Methods for Web Content Analysis and Context Detection

Bryan Mierzwik, Charles Hanchett, Donovan Finch, Jonathan Hasbun, Katie Abrahams, Kiarash Torkian, and Tyler Sorg

§: All authors contributed equally to this work.

***Abstract*—This paper focuses on highlighting current methods for web content analysis, their constraints, and how they can be improved upon in order to support the process of minimum contextualization. Further, the process of minimum contextualization (referred to also as contextualization) is defined by the implications it has on four core problems: accessibility, data usage, maintaining data context and the Internet of Things. This process is based around the idea that accessibility is central to the Internet and that by pre-processing web content one can have greater access to the Internet regardless of variable constraints such as geographic location, data costs, target devices and disabilities. The specific problem which contextualization solves revolves around how data is processed and the restrictions faced in its consumption when put in context of the system in which it resides. Through our research we were able to identify several emerging technologies which could facilitate this process in a reasonable manner.**

Keywords*—contextualization, web content, content analysis, accessibility, optimization, iot.

I. Introduction

The way people interact with the Internet has changed drastically over the past few years, and the devices which interact with the Internet have seen an equally drastic increase in diversity. There are many considerations that need to be taken into account when designing a modern website, the following are those that will be addressed in this paper: what is the quality of the users Internet connection[1], what is the target device of the user, what content is important to the user, and is the data accessible to those with disabilities. These are all complex problems that can incur a huge cost on web development and individual users who have restricted access to the Internet, however by utilizing a process called minimum contextualization, which will later be defined, one can easily abstract these problems into a simple solution.

It is important to note that this paper was prompted as part of a Mozilla sponsored senior capstone project at Portland State University. The process outlined in this paper is meant as a proof of concept for future use by Mozilla products and projects; the process would implement minimum contextualization and for the purposes of this paper is referred to as the "Phoenix Browser." This paper is focused on analysing existing methods that may be used to solve one or more of the aforementioned problems as well as their limitations and potential ways to leverage them into a new process. Additionally, the process of minimum contextualization will be defined as an outline of how to implement it and further research topics that should be pursued.

[1]This refers to users with limited data, where ISPs offer less data or offer data at higher rates

II. Background

Before exploring how to solve the four core problems this paper addresses, it is necessary to understand why these issues matter and how they are interconnected. The issue of which device a user is accessing the Internet from limits the data which can be consumed; the wide-variety of hardware makes it impractical for developers to optimize views for every conceivable device, especially with the coming of the Internet of things. This in turn determines which content is absolutely necessary in order to understand the context of the data on a given site. Not all content on a site is necessary to understand what data it is presenting, such as ads. Once the desired data has been identified it is possible to ensure that this data can be made accessible to those with disabilities. Finally, the last and possibly most important issue revolves around the previous issues: data usage. By identifying extraneous data it is possible to limit the data usage required to view certain sites and by properly formatting this data it is possible for the original context to be preserved. In addition to the core issues it is also important to get a brief overview of the tools that built the foundation for this research: the document object model API, Phoenix and reader modes.

A. Document Object Model

The Document Object Model (DOM) is a programming interface that allows the manipulation and interaction with objects in HTML, XML, and SVG documents. It is often displayed as a tree (DOM tree) consisting of nodes each having their own unique methods and properties which as a whole represents the entire structure of the document. The nodes can then be individually modified to change the structure, style, and content of the document. The fundamental purpose of the DOM is to connect web pages to programming languages for further modifications which often is done in Javascript. Many powerful tools leverage the DOM API due to the control it allows one over the content, style, and functionality of a site.

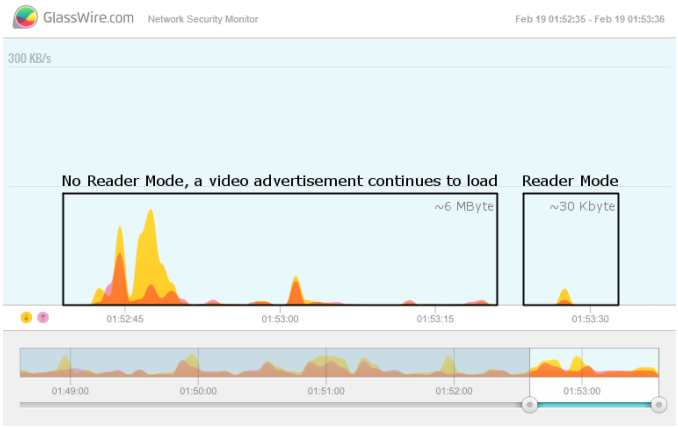
B. Phoenix

Phoenix is a tool built by our research team for the purpose of our research. It simply grabs the DOM of a site and displays

it as a tree[2]. Use of this tool will be described in further detail later in the paper. Reader modes are a general tool implemented by many of the major browsers, such as Firefox, Chrome, Safari and Internet Explorer. These tools vary in their implementation and functionality, however they generally serve one purpose. They are primarily used to extract text from a page, remove extraneous content and display that to the user. Though this may sound simple it is the basis for our paper and a good starting point for contextualization.

C. Reader Modes

Reader modes are powerful tools that allow the stripping of extraneous content so a user can more easily read the content of a web page. They are often an extension of a browser; some of the most popular browsers, including Chrome and Firefox, have Reader Modes on both their desktop and their mobile platforms. Reader modes can be activated or deactivated to either show an optimized version of a page or its original one. Some of the most powerful ones also allow additional modification to a page based on the users preferences such as changing the texts font, size, and color or the background color of a web page.

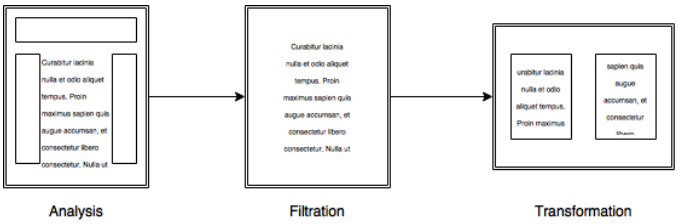


III. Methods of Analysis

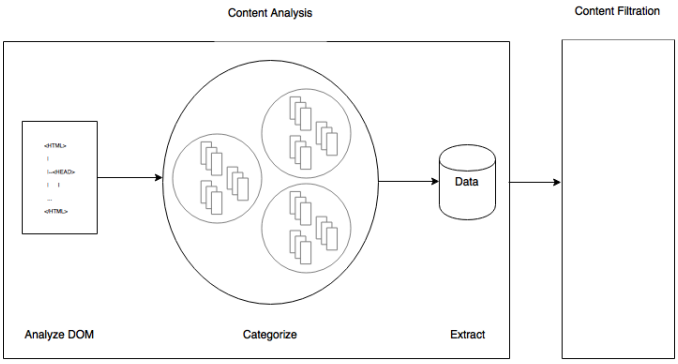
A. Phoenix

The goal of the Phoenix Browser is to provide Firefox OS with the capability of contextualization[3]. In theorizing this method it became clear that there were three distinct steps associated with contextualization: content analysis, content filtration, and content transformation. It makes sense to separate content analysis from transformation. By creating a filtration step it allows for a layer of abstraction between the user's query and the actual data that is returned.

The first step of the contextualization process is content analysis. Content analysis determines what type of content is on a website. Once the structure of a website is determined, the type of site can be determined through a set of heuristics. The results can be fed to the content filtration step. Content filtration is closely tied to content analysis. However, where content analysis is concerned with discovering a hierarchy of data within a site, content filtration is concerned with how much of that data to collect and send to the client. The final step in contextualization is content transformation. This process is facilitated by the client in order to produce the desired results and is specific to each target device.



As this paper will show, the current methods utilized by Reader Modes are severely limited in their ability to detect content other than text and are not suitable for achieving true contextualization. The process laid out in this paper can be used to build off of this existing body of work and greatly expand its functionality. In focusing the scope of this project, it was decided to concentrate on the content analysis process. This decision was made based on the restrictions inherent in this process as a whole. Content filtration requires access to the inner workings of a browser (specifically, the Firefox Browser in this case) in order to filter data and send only the desired content to the client. As for content transformation, this is a process that is specific to the target device and must be determined by the client. The theory for this project involved taking a two-step approach to content analysis. In the first step, sites would be clustered into categories based on the type of site[4] then the relevant data would be filtered and downloaded by the browser.



The original design was to be able to cluster various

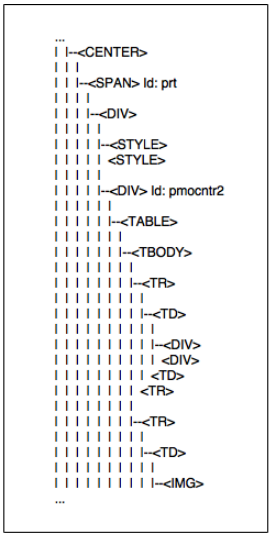
[2]Source code for this tool can be found at: https://github.com/Cap7pdx/phoenix-node

[3]Contextualization or minimum contextualization, for the purposes of this paper, will be defined as the process of: analysing, filtering and transforming the content of a site into the desired form dictated by the client.

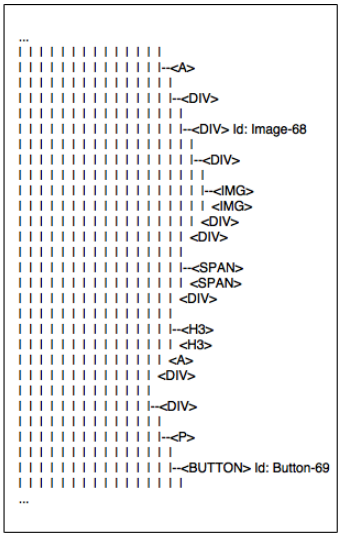
[4]For the duration of this paper, reference to the "type" of a site, is a reference to a specific category of site as defined arbitrarily by this paper.

types of sites together. The method chosen to do this was to parse through the DOM of a website and determine the underlying structure of the site. Using this structure and a set of heuristics, the site would be placed into a cluster of similar sites, such as news, social media, games, etc. This step would be performed during content analysis. Once a website is categorized into a cluster, it is easier to extract the content from the site. Each category has defined rules for what data is necessary to preserve the importance of the content.

Initially it was believed that in order to cluster sites into categories by type, one needed to analyze the structure of a site. The underlying thought being that a site which hosts similar content will have an inherently similar structure and therefore be processed in a similar manner. However, it became difficult to empirically define a set of categories based on type alone. These categories became more subjective and arbitrary groupings based on human perception of what classifies a site under a specific grouping, as opposed to objective and empirical groups that would enable automated classification. Additionally, the structure of sites with a similar type varied greatly. For example, when using the Phoenix tool built for this research, it was found that both the Huffington Post and Pinterest had a similar structure.



Sample Output from Phoenix: Huffington Post



Sample Output from Phoenix: Pinterest

Both these sites have a similar layout (nested divs used to bundle images with text/links, in a tile formation), however on Huffington Post the images are unessential, in many instances these images are stock photos that are loosely tied to the actual article, if at all, and without them the user can still understand the remaining content. On the other hand, images on Pinterest carry more value to the user in terms of understanding the content and in some cases may be even more important than the text. This was a pattern with several other pairs of sites that had been thought to belong in different categories. Due to this, other methods of categorization were pursued.

In this paper, we outline approaches to analysing web pages in order to understand and manipulate the content, generally with the intention to filter out unwanted or unnecessary content. These approaches can be separated into authorial approaches that require human input in order to make correct decisions and algorithmic approaches that dynamically adjust to content. Authorial approaches are approaches that require a human being viewing the web page at some point and making a decision about how the algorithm should behave. Authorial approaches discussed include specifying external standards about how relevant content is included on a page, which requires the original author (or an editor) ensuring a page conforms to standards, and specific white-listing of pages for a template for transformation (such displaying only the video on a page from YouTube), which requires an external decision maker to decide what template to use for a given page.

The algorithmic approaches that dont rely on human guidance all broadly have three phases: content analysis, filtration, and transformation. Content analysis identifies attributes about content on the web page; filtration determines whether content will be displayed based on attributes; and transformation determines any structural changes needed to ensure coherence in the final web page.

In further defining the details of how a site could be categorized, several existing methods were explored. Of particular note was the readability algorithm utilized by many popular reader modes, as well as the most popular reader modes themselves and semantic segment detection.

B. Reader Modes

1) Readability: Readability is a popular algorithm for extracting the main content of an article. The rough algorithm, described in Kovacics diploma thesis, “Evaluating Web Content Extraction Algorithms,” is outlined below.

Each of the paragraph tags from the DOM are scored based on text density, and the high scoring paragraph hierarchy is elected as the top candidate. These are hand-crafted, arbitrary heuristics that happen to work well through intuition and experimentation [15].

Readability has become proprietary software, but its basic algorithm has inspired Safari and Firefox reader modes [1][14][3]. The reader modes have varying results between browsers, but their basic functionality is the same.

Readability, at least in the Firefox version, categorizes web pages based on more arbitrary, simple heuristics [3]. If a page is probably suitable for representing as an article type format in reader mode, then the main content text is extracted. If the page is not eligible, it is not categorized as another type, such as a list. This simple distinction is not enough to identify eligible list-type websites or which elements of the page are relevant, for they could have similar structures but different meanings.

Algorithm 1 Readability Algorithm Outline

Require: $h \leftarrow$ HTML Source code
1: $dom \leftarrow$ convert h to DOM
2: $dom \leftarrow$ remove `script` nodes
3: $title \leftarrow$ heuristically extract the title from $\langle title \rangle$ node
4: $dom \leftarrow$ remove nodes whose class names match a hand-crafted list of non-content class names
5: **for** each paragraph p in dom **do**
6: `// proceed by assigning a score to each paragraph`
7: $p.score \leftarrow$ assign a fixed score based on text density
8: `// propagate scores to ancestor nodes`
9: $parent \leftarrow getParentNode(p)$
10: $parent.score \leftarrow parent.score + p.score$
11: $grandParent \leftarrow getGrandParentNode(p)$
12: $grandParent.score \leftarrow grandParent.score + (p.score/2)$
13: **end for**
14: $topCandidate \leftarrow$ extract the top candidate node based on propagated scores in the previous FOR loop
15: `// Heuristically determine if sibling nodes contain content`
16: $topCandidate \leftarrow findSiblingContent(topCandidate)$
17: **if** $length(topCandidate.text) \leq 249$ **then**
18: repeat the procedure with relaxed parameters
19: **else**
20: **return** $\langle title, topCandidate.text \rangle$
21: **end if**

2) Chrome: DOM Distiller is the equivalent to a reader mode in the desktop and mobile versions of Google Chrome and Chromium. It is not enabled by default in any version, so it is assumed to still be experimental, approximately a year after it was introduced [7]. It comes in two parts, the C++ code within the browser, and the Java project DOM Distiller which is compiled to JavaScript before being included in the browser.

The browser component uses an AdaBoost Machine Learning classifier, trained by humans, to quickly determine whether an individual page can be parsed by the DOM Distiller. This "distillable" property is determined by people viewing the static rendered results of a DOM Distiller run beside the normal text. This classifier is claimed to provide much greater speed than the alternative - running the DOM Distiller upon every page at page-load-time and determining from the exit status and result if the render succeeded [4].

The Java core is open source and located on GitHub. The DOM Distiller Java project is based on the work of Dr. Christian Kohlschtter, "Boilerplate detection using shallow text features," and the code produced for it, Boilerpipe [2]. Reading said source, it contains several hard coded conditions for distilling a page. One is the presence of the HTML5 Article tag, the other is a strictly literal regular expression for common situations in news websites - the presence of a comment section in the body tag. This is assumed to be after the bottom of the article, which terminates further parsing [4].

Once the browser classifier determined that a page can be distilled, it is run through the DOM Distiller which pulls out interesting content - the page title, links, images, and paragraphs of text. This content is placed into an HTML5 Article template page, filling in the page title, then each in-line image and paragraph of text in order. The content is re-wrapped in new tags with slightly different properties. The page URL shown to the user changes to a "chrome-distiller://" to indicate its local source, and the text width is fixed at 35em. Firefox's much more polished Reader Mode does not change the URL, but it likewise generates a new page with the same fixed width which makes viewing worse on large displays.

Current reader modes focus on one type of content (news articles) and consider only two of the concerns we focus on: fitting the content to the target device, and displaying only the important content. Both require that the browser download and render the page normally, before presenting the user with the option to view it in either Reader Mode or "Mobile-friendly view", respectively, so there is no bandwidth savings. It does nothing to improve the accessibility tools access to the content - any success in that regard would be in the simplified code of their templates. More sophisticated website structure detection and contextualization models are needed to make the rest of the Internet more accessible, lightweight (low data consumption), relevant, and universal.

C. Shallow-Text Features

In Boilerpipe, text is not inspected at the topical level – rather, it is inspected at the functional level. That means that the words in the text are not classification features; doing so would skew the results toward that pages domain only [2]. Instead, Boilerpipe aims to remove boilerplate HTML by considering only shallow text features, including average word length, average sentence length, local context, and several other heuristics features [2].

These features are simple to calculate. Average word length is the average length of a non-whitespace string, and average sentence length is the average length of words between full stops, question marks, exclamation points, and semicolons. It is also based on the absolute number of words.

The local context is the absolute and relative position of a text block in the document. If the granularity of the pages segmentation is high, [2] it is inferred that full-text is followed by full-text, and template followed by template. With lots of boilerplate text, the main-content text was usually found between boilerplate HTML and not the other way around.

The heuristic features included the absolute number of capitalized or all-caps words, the ratio of capitalized/all-caps words to all words, the ratio of full stops to all words, the number of date/time related tokens, the number of "—" symbols (used commonly in navigational boilerplate text), and link density (also referred to as "anchor percentage").

Kohlschtter et al. [2] concluded that "removing the words from the short text class alone already is a good strategy for cleaning boilerplate and that using a combination of

multiple shallow text features achieves an almost perfect accuracy." Also, detecting boilerplate text usually did not require a complicated machine learning model or inter-document knowledge, such as frequency of text blocks or common page layouts. The costs were negligible – essentially just counting words. They go on to say that, "surprisingly we observed high performance for methods that followed intuition and experimentation in contrast to those that relied on complex mathematical models and scientific approaches." [2]

Their results are great for article-like websites, but this intuitive and experimentally validated method for extracting main content text relies on the way articles are usually represented in HTML. It is a good shortcut for minimally representing a subset of all the websites on the Internet. If some website detection algorithm designates a web page as an article, this is a lightweight, practical way to extract relevant content.

D. Semantic Segment Detection

The semantic segment detection algorithm works on the assumption that the structure of an HTML page implies something about the content of the page. The algorithm analyses the structure of the HTML page retrieved from the provider and identifies structurally unified portions of the page, and caches this information. When the page is to be rendered, the algorithm uses the specifications of the device, such as screen size and pixel density, as constraints for restructuring the page in the most efficient arrangement for the given device.

The semantic segment detection algorithm works on an abstract level by classifying all HTML tags (called syntax units) into a set of categories (called semantic units) according to the meanings determined by the algorithm. The developers of the algorithm have identified five different classes of semantic unit: dummy, element, object, block, and body. Elements are basic semantic members on the HTML document. An element is then composed of a text element and an embedded element, where the text element is any text content in the parent element and the embedded element is any non-text elements (images, video, etc.) in the parent element. The object semantic unit refers to a unit that contains within it elements and other objects that have similar properties, with the properties identified as functional similarities (sections encapsulating functions such as anchoring), presentation similarities (such as presentation styles), and semantics similarities (such as structural relationships). Block semantic units separate and arrange other semantic units on the web page. Block semantic units can be divided into wrapper types and container types, depending on their contents. The body semantic unit refers to the entire content of the HTML page. Lastly, the dummy element refers to any element that has no presentation information such as white spaces.

To actually classify a page into semantic units, the developers of the model have determined a hierarchy of structures in real world HTML pages. From the bottom up, six levels were determined and matching steps were implemented in the algorithm. The algorithm uses the structure of HTML tags and CSS properties to generate an annotated tree with structure and display information. This tree is then analyzed, and similar tags are grouped together. The developers of the algorithm note that the implementation only takes into account attributes that affect appearance and arrangement, though there is clear possibility for analysis of attributes such as authorial style. The algorithm then analyses the coherence of the determined groups. The algorithm bases coherence on literary context (similar words), similar functionality, similarity in time and space, and similarity in typesetting. All of these attributes are used to determine how easily groups of content can be separated from each other. These grouped sections of content can then be used to redistribute or remove content from the page.

The semantic segment detection algorithm is not a solution for identifying relevant content on a page; it instead identifies connected groups of content. However, the connected groups of content could then be compared using a more simple algorithm to determine which group or groups should be displayed. The paired algorithm would not need to be as complex as those mentioned in this paper, as it would not not need to analyze a fully formed web page. It could instead focus only on text attributes and content in isolation.

IV. Observations & Discussion

None of the above methods can alone achieve the desired results of limiting data usage, adapting to a wide range of devices, filtering the most important content, and ensuring accessibility to all. While basic reader modes are sufficient for a small set of the data usage problems and an even smaller subset of accessibility problems, they are not sufficient for sites with more complex forms of content. Reader modes can significantly cut data usage to sites which are text heavy, and this data is easily filtered to suit a wide-variety of devices. The downside is that the set of sites in which reader modes work are limited significantly. The desired outcome of this paper is to find a solution which is capable of analysing all types of content for all sites. Shallow-text features have similar limitations as most modern reader modes (such as those previously covered) as does the semantic segment detection method (though this method works on a larger set of sites due to its implementation). After having explored these methods, it is evident that the original process of extracting the minimum data from web content while still preserving context can be further improved by building off this foundation.

The initial idea of categorizing sites by type turned out to be inefficient; it makes more sense to cluster sites into groups of similar structure. That way each group will be handled in a similar manner. This grouping can be handled by several machine learning techniques, specifically cluster analysis (further discussed in the conclusion).

Once sites are grouped, one of several methods can then be used to filter the important content, which lowers data

usage. The two most promising methods for filtration are implementing standards (which is described in Appendix B) or a modified version of the semantic segment detection. This content can be transformed and represented in a common theme specific to the device and the data presented in an accessible manner if necessary.

V. Conclusion

After having examined the various methods in this paper it is clear that research into this topic (content analysis of sites, beyond text evaluation) is still relatively new and unexplored. This means that the potential is far greater than previously thought. The process of minimum contextualization can have a significant impact on the way people interact with the Internet, and will be far more sophisticated than currently available tools. Such a process would have the same impact on data usage as reader modes, but would extend to any site, not just text primary sites. Users would be able to digest the same information on any device in a manner tailored specifically to that device. With several transformations this data can be made more accessible, either configuring the contrast and view or creating a unified audible of the contents of the site (not just text). This means, for example, that a user accessing a site such as Facebook or Pinterest will be provided with the same information whether they access it from a tablet, a refrigerator with an LCD screen, or even a device without a screen. Though the researched tools are not capable of all this, they have created a promising foundation for implementing such a process and helped indicate which path is likely to produce the desired results and how to go about it.

As stated previously, the research conducted in this project showed that it was unproductive to base a sorting algorithm for sites solely on the type of content contained; rather, the research points towards promising results for structure-based categorization during the content analysis phase of contextualization. By first sorting sites by structure, as opposed to first sorting by an arbitrary type, a group is formed with similar attributes. Site content does not imply structure, though structure may be a good indication of content. By further categorizing sites by type, one can filter the necessary content effectively. In order to productively facilitate this process of content analysis there are several methods that can be utilized; these are detailed below.

The first method involves the use of standards and regulations (discussed further in Appendix B) to dictate the development of elastic sites[5]. While this makes it simple to add contextualization functionality to a browser, it also involves a large amount of work from a business perspective to get developers to buy-in and adhere to these standards. This is a possible implementation but not our recommendation.

Another method, encountered near the end of our research, utilizes a machine learning technique called cluster analysis to group like objects. Such a tool could learn about the archetypal structure in a cluster and group sites with similar structures together.

After the necessary grouping has occurred (via either method) a process similar to the semantic segment detection algorithm can be implemented to grab the desired content from a site. Alternatively, a white list approach can be taken. Rules can be set for what kind of data is considered important for each type of site. Once the content analysis phase is complete, the rest of the contextualization process can be implemented trivially.

By following the cluster analysis process, we believe it is possible to fully implement minimum contextualization. Data usage can be significantly cut by filtering what data to download from a site as shown by existing reader modes. Content can be detected by a site's structure as proven by the semantic segment detection algorithm and data can easily be transformed by the target devices UI implementation as well as increasing accessibility through the same means. With several open source tools (or built in-house based off existing tech) Firefox OS can viably implement contextualization.

In addition, this approach of categorization (aided by cluster analysis) allows this problem to further be modularized. While basic reader modes are not suitable for most sites, they are incredibly efficient for a subset of sites; those in which text is the only important content. For example, when a site is detected in which text is its sole or primary content, then basic reader mode functionality can be used (such as shallow-text features). Otherwise a more advanced approach can be taken that is tailored to that specific group of sites.

The Phoenix tool created to aid in researching this topic, although limited in its current form, can potentially be used as a basis for a proof of concept following the recommended methods outlined in this paper. Looking forward, cluster analysis techniques in machine learning are promising areas of research for minimum contextualization. The data gathering and learning curve involved in website categorization lends itself well to machine learning. Rather than being limited to one algorithm or technology, cluster analysis can be used with various techniques and can be adapted to the specific task at hand. A trained machine learning algorithm would likely be better at recognizing and categorizing sites than traditional algorithm techniques. There are already research projects working towards this or similar goals.

Appendix A
Phoenix Browser

The Phoenix browser was intended to be a proof of concept to see if there was a way to dynamically detect the type and categorize a website. This categorization would then be used to extract the relevant content from a site and allow a device to display the extracted content while still maintaining the same context of the original data. This would not only allow for non-mobile friendly content to

[5]This refers to sites which are capable of minimum contextualization.

be displayed correctly on mobile devices correctly, but it would also allow for the use of less data being transferred to the device. By reducing the amount of data that needs to be transferred to a device, the overall cost to the end user could be reduced. This would be especially important in regions where mobile data is extremely slow and expensive.

Originally the Phoenix browser was designed to utilize Mozilla's Firefox OS Browser API. This API extends the functionality of the iFrame interface and allows for more fine-grained control over its management and elevated permissions. However, several issues were encountered through its use. The first, and primary, issue involved Cross Site Scripting (XSS) errors (this will be further explained in the next paragraph). The second was a design issue. By further abstracting the analysis component, it allows the project to be more platform agnostic and serve as more of a proof of concept.

During the implementation of the Phoenix browser application we ran into XSS issues. This was a fairly large setback in the project which ended up forcing us to rethink what we were trying to accomplish and find a better way to do it. Our first version of the Phoenix browser was going to take a URL and retrieve the DOM from an iFrame that we loaded the website in and perform heuristics on the structure of the site. From our research and despite our attempts, this seems to be something that cannot be performed with the Firefox OS Browser API. Due to the security provided by restricting XSS, we were unable to access the DOM of the iFrame. Although being able to perform functions on the DOM of a site loaded into an iFrame would have been good for the project, security and the prevention of XSS is more important than the features it would provide.

Given the XSS issues we ran into, it pushed the project in a slightly different direction than originally planned. We refactored the code to work with node.js with the jsdom node module for the analysis portion of the project. This allowed us to abstract the process away from the device and move it server side, where other methods of analysis (such as machine learning) that would not work well on a device can be performed.

Appendix B
Standards & External Pressures

Another method for broadly implementing site filtration was that of developing external pressures or standards by which sites can broadcast their capability for content filtration. This method is less desirable due to the amount of time this process takes to become accepted, but could be effective. Also, it should be noted that this method deviates from the initial problem slightly; the focus of this paper is on how to solve the four problems mentioned in a dynamic manner. For this reason, it has been placed in an appendix.

In 2015, Google rolled out an update to their search algorithm, giving preference to sites that were deemed "mobile-friendly" according to Googles standards [9]. Due to Google's prominence, this put pressure on many web developers to create sites that were mobile-friendly in order to maintain their site's ranking. Such a standard could feasibly work for a platform such as Firefox OS where web apps would be given preference for adhering to a standard format that promoted the use of contextualization. If pursued, this format would enable Firefox OS to pre-process sites, downloading only the desired content and allowing the target device to reformat this data to best suit its uses.

Standards are an efficient way of maintaining consistent programming practices and functionality. A modern trend that has seen increasing popularity is the use of web frameworks, as opposed to hand-coded sites, to create sites. For example, Wordpress is a common framework used among many blog sites; it has its own consistent coding standards and structure that it maintains, one of which is mobile-friendliness. It is possible that a framework can be made (and existing ones updated) to implement a structure that would be easy to contextualize and take away that burden from any browser's implementation. However, this would require a large effort to create such a framework and make it prevalent. This method was not explored further due to the fact that it diverges from the paper's original goal. The paper focuses on how a browser can be made to implement this process dynamically and for all sites. A framework approach would only be relevant to the subset of sites that used this framework.

Acknowledgment

The authors would like to thank Mozilla for proposing this research topic and sponsoring our efforts.

References

[1] arc90labs-readability.googlecode.com. (2010). *Google Code Archive: Readability* [Online]. Available: http://arc90labs-readability.googlecode.com

[2] C. Kohlschütter et al. (2010). *Boilerplate detection using shallow text features* [Text] Proc. of the third ACM int. conf. on Web search and data mining pp. 441-450.

[3] Firefox, Readability. (2010) *Readability.js* [Online]. Available: https://dxr.mozilla.org/mozilla-central/source/toolkit/components/reader/Readability.js\#1715

[4] Github. (2016). *chromium/dom-distiller* [Online]. Available: https://github.com/chromium/dom-distiller/blob/36a509ca42a49285ad4a770b38a8558f102c3337/java/org/chromium/distiller/filters/english/TerminatingBlocksFinder.java\#L49

[5] *Google Algorithm Change History* [Online]. Available: https://moz.com/google-algorithm-change

[6] Google. (2015). *Finding more mobile-friendly search results* [Online]. Available: https://googlewebmastercentral.blogspot.com/2015/02/finding-more-mobile-friendly-search.html

[7] Google. (2016). *The DOM Distiller is an exciting open-source project created by the chromium* [Online]. Available: https://plus.google.com/u/0/+FrancoisBeaufort/posts/KEcP88z2tfh

[8] Groups.google.com. (2016). *Moving FirefoxOS into Tier 3 support* [Online]. Available: https://groups.google.com/forum/\#!topic/mozilla.dev.fxos/-X5fw4WDr7U

[9] Official Google Webmaster Central Blog. (2016). *Finding more mobile-friendly search results* [Online]. Available: https://googlewebmastercentral.blogspot.com/2015/02/finding-more-mobile-friendly-search.html

[10] Mozilla. (2016). *Google Algorithm Change History* [Online]. Available: https://moz.com/google-algorithm-change

[11] Mozilla. (2016). *The Mozilla Manifesto* [Online]. Available: https://www.mozilla.org/en-US/about/manifesto/

[12] Mozilla Developer Network. (2015). *Introduction to the DOM* [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

[13] S. J.H. Yang et al. (2011). *Applying Semantic Segment Detection to Enhance Web Page Presentation on the Mobile Internet* [Online]. Journal of Information Science and Engineering Available: http://repository.cmu.edu/cgi/viewcontent.cgi?article=1114

[14] Safari, Readability. (2010). *adadgio/safari-reader-js* [Online]. Available: https://github.com/adadgio/safari-reader-js

[15] T. Kovacic. (2012). *Evaluating Web Content Extraction Algorithms* [Online]. Faculty of Computer and Information Science, Undergraduate thesis, University of Ljubljana. Available: http://eprints.fri.uni-lj.si/1718/1/Kovacic-1.pdf

[16] Vinay's Blog. (2008). *Ajax and DOM* [Online]. Available: https://vinaytech.wordpress.com/2008/11/24/ajax-and-dom/

[17] W3schools. (2016). *JavaScript HTML DOM* [Online]. Available: http://www.w3schools.com/js/js_htmldom.asp