef et al. [10] identify blocks in a web page and hash these blocks. By analyzing the occurrences in the website they remove each block that occurs more than once. By counting similar DOM tree paths Yi et al. [11] find frequently used HTML markup, which is characteristic for boilerplate code.

Beside those general boilerplate removal approaches, Kushmerick et al. [12] define a semi-automatic algorithm, called wrapper induction. They define a wrapper in information extraction context as "a procedure for extracting tuples from a particular information source" [12]. Instead of manually building wrappers for each web page, the algorithm uses a set of sample pages and manually extracted data to create wrappers.

Crescenzi et al. proposes a method to automatically create a suitable sample set of web pages for wrapper induction [13]. The authors cluster page by their structure to create the sample sets. The clustering exploits that today's pages often share a common server-side template while differing in the presented content. Chakrabarti and Mehta describe a different approach for clustering arbitrary web pages without using manual input [14]. They identify important parts of a page using search logs with keywords and clicked pages for these search terms; assuming that the keywords are part of the content the user is looking for. Therefore, the DOM-tree paths, e.g. light weight XPath expression, that point to important portions of a page will be considered as identification candidates for clustering pages. This path is called the key-path. Since there can be multiple key-paths for a host, they form the clusters of the host for which wrappers can then be induced automatically using the key-paths to first cluster pages and then extract the important portions of a page.

All these approaches focus on information extraction from arbitrary websites. Because our approach only focuses on posts on blogs, we can make assumptions to simplify the problem, while still keeping some of the basic concepts mentioned in these works. Especially the idea of key-paths will be utilized in our approach to find characteristic XPaths in the DOM tree (see Section III-D).

III. Algorithm and Extraction Concepts

In this section we will describe what concepts we developed to realize an automatic template detection and how this helps us to extract the contents of interest from blog posts. First, we give a general overview which concepts are needed and how they work together to form our algorithm. The following subsections will then go more into details about the concepts and steps, highlighting major problems we encountered along the way. Hereby, we focus on the extraction of the article content that is exemplarily for the handling of other article properties.

A. Overview

In order to automatically detect blog post templates, we need to find a way to only cluster blog posts together. Clustering web pages requires a set of meaningful sample input pages so that we can form clusters and deduce how pages, which belong to a cluster, are structured. Since we only want to cluster blog posts and want other pages to be discarded in the clustering process, we can take the feeds of a blog as sample pages, assuming that feeds point only to blog posts.
Feeds are a must-have feature for today's blogs to enable their users to receive the latest posts through news feed aggregators, so we can simply assume that every blog has at least one feed to subscribe.
A widely established practice is to not distribute the whole blog post content through the different feeds a blog might have, but instead give the user a small preview of the content – mostly the beginning of a post – along with its title to awake the interest in the user. Apart from the preview, feeds of course contain a link to the full post, and they can also contain additional properties like author information, categories or tags and a publication date.

Our main idea is to utilize the key-path idea of Chakrabarti et al. [14] and extract *key-paths* from blog posts a feed points to, and match crawled HTML pages against these key-paths. A *key-path* is a characteristic path in the DOM tree, often starting at the HTML document root and ending with a node within the page. It is essentially a very lightweight variant of an XPath expression.

It contains a number of nodes optionally with attributes attached, as explained in Section III-B.
Items of a feed point to posts that are all similar in their structure, most likely due to a shared server-side template. Therefore, we are able to extract very similar key-paths from those structurally similar posts. We extract a key-path from every post a feed points to and try to unify (merge) the set of

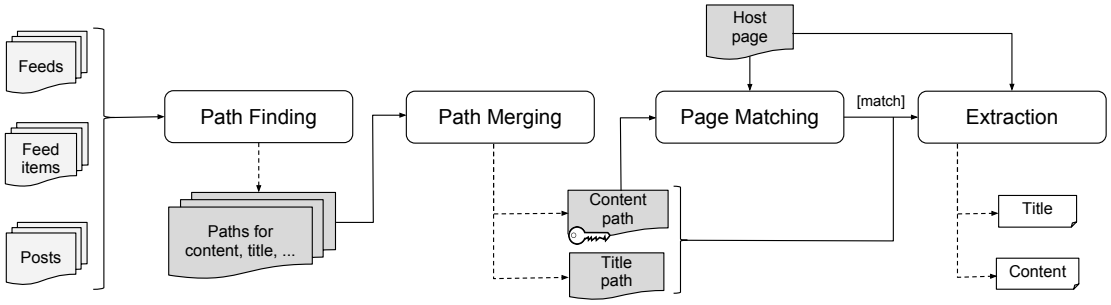

Figure 2: Overview of the conceptual process

key-paths for a single feed. The result is a universal key-path identifying a set of similar blog post, which can then be used to decide whether a randomly download HTML page is a post and to extract the article properties.

The idea behind our extraction approach is that we do not exclusively create a key-path from the items of a feed, but we first create paths to all information we want to extract later on and promote one path to be the key-path (for page identification). The concept of path detection applies for each feed property set for a blog feed. Each set of paths – e.g. title, content and author paths – will be merged to create a universal path pointing to the corresponding content area for one cluster. They can then be used to extract content from posts, which match against the key-path of this feed. Figure 2 gives an overview of the described conceptual process, beginning at the Path Finding.

B. Path Model

Since our algorithm relies on the DOM tree to cluster pages and extract data, the definition of a suitable path model is an important prerequisite. This model is necessary to represent certain subtrees that contain the information we want to extract and also to serialize these subtrees for later use. The path model should not be too complex because a lot of operations are necessary to find paths, but complex enough to represent all necessary information.

In our implementation we use a model that is essentially a lightweight XPath[1]. Our subset of XPath only focuses on the type of DOM nodes, their parent/children relations and their attribute value pairs. This very simple path model also guarantees that a path contains enough information to find specific nodes in a web page, but is general enough to ignore small differences in the subtrees, e.g. different ids, different order of nodes or different style classes.

The two examples shown in Listing 1 illustrate how our simplified version of XPTH works.

A path consists of number of nodes characterized by their element name, attributes and attribute values. The order of the nodes models the parent/child hierarchy of the nodes in the DOM tree. The first path in the example is an absolute path which means that all nodes from the document root node to the last node have to match the nodes in the path for the whole path to match. The second path on the other hand is a relative path. A node can occur anywhere in the DOM tree to match this path, only the hierarchy of its parents is important.
Our path model also supports wildcards (the asterisk-character '*'). This brings the possibility to create paths with an arbitrary number of nodes in between two nodes. It is also possible to create simple regular expressions for the value of an attribute with the * character. This is very helpful if a web page uses ids for the nodes that consist of a constant and a unique part. With the help of wildcards we are able to create generic and reliable paths.

Our subset of XPath is by no means fixed; it can be replaced to further improve the precision. Any model which supports the three necessary operations will suffice: (i) find a path from a given DOM node to the root node of the document, (ii) merge two paths to create a path that contains all common characteristics of both paths and (iii) find a node in the DOM tree that matches a given path.

C. Merging Paths

Given multiple XPaths, we like to unify those into one characteristic XPath. Our algorithm is divided into two parts:

1) Finding the common prefix of both XPaths
2) Finding the common attributes and attribute-values for each node

To find the common prefix of two XPaths, the algorithm simply iterates over the nodes of both paths and adds the node to the merged path if the node type matches in both paths. The iteration is aborted if the node type is not equal. To prevent the creation of too generic paths we set a minimum prefix length. All paths that do not satisfy this threshold are not merged.

In the second step, the attributes of each node are compared. If an attribute exists in both paths, it is added to the node in the merged path. For the attribute values, finding the common prefix of both values and adding the wildcard character create a simple regular expression. If the values are equal, no regular expression is created and instead the specific value is used. In this case both values have no common prefix, only the wildcard character is used as value. Thus, only the presence of this attribute is relevant during matching.
The example shown in Listing 2 illustrates the functionality of the merge operation:

[1]XPath-Specification: http://www.w3.org/TR/xpath20/

Listing 1: Path examples for our XPath variant.

```
(1)  |html|body|div[@id=post-*]|div[@class=post-header]|h1

(2)  div[@id=post-*]|*|div[@class=post-content,@id=post-*]
```

Listing 2: Examples for merged paths.

```
(P1)  |html|body|div[@id=post-2]|div[@class=post-header]|h1
(P2)  |html|body|div[@id=post-1]|div[@class=post-header]|b
(P3)  |html|body|span[@id=post-3]|h1

merge(P1, P2) = |html|body|div[@id=post-*]|div[@class=post-header]
merge(P1, P3) = null  (XPaths incompatible, no merging is possible)
```

D. Finding Paths

Given our simplified XPath and the mechanism for merging, we can create paths. As already discusses, our plan is to derive the characteristic HTML path by locating the post attributes of a blog feed in the HTML page. The information extraction and of the post detection relies on those extracted paths. First, we investigate the HTML structure of blogs among different blog platforms.

a) Observations: Before developing our algorithm we investigated the HTML code of blogs from different origins like BlogSpot, WordPress, news sites and other much smaller blogs. We observed that the general structure of blog posts is mostly a simple schema with a container wrapping the title and content container. The container nesting varies to a certain extent, meaning that the title is sometimes for example a sibling node and sometimes a parent element of the nodes which encapsulates the content of a post, which is something we have to consider when choosing a suitable paths. Sometimes we only have a content container but no title container or vice versa.
Looking more into container elements we see that titles are often inside an *h** element as expected, but there are some cases which have a main title and a subtitle wrapped into a *heading* element. The internal structure of content containers can differ greatly between hosts. A general observation is that news sites tend to use more convenient *p* and *span* – in case of HTML5 *section* – elements to structure the post content itself, while some blogs on BlogSpot use a mix of everything, including *div* and *br* for formatting. Another observation, HTML attributes are frequently used that can help to identify the containers on a page, like *id* or *class*. They contain values such as *post-id-** where the asterisk stands for a unique post id.

b) Matching Properties: Knowing the principle structure of HTML documents, we can now derive concepts for matching the XML properties contained in a feed with the actual nodes contained in the HTML page.

First, we clean the XML property values by removing any HTML construct, which might be left in the feed and could deteriorate the matching process.

To match the content of an XML property with the content of an HTML node, we need to keep in mind that the XML value often spans multiple HTML elements like *p* elements[2]. We shrink the XML-content until we find an matching HTML-content, as shown in Figure 3. To avoid useless matches, we define a minimum length for the content.

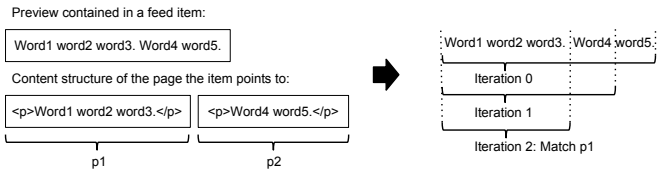

Figure 3: Removing words to match a preview that spans multiple elements

We execute this step for each XML property to find the starting HTML node. The final result is a XPath-like expression as already shown in Section III-B In case of no matching HTML node, we skip the current feed item and continue with its successor.

We prefer to step-wise shorten the XML attribute value to editing distances like Leventhein [15], because feeds are automatically generated. Thus, the typical editing distance and other text metrics do not apply, because these distances are adapted for common typing mistakes or signal processing errors. In contrast, the XML values of feeds originate from the exactly same data as the HTML content, but feeds often show only a few paragraphs or characters of a data value to motivate the user for visiting the webpage (mostly for placing ads).

Given a successful match of the XML value, we have to incorporate the following three stages: resolving multiple path matches, generalizing an XPath and applying blog-specific adaptions.

1) Resolving multiple matches: In general, the shorter an XML value the more probable is multiple matches inside the HTML. For example, the title of a blog post can occur in the *header* and in an *h1* element. This problem also occurs when a XML value gets strongly truncated. Figure 4 illustrates an example of a page structure that leads to multiple hits for a title

[2]These elements are either omitted or stripped in the feed preview, depending on how the client application preprocesses the feed and its items.

path. Shaded rectangles represent elements that may contain the title (dark shade) or content (light shade) string from a specific feed item.

Especially news sites tend to include columns (side bars) on their pages that show popular articles with a title and (sometimes) a picture, but a short preview like the one we have in feed items is possible as well, which can lead to multiple matches for the preview.

In case of truncated attributes, we expand the identified key paths until we are able to find a match for the non-truncated content. We step-wise add sibling nodes or truncate the paths. If we are able to match the non-truncated content, we found a valid path.

Otherwise, we resolve these ambiguities with a dependency-based resolution of paths. Suppose we found a single match for the content path and multiple matches for the title path. The correct title path has to be close to the content path, meaning that we maximize the shared prefix length of the content path and the title paths. This applies for all other values as well.

2) Generalizing paths: At the end we ideally have a single path for every piece of information the feeds items provided – either because we had a single path match the first time or we successfully resolved multiple paths. Thereby, it is possible that the final path it too specific for the searched property caused by the property value truncation.

We observed that XML properties like title and content can be structured into multiple HTML elements like a set of paragraphs. If we find and/or resolve a path, we have to ensure that the path does not end in an element too specific for extraction. Taking the content path as an example, the path may end in a *p* element. But this element only contains a fraction of the post content, so we need to find a way to include sibling nodes to get the content as a whole – in this example sibling *p* elements. Therefore, we traverse the parents of the matched HTML element to search for a non-truncated XML value match.

Essentially, we apply the same procedure as for resolving multiple paths. For example, elements like *p* and *section* get dropped from the end of the final content path, or in case of a title we cut *h** elements from the end as long as there is a parent which groups header elements (like *heading*).

3) Blog-specific adaptions: To further adapt this generalized approach to the specific characteristics of blogs we also integrated a set of heuristics. Figure 4 shows that the *title* element in the HTML head may contain the name of the post title. According to our observations, we can assume that the actual observable content of blogs is only contained in the body element.

Attributes like class and id often have unique values that help identifying the correct path from among a set of candidates. Especially, the content of a post is frequently annotated with a *class* or *id* like *article*, *post-body* or *post-id-**. To take advantage of this we cluster the attribute ids by common prefixes identifying to automatically identify indicators like *id-**.

By investigating the feeds of blogs, we found malicious feeds that contain more than one article property in a single XML property. For example, the title attribute of a feed contains both, the title and the actual content. Similar to our blog heuristics for path matching, this is a blog-specific issue that is caused by wrong semantic annotation. If we found a path – either title or content – this path actually points to the title, because if title and content are contained in a single feed XML property, the title will be right before the content at the beginning of the text. Since we iteratively truncate the XML property content until a HTML node match occurs, we end up with the title string. Given a title-only or content-only feed, we than remove the title string from the concatenated content and search again for the content to find a HTML node match. Thus, we have the case, that the path generalization will exceed a specific threshold...

E. Matching a Page

The result of the path finding is a set of XPath expressions for each host (with ideally one key-path per feed). To determine whether a web page is a blog post the algorithm tests if one of the key-paths for a host matches the web page i.e. if the node designated by the key-path can be found in the web page. Thus, we traverse the DOM tree beginning from the document root node. If the root node matches the node in the path, i.e. node type and node attributes are equal, all children of the node will be traversed. This algorithm continues recursively until a node matches the whole XPath or no node is found.

Listing 3: Exemplary HTML page of a blog.

```
<html>
    <head>
        <title>A blog post</title>
    </head>
    <body>
        <div class="advertisement">
            <div class="flash-container">
                [...]
            </div>
        </div>
        <div id="post-1337" class="post">
            <div class="header">
                <h1>A blog post</h1>
                <h2>Subtitle</h2>
            </div>
            <div class="body"/>
                <p>Some Sample content</p>
                <p>Another Paragraph</p>
            </div>
        </div>
    </body>
</html>
```

If the web page is detected as a post the article properties of the page can be extracted using the property-specific XPath expression created for the feed of the matched . The information extraction works after the same principle but with one addition: not the first matching node is used to extract the content but the first matching not that is not empty. This simple heuristic helps in cases when the path was too generic to safely determine the right node e.g. the nodes do not contain unique attributes or the path is very short.

The following example shows the page matching and content extraction, but also what problems can occur with to general paths.

The page HTML is shown in Listing 5.

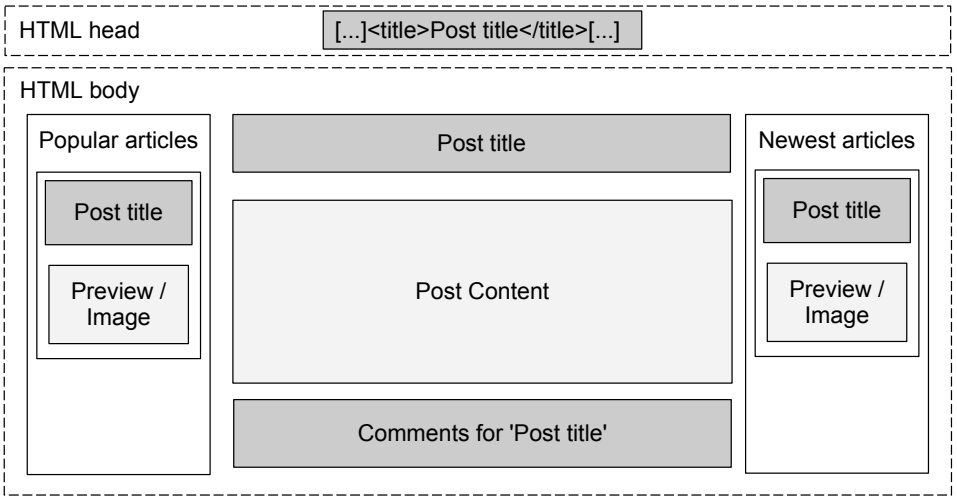

Figure 4: Elements of a page that causes the identification of multiple paths

Listing 4: Identified key paths.

```
(1) |html|body|div[@id=post-*,@class=post]|div
(2) |html|body|div[@id=*,@class=news]|div[@class=content]
(3) |html|body|div[@class=*]|div[@class=*]
(4) |html|body|div[@id=post-*,@class=post]|div[@class=head]
(5) |html|body|div[@id=post-*,@class=post]|div[@class=body]
```

Lets assume the path finding algorithm has found the key paths for the host shown in Listing 4.

The first path matches the page, while the second one does not. It is possible that nodes in a page contain additional attributes that are not part of the path. But if a path specifies an attribute for a node it must be within the page as well.

The third path is an example for path being too general. There are two problems that can occur in this case:

1) The key path matches a non-post page (false positive)
2) The wrong node is used for extraction and thus the wrong property content is extracted

It is very likely that a page contains at least two *div* nodes with the same *class* attribute. As you can see it is important that the path finding algorithm does not create paths like this to prevent false positives and wrong property extraction. Path four and five are examples for sufficient paths for information extraction.

Path four points to the div node that wraps the whole title of the post (*A blog post Subtitle*), while path five would point to the post content. Note that the only difference between these paths is the class attribute of the last *div* node. This shows how important it is that a path contains as much information as possible.

IV. Evaluation

We develop a stable prototype that implements the presented template generation concept and is used for our test runs. First, we evaluate the detection performance and discuss challenges of real world data. Second, we show that the quality of the XML property extraction for the example of the content property. Hereby, we empirically show that our algorithm can outperform a well established content extraction framework namely boilerplate [3]

For our evaluations we use a dataset provided by the BlogIntelligence project [4]. We choose this project because there are using a feed-focused blog crawler to collect post pages.

A. Detection performance

In this section, we discuss the result conceding precision of our post HTML page detection. We randomly sample 5000 pages from the project's database. The pages are a mixture of 2217 actual weblog pages as identified by the *generator*-tag of the HTML page.

Listing 5: Exemplary Generator Tag.

```
<meta content='blogger' name='generator'/>
```

We investigate the remaining pages and conclude that these are mainly forum and news portal pages that actually also consists of a feed.

Now following are the main observations we made as we evaluated the result set shown in Table 1.

The result shown in Table 1 are very promising. We nearly identify 81% of blog pages as posts. Every blog consists of

[3] code.google.com/p/boilerpipe/
[4] http://www.blog-intelligence.com/

Sample set size:	5000
Blog pages :	2217
From these pages detected as post:	1814
Number of pages with extracted content and title:	790
Number of pages with extracted content only:	302
Number of pages with extracted title only:	722
Number of pages which have an author:	125

Table 1: Results of a first test run

posts, but it also has landing pages, archive pages, imprint and other pages. We conclude that our precision is actual very high.

Further, we show the number of extracted content and title fields. We have to mention that these results greatly depend on the quality of the published feeds, since we completely rely on feeds to find paths and extract content. Many feeds have a surprisingly low quality when it comes to the content of their items. Some have inline Javascript elements, miss fields or have the title in the content field and vice versa. For feeds which are well-formed (contain title as well as content information) we have a high rate of paths which point at the correct nodes and have many attributes which uniquely identify the key-path.

A different problem we encountered are comment feeds. Many blog post have a feed only with the comments users wrote for this post. These feeds can also produce key-paths that distort the extraction. This does not affect the detection because comment are contained inside the html post pages and thus can be successfully identified. However, comment feeds do not allow the correct information of the blog post to be extracted and instead grab the contents of the corresponding comment.

Overall we can say that our detection approach works good for well-formed feeds, but there is still work to do when it comes to fine tuning such as thresholds and smaller heuristics like the truncation of paths (as described in Section V), which can greatly impact the extraction results because even missing the right container by a single node can cause the extraction step to extract too much or less information form the post.

B. *Extraction performance*

Secondly, we compared a feature of our approach, the content extraction, with a state-of-the-art solution *boilerplate*. We initiated a user survey with 30 participants from our institute. All of them have strong IT and web development background. For the survey, we crawled a data set of 7000 posts with a seed list of the top100 German blogs [5]. We randomly selected 600 of these posts and executed our content detection and the boilerplate content extraction to get the two different texts. Each participant gets the original web page (integrate as iFrame) and, in a random order, the text of boilerplate and of our approach. The participants have the option to select one solution as most appropriate or mark both as inappropriate for 20 pages per participant. The setup is shown in Figure 5

The results of the survey are shown in Figure 6, 323 select adaptive post and 192 boilerplate as best solution. 85

[5]http://www.deutscheblogcharts.de/

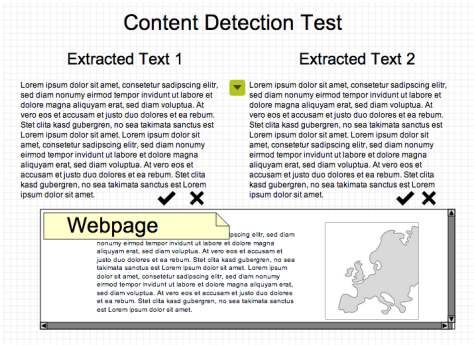

Figure 5: Experimital Setup, users can mark the appropriate content

of the evaluate posts could not be detected successfully. Thus, adaptive post performs 22% better than boilerplate in our survey.

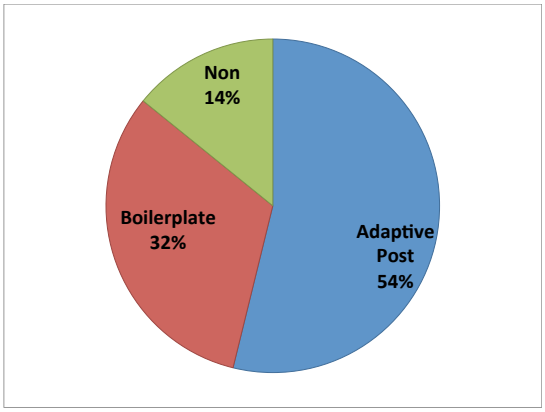

Figure 6: In the set of 600 pages, the 30 participants marked 323 times adaptive post as most appropriate solution.

We also like to mention some of the comments the participants made during the small study. One participant mentions that it feels like one technique has a high precision, but also a comparable low recall. He feels like one of both is totally wrong or produces a exact match to the content. By investigating the trail of the participant, we found that this comments actually apply to our technique. In case of a wrong match, our method creates a key path that actually points to a totally different element.

Another participant criticizes that one method (e.g. boilerplate) frequently includes unnecessary information into the content. For example, the content also consists of the publishing date, the text of Facebook or Pinterest sharing button and also the author name. By manually sampling our dataset, we also observed the wider focus of boilerplate. Although boilerplate do not miss text of the content in these case, it definitely produces a wrong content property for a post.

V. Future Work

During the extraction of dates, we face the problem of different format in the feed and HTML. For the publication

date extraction the matching algorithm should be able to recognize the format of the date from the feed and convert it in a set of common date formats. These converted dates can then be used to find the right node in the DOM-tree.

To find the category/tag paths the algorithm should create separate paths for each tag and than merge them to find a common path. A problem that could arise is that the post content can also contain these tags since they are just single words. To determine similar paths from the result sets could be a good way to solve this problem.

We presented how to find a path to the container node for the different feed properties in Section III-D2. Future work shall expand the algorithm from the textual HTML structure to the visual structure of a blog post to improve the results of the path finding algorithm, as described in [16]. This idea could also be used to improve the separation of attributes.

VI. Conclusion

In this work we presented an approach for automatic detection of blog posts and to extract their content together with meta information. This approach replaces the creation of tailor-made parsers for different templates and blog systems by incorporating the common standard syndication format e.g. feeds.

Based on the approach of Chakrabarti and Mehta, we create paths to cluster web pages . For that purpose we utilize the feeds of a host to create characteristic paths for every information we want to extract and promote a key-path for clustering, using a simple yet effective path model to represent and manipulate paths. Further, we discuss how to merge the various paths of a page to get a single key path and how we can use this key path to identify a post.

We evaluate the detection performance of our approach and discuss the influence of feed quality. Further, we introduce improvement that can circumvent detection drawbacks. Finally, we compare our content extraction with the boilerplate content extraction. Our user survey showed that our template generation method performs ca. 22% better.

References

[1] N. Incite, "Buzz in the blogosphere: Millions more bloggers and blog readers," 2012, [Online; accessed 7-February-2013]. [Online]. Available: http://www.nmincite.com/?p=6531

[2] J. Bross, K. Richly, P. Schilf, and C. Meinel, "Social physics of the blogosphere," in *From Sociology to Computing in Social Networks*. Springer, 2010, pp. 301–321.

[3] S. Wasserman, *Social network analysis: Methods and applications*. Cambridge university press, 1994, vol. 8.

[4] P. Berger, P. Hennig, J. Bross, and C. Meinel, "Mapping the blogosphere–towards a universal and scalable blog-crawler," in *Privacy, security, risk and trust (passat), 2011 ieee third international conference on and 2011 ieee third international conference on social computing (socialcom)*. IEEE, 2011, pp. 672–677.

[5] R. Ferreira, O. Holanda, J. Melo, I. Bittencourt, F. Freitas, and E. Costa, "An agent-based semantic web blog crawler," in *Proceedings of the 7th International Conference on Information Technology and Applications. ICITA, Sydney*, 2011.

[6] B. Hammersley, *Developing feeds with RSS and Atom*. O'Reilly Media, Inc., 2013.

[7] A. Finn, N. Kushmerick, and B. Smyth, "Fact or fiction: Content classification for digital libraries," 2001.

[8] M. Marek, P. Pecina, and M. Spousta, "Web page cleaning with conditional random fields," in *Building and Exploring Web Corpora: Proceedings of the 3rd Web as Corpus Workshop, incorporating Cleaneval*, 2007, p. 155.

[9] 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*. ACM, 2010, pp. 441–450.

[10] Z. Bar-Yossef and S. Rajagopalan, "Template detection via data mining and its applications," in *Proceedings of the 11th international conference on World Wide Web*. ACM, 2002, pp. 580–591.

[11] L. Yi, B. Liu, and X. Li, "Eliminating noisy information in web pages for data mining," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 296–305.

[12] N. Kushmerick, "Wrapper induction: Efficiency and expressiveness," *Artif. Intell.*, vol. 118, no. 1-2, pp. 15–68, 2000.

[13] V. C. A, P. M. A, and P. M. B, "Clustering web pages based on their structure," 2004.

[14] D. Chakrabarti and R. Mehta, "The paths more taken: matching dom trees to search logs for accurate webpage clustering," in *Proceedings of the 19th international conference on World wide web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 211–220. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772713

[15] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 20, no. 5, pp. 522–532, 1998.

[16] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma, "Extracting content structure for web pages based on visual representation," in *Proceedings of the 5th Asia-Pacific web conference on Web technologies and applications*, ser. APWeb'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 406–417. [Online]. Available: http://dl.acm.org/citation.cfm?id=1766091.1766143