Reducing Client-Side JavaScript Evaluation with *JSAnalyze*

Jacinta Hu
Computer Science, NYUAD
jacinta.hu@nyu.edu

Advised by: Yasir Zaki

ABSTRACT

JavaScript evaluation is one of the most costly and time-consuming parts of loading a web page. This project introduces a tool called *JSAnalyze* which automatically analyzes a web page to determine what JavaScript can be removed without impacting the user experience. *JSAnalyze* allows users to better understand the scripts present in their web pages and verify or alter the script removal choices made by the tool before saving the simplified page to an external proxy for future retrieval. Analyzing 100 popular web pages using *JSAnalyze* showed a reduction in overall Page Load Time of more than 30% on mobile devices while retaining more than 90% of the original page content, as evaluated by a user study of 22 users.

KEYWORDS

JavaScript, optimization, web page, user experience, human-computer interaction

Reference Format:
Jacinta Hu. 2020. Reducing Client-Side JavaScript Evaluation with *JSAnalyze*. In *NYUAD Capstone Project 2 Reports, Spring 2020, Abu Dhabi, UAE.* 8 pages.

1 INTRODUCTION

The current status of the World Wide Web (WWW) shows an increasing trend in accessing web pages via handheld mobile devices [8], which is referred to as *Mobile Web* [17]. Since web pages were originally designed for desktop/laptop computers, the mobile web user experience has become a major concern due to two main reasons: the complexity of the pages, and the processing limitations of mobile devices.

The increasing complexity of the web pages has caused considerable increases in Page Load Times (PLTs) [27], which are mainly affected by the download cost of pages' resources and their associated processing cost. During the last decade, the total average download size per page has increased by 300% [5].

Among all downloaded resources, the most dominant category in browser processing is JavaScript. This is because byte-for-byte, the parsing and execution of JavaScript is more computationally expensive than that of any other web resource [24].

Despite its impact in performance degradation of web pages, the current status of the WWW shows a median use of 18 external JavaScript elements on mobile and 19 on desktop. These elements triple the processing time on mobile devices compared to desktop computers [2].

In this paper, we propose *JSAnalyze* as a tool to help web page owners simplify an existing page for mobile web. This is achieved by removing all non-essential JavaScript elements from the existing page without sacrificing page content. *JSAnalyze* allows users to enable or disable each script element on a web page via a Graphical User Interface (GUI) and see these changes in a side-by-side comparison between the original page and the modified page.

2 RELATED WORK

As web pages become increasingly complex, research has turned towards addressing this growing issue, and particularly to reducing PLT [23]. Some researchers have found ways of recreating the dependency graph of a web page to better study the inefficiencies present in modern web pages. Wprof is an in-browser profiler that details the dependency graph of a page load, and the researchers who created it found that JavaScript plays a significant role in page load time by blocking HTML parsing [25]. Shandian attempts to improve PLT by restructuring the page-load process [26], and Polaris builds upon that with the help of Scout, which tracks

This report is submitted to NYUAD's capstone repository in fulfillment of NYUAD's Computer Science major graduation requirements.

جامعة نيويورك أبوظبي
NYU ABU DHABI

Capstone Project 2, Spring 2020, Abu Dhabi, UAE

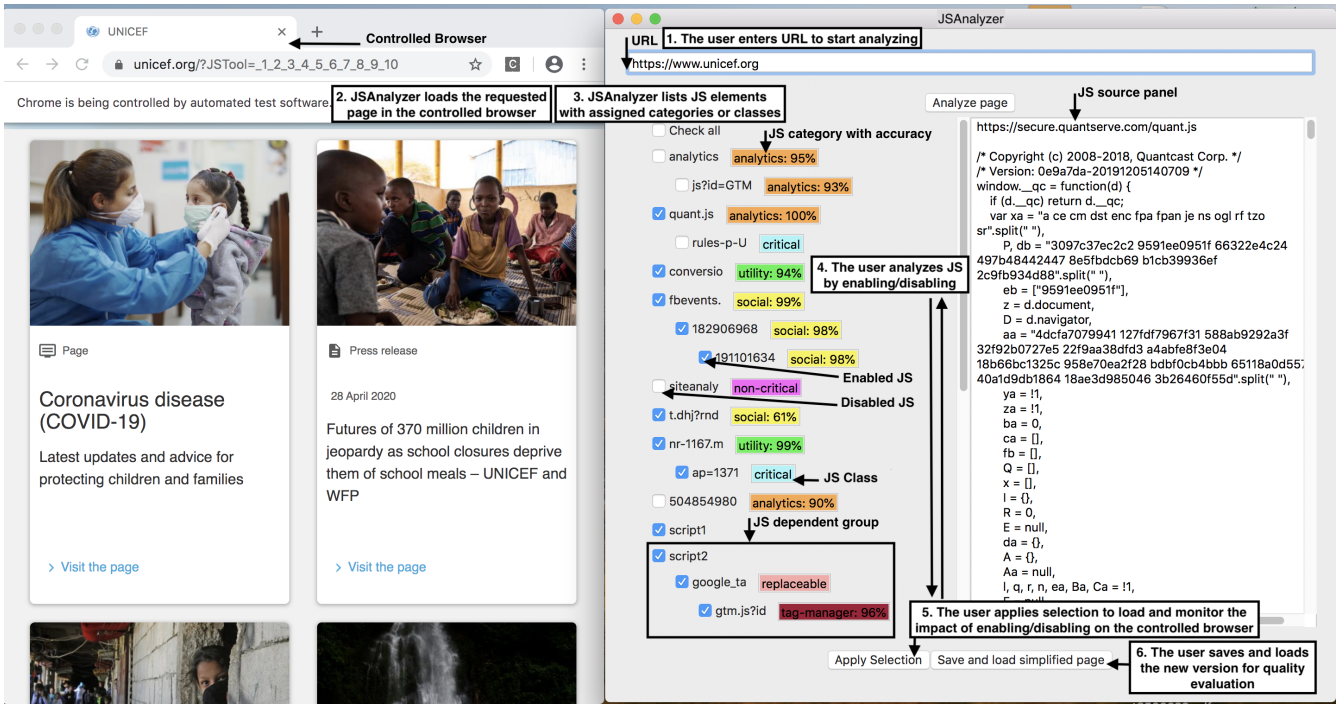

Figure 1: *JSAnalyze* User Interface: From entering a URL to generating a simplified page for mobile web.

JavaScript variables to identify fine-grained dependencies [22].

Other research has proposed entirely new web specifications, such as GAIUS, a content ecosystem targeted specifically for developing regions which suffer most from the cost of JavaScript [14]. Google has also attempted to tackle the complexity of modern web pages through Google Accelerated Mobile Pages (AMP) [18], which redefines how pages should be written. In a recent work, the authors presented the first characterization of the impact of AMP pages on the user experience [20]. A major difference between *JSAnalyze* and GAIUS and AMP is that GAIUS and AMP provide frameworks for developers to create new pages, whereas *JSAnalyze* aims to optimize existing pages for the mobile web.

JSAnalyze aims to fill the gap in existing tools [16, 21] which allow for JavaScript debugging in web pages without attempting to identify the main functionality of each of the embedded elements. *JSAnalyze* optimizes JavaScript usage in web pages according to the detected functionality of each JavaScript element embedded in these pages.

3 IMPLEMENTATION

JSAnalyze is built using wxPython, a cross-platform GUI toolkit for the Python language [9], and Selenium WebDriver, an open-source browser automation tool [7] on top of the Google Chrome web browser. Through interacting with the GUI, the user can request a web page for analysis, remove unnecessary scripts, and save the simplified version of the page to a remote proxy for future retrieval, as seen in Figure 1. *JSAnalyze* implements proxies using mitmproxy [12], a free and open-source interactive HTTPS proxy, and the saving of the page is enabled using an Apache web server set up with the Common Gateway Interface (CGI). This section contains detailed descriptions of each of *JSAnalyze*'s functions and components.

JSAnalyze performs three main functions:

(1) *Detailed script analysis*: *JSAnalyze* displays detailed information about the scripts used in a web page and the ways in which these scripts are used.
(2) *Real-time script pruning*: *JSAnalyze* allows a user to toggle individual scripts to visualize how they affect the look and feel of a web page.
(3) *Simplified page creation*: *JSAnalyze* allows a user to simplify and save a page to an external proxy with unnecessary JavaScript removed, speeding up page load.

3.1 Detailed Script Analysis

JSAnalyze's detailed script analysis includes the script body, prettified and view-able in the GUI window; the network dependencies of the script, represented in the horizontal

tree arrangement of the scripts; and script categorizations, viewed as labels next to the script names.

3.1.1 Script Body. *JSAnalyze* uses Selenium WebDriver to fetch and load the requested URL in Chrome. Unfortunately, Selenium does not have direct access to the JavaScript files that are evaluated during the page load. To access the JavaScript files for further script analysis, the Selenium performance log containing all network activity is parsed for the script sources, which are then requested using Chrome Devtools Protocol (CDP) [3]. The content of these scripts is then prettified using jsbeautifier [10] and saved locally. The content of a particular script is displayed in the GUI when the user clicks the button corresponding with that script. Although a similar result can be achieved using Chrome Devtools directly, displaying the script in the GUI prevents the user from having to leave the GUI to see the script content.

3.1.2 Network Dependencies. When the user requests the URL of a web page from *JSAnalyze*'s GUI, the user is then presented with a multilevel list of all of the scripts that were run or requested during the page load as seen in Figure 1. The levels of the list show the network dependencies of each script. If script A is downloaded as a result of script B, then script A will be listed under script B with a higher indentation level. This network dependency tree is built by parsing the Selenium performance log for each script call and connecting each script to the parent script that called it. The tree gives users a clearer picture of the chain of cause and effect in the page load. Although this tree is generated with calls to CDP, Chrome Devtools does not currently have any comparable visualization of the network dependency tree. Chrome Devtools does include a Waterfall breakdown of network activity which shows when resources are loaded [15], but it does not show how network resources are interconnected the way that *JSAnalyze* does.

3.1.3 Script Categorization. Each script detected by *JSAnalyze* is also given a colour-coded label to specify its category. Third-party scripts are categorized into broad categories such as ad, analytics, social, and video. The full list of categories was taken from Web Almanac's [2] third-party provider categories, and colour combinations were taken from Third Party Web's open-source database [19]. *JSAnalyze* determines what category each script belongs to using a random forest classifier developed by Vladyslav Cherevkov. If the classifier is unable to determine a category with greater than a given confidence threshold, then the script is then classified as critical or non-critical using a clustering algorithm developed by Waleed Hashmi. The confidence threshold is set to 50% by default, but can be specified by the user depending on their preferences.

3.2 Real-time Script Pruning

Once all of the script information for a web page has been loaded, *JSAnalyze* allows the user to enable and disable individual scripts and see how the presence or absence of different combinations of scripts affects the page's appearance and functionality. This is achieved through side-by-side browser windows, as in Figure 2, where the right window fetches the original page from the remote proxy, and the left window fetches the simplified page from the local proxy.

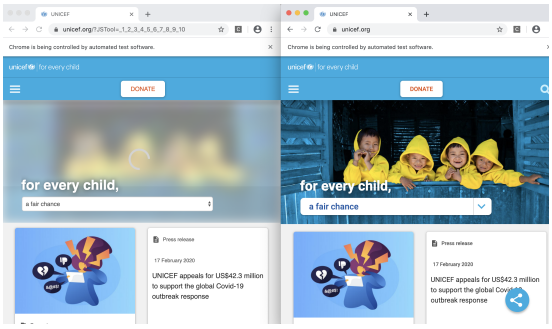

Figure 2: Side-by-side browser windows allow for easy comparison.

By default, *JSAnalyze* disables scripts in the ad, analytics, customer-success, marketing, and non-critical categories as classified during the script categorization, but these categories can be specified by the user depending on their preferences. In the *JSAnalyze* GUI, users can then choose to toggle specific scripts on and off and reload the modified page to see how the changes compare to the original page. *JSAnalyze* maintains the dependencies discovered while building the network dependency tree, and ensures that disabling a script means that all of its descendants will no longer be requested. Likewise, it is not possible to enable a script without first enabling its ancestors.

The rest of this subsection describes how the proxies in *JSAnalyze* are used to fetch and simplify web pages. This is then followed by further details about the way *JSAnalyze* handles the enabling and disabling of different types of scripts.

3.2.1 Proxy Server. A remote proxy server was used so that requested resources could be cached, allowing for faster serving of web pages. When a previously unseen resource is requested, the proxy fetches the resource, encodes and saves the data, and creates a new entry in the SQL database to map the URL to the local filename before serving the resource to the user. The next time a user requests the same resource, the proxy simply looks up the URL in the SQL table and serves the resource from its database without having to make a request to the web server hosting the resource. Caching also

ensures that the page requested does not change during comparison and evaluation. The page saved in the remote proxy is static, whereas the original page on the internet is subject to changes at any time.

The remote proxy ensures that the original page remains the same for evaluation, but for enabling and disabling of inline scripts, a local proxy is used. This decision was made so that multiple users could analyze the same page at the same time through connecting to the remote proxy without affecting the local script changes of *JSAnalyze*. The local proxy also ensures that if the connection to the remote proxy is unstable, scripts can still be enabled and disabled because everything is done locally.

3.2.2 Types of Scripts. When a user fetches a web page, the web server hosting the page first returns the index.html file to the user. Then, as the user's browser parses the HTML file, it may encounter links to other additional source files. These can be additional HTML documents, CSS files, snippets of JavaScript, or other resources. The browser then makes additional network requests to get these files and incorporates them into the page load once they have been received.

Based on the procedure described above, all JavaScript involved in a page load can be placed into two categories: inline scripts and external scripts. Inline scripts are JavaScript that is included directly in the web page's index.html file as HTML <script> tags. External scripts may also exist as <script> tags, but instead of having the script content placed between the starting <script> and ending </script> tags, the <script> tag contains an additional attribute that links to a location on the web where the script content can be found. In this case, the browser's parsing of HTML must be paused until that resource is fetched and evaluated. External scripts can also be found in the index.html file as preloaded <link> tags with an additional script attribute. Because inline and external scripts are loaded in different ways, different methods of script extraction and deactivation needed to be developed for each script category when creating *JSAnalyze*.

Inline scripts. Inline scripts are detected by parsing the original index.html file and looking for places where <script> tags appeared. In *JSAnalyze*, the content between each of the script tags is numbered and stored in a local variable so that it can be displayed later. At the local mitmproxy, each instance of the <script> tag is replaced with a numbered comment so that the script can later be reactivated in the same place. This JavaScript-stripped version of the index.html file is also stored in the local proxy's database so that it can easily be retrieved and altered later. When the user selects the numbered inline scripts that they want enabled, the numbers associated with these scripts are included as arguments in the URL request to the proxy. The proxy server then processes this request by parsing the URL, retrieving the script-free version of the index.html file for the website requested, and uncommenting the scripts that match the requested script numbers.

External scripts. Some external scripts can be detected by looking at the src attributes of the <script> tags in the index.html file, but JavaScript code can also generate URLs and fetch additional external resources. Simply looking for a script source in the HTML tag will not account for all of the other resource URLs that are created dynamically and requested when the JavaScript code is evaluated. Additionally, any external scripts that are requested by the index.html file may go on to request even more scripts of their own, resulting in multiple layers of script calls that would require additional evaluation of each external script in order to detect all of the external scripts. *JSAnalyze* detects external scripts by parsing the network activity logged by Chrome. This includes external scripts with the source included as a <script> tag attribute, external scripts fetched during preload using the <link> tag, and external scripts requested dynamically through the evaluation of other scripts. When the user selects the external scripts that they want enabled, the list of external scripts that are not enabled are blocked by the browser using CDP.

3.3 Simplified page creation

Once the user is satisfied with the changes they have made to the web page, they can save a copy of the simplified page to the remote server for future retrieval and further analysis. *JSAnalyze* saves a copy of the simplified page locally, and then uses an Apache CGI call to upload the page to the remote server. When the simplified page is re-requested, the remote proxy serves the simplified index.html page and performs the external script blocking at the proxy level so that blocked scripts do not need to be sent over the network all the way to the client.

4 EVALUATION

In order to evaluate the effectiveness of *JSAnalyze*, we used *JSAnalyze* to analyze, simplify, and save 100 popular web pages listed by Alexa [13]. These pages were then evaluated for similarity and for speed. To evaluate the similarity of the simplified page to the original page, we conducted a user study and also computed similarity scores automatically using JSQual. To evaluate the PLT of the original pages and the simplified pages, we used Lighthouse [11] and webpagetest [1] metrics. To isolate the effect of limited computation power on page load, these evaluations were conducted on real mobile phones. A Xiaomi Redmi Go was used to represent low-end mobile device, and a Samsung Galaxy S8+ was used to represent a high-end mobile device. For this evaluation, the focus was on JavaScript evaluation reduction,

so network bandwidth was not tampered with to simulate different network conditions.

4.1 Similarity Evaluation

4.1.1 User Study. Twenty-two users were recruited from NYUAD and informed to spend a maximum of 30 minutes on evaluating the similarity of two pages that were shown side-by-side. The user study was conducted online, with all the required explanations available on the evaluation page. Users were asked to evaluate as many pages as they could within 30 minutes. An institutional review board (IRB) approval was given to conduct the user study, and all the team members have completed the required research ethics and compliance training, and were CITI [4] certified.

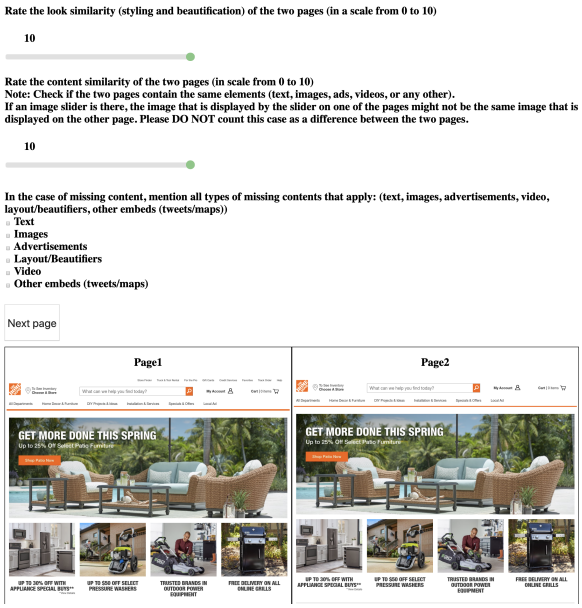

Figure 3: Screenshot of the online user study.

The recruited users were asked to evaluate the similarity between the two versions of the pages, by providing answers to the following questions (Figure 3):

- Rate the look similarity, styling and beautification (on a scale from 0 to 10)
- Rate the content similarity (on a scale from 0 to 10)
- In the case of missing content, mention all types of missing content that apply: (text, images, advertisements, video, layout/beautifiers, other embeds (tweets, maps and etc.))

4.1.2 JSQual. *JSQual* is a structural similarity comparison tool developed by Waleed Hashmi that was used to compute the structural similarity between the original pages and the simplified pages. It computes the similarity not only for the static view of the page, but also takes the effect of some of the page functionality into consideration. Details of JSQual (including source code) can be found on the Github repository [6].

4.1.3 Results. Figure 4 is a combined Cumulative Distribution Function (CDF) of the similarity scores computed by *JSQual* and the user scores collected during the study. The results show that about 90% of simplified pages exhibit greater than 90% structural similarity to their original counterparts, determined by both *JSQual* and the user study. The results also show that *JSQual* computes a comparable similarity score to the user study scores, with minor differences. It can be seen that *JSQual* is generally less forgiving that the real evaluators (users), evident by the smooth and gradual increase of the scores between the 90% to 100% similarity score range. In contrast, the user study results show a sudden and sharp increase in the score within the same segment. There are multiple reasons behind this: 1) real users tend to overlook minute and small differences between pages, 2) real users are more forgiving when giving scores when major parts of the page match the original, and 3) *JSQual* also takes into account the effect of actionable tags when comparing the differences between the pages. Another interesting fact seen in the figure, is that the worst score given by human evaluators was somewhere around 65% compared to 40% in *JSQual*. Low JSQual similarity scores are most likely the result of removing ads that we deemed unnecessary and unhelpful to the user experience, but that *JSQual* had no way of differentiating from important content.

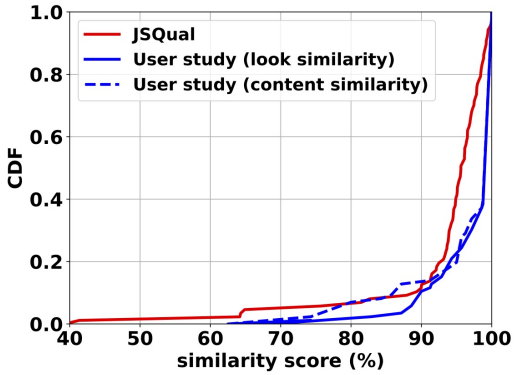

Figure 4: CDF of similarity scores.

4.2 Speed Evaluation

4.2.1 Lighthouse. In this evaluation, we focused on three different metrics from the Lighthouse report:

- First Meaningful Paint (FMP): a measure of when the primary content of a page is visible to the user. This metric measures the time in seconds from when the user initiates the page load to when the page renders the primary above-the-fold content.
- Speed Index: a measure of how quickly page content is visually displayed during page load. This is calculated by taking a video of the screen during page load and measuring its completeness at each time interval.
- Time-to-Interactive (TTI): a measure of how long it takes a page to become fully interactive. This metric measures the time in seconds from when the user initiates the page load to when the page displays useful content that is responsive to user interaction within 50 ms.

Figure 5 shows the CDF and box plot of the above Lighthouse metrics for the 100 pages evaluated using the Samsung high-end device. Although there is little improvement in the FMP, there is some improvement in the speed index as seen in the slight gap between the solid and dotted blue lines in the CDF, and a significant improvement in TTI. The median TTI for the simplified pages has reduced to 40% of that of the original pages.

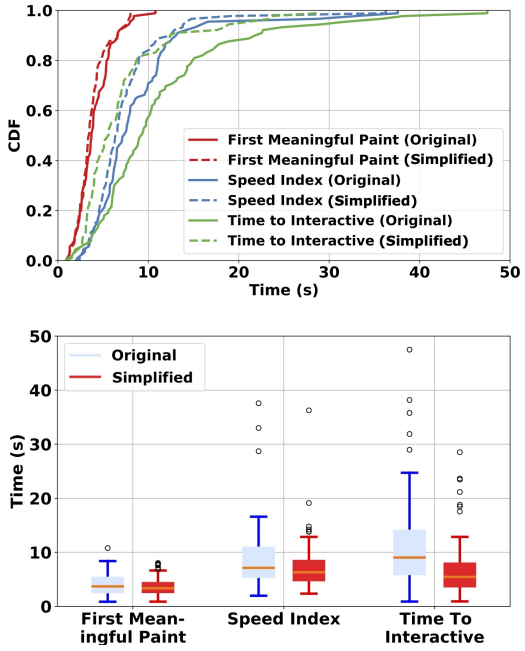

Figure 5: High-end device Lighthouse metrics.

Figure 6 shows the CDF and box plot of the selected Lighthouse metrics for the 100 pages evaluated using the Xiaomi low-end device. Here, the FMP and speed index show clearer improvements for the simplified pages, and the median TTI for the simplified pages has reduced to almost 50% of that of the original pages.

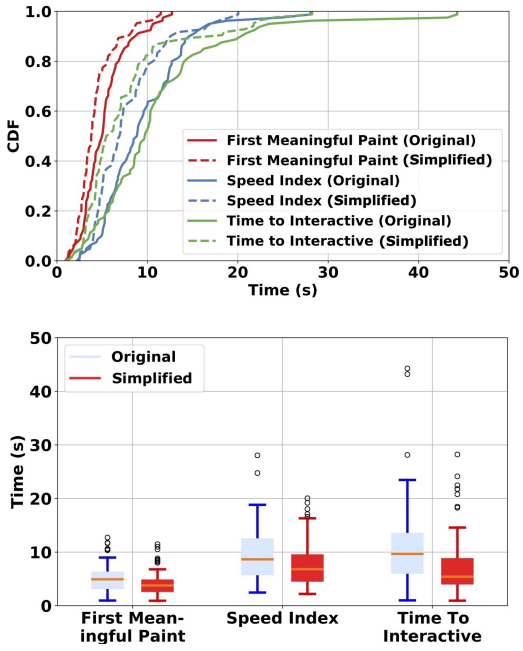

Figure 6: Low-end device Lighthouse metrics.

Figure 7 shows the average overall Lighthouse performance scores of the original and simplified pages for the low-end and high-end devices. The Lighthouse performance score is a weighted average of all of the different Lighthouse metrics, each scored out of 100. These scores are scored relative to real websites based on data from the HTTP Archive [5]. A score of zero represents the lowest possible score, whereas a score of 100 indicates that the page is in the ninety-eighth percentile of websites for that metric. The figure shows that simplifying pages using *JSAnalyze* improves the overall Lighthouse performance of pages by over 35% for the high-end phone, and almost 90% for the low-end phone. In both cases, the average score has been improved from a slow (red) score to a moderate (orange) one.

4.2.2 Webpagetest. In this evaluation, we focused on three different metrics from the webpagetest results:

- DOM Interactive: this marks the point when the browser has finished parsing all of the HTML and the DOM construction is complete.
- Document Complete: this marks the point when the browser onLoad event fires, which generally means that all of the static content of the page has loaded. Any activity beyond the Document complete comes from JavaScript loading some dynamic content.

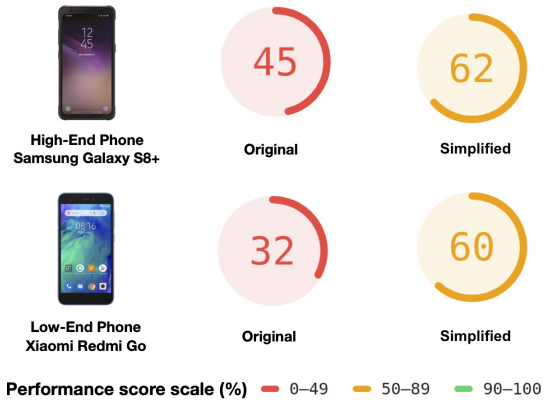

Figure 7: Overall performance metric score.

- Fully Loaded: this marks the point when the network has been idle for about 2 seconds.

Figure 8 shows the CDF of the selected webpagetest metrics for the 100 pages evaluated using the Samsung high-end device, and Figure 9 shows the corresponding metrics using the Xiaomi low-end device. Due to the greater processing power of the high-end device, the PLT metrics are faster for all metrics on the high-end device than the low-end device. However, on both graphs, it is clear that there is a significant reduction in PLT for the simplified page for all metrics. The document complete and fully loaded metrics are reduced by 33% for the low-end device. Using *JSAnalyze* to simplify web pages improves PLT in every scenario.

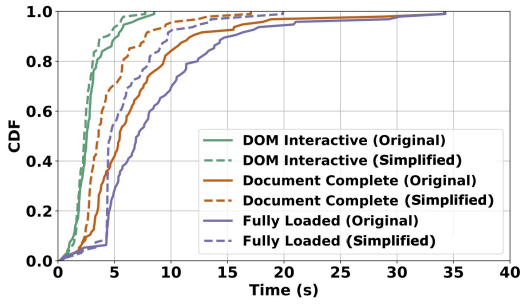

Figure 8: High-end device webpagetest metrics.

5 DISCUSSION

5.1 Large Labeled JavaScript Dataset

One of the potentials of using *JSAnalyze* is the ability to create a large dataset of labeled JavaScript elements. These elements can be crawled from the most popular pages, and

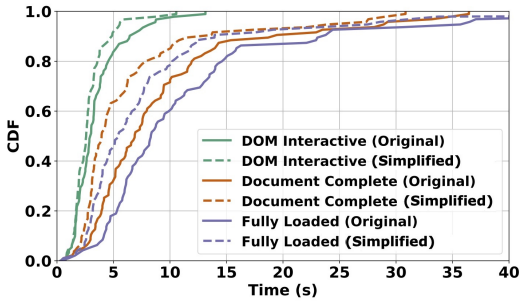

Figure 9: Low-end device webpagetest metrics.

these pages can be distributed via a crowd-sourcing platform. This approach can help in distributing the labeling and generating a large set of labeled data, which can then be used to expand the machine learning classification approach beyond the current dataset. In addition, the tool analysis part can be extended to give the user the chance to modify existing labels, as well as adding their own custom labels. By doing so we can fine-grain the data set labels and enhance the accuracy of the machine learning model. This would enable removing the human role from the loop for a fully automated tool that can simplify web pages based on machine learning.

5.2 Mobile Web Optimization and a Browser Extension

As seen earlier, *JSAnalyze* can be utilized to create lighter versions of web pages, not only to aid users with limited connectivity or low processing conditions, but also to enhance the entire mobile web browsing experience based on user preferences. *JSAnalyze* can help in identifying different classes of JavaScript elements, such that web developers can rapidly generate optimized versions of their pages and make them available for mobile web users. On the other hand, instead of using existing JavaScript-blocking extensions where users have to either explicitly specify domain names/host URLs, or to block a specific known class of JavaScript (such as Ads or trackers), *JSAnalyze* can aid the development of a browser extension where users can block JavaScript elements based on their predicted categories, according to the user preferences.

5.3 Bridging the Digital Divide

Billions of people in developing countries rely on handheld mobile devices to access the web. Users in these countries are not only plagued with poor infrastructure, but they also can not afford to use high-end smartphones. Thus, *JSAnalyze* has a massive potential in simplifying web pages for those users to enhance their browsing experience. One of the main

challenges behind using *JSAnalyze* at a large scale is the fact that it needs to be utilized by web developers around the globe in order to create optimized web pages. However, by using a fully automated approach of *JSAnalyze* without the human in the loop, web pages can be simplified without the intervention of the web developers. *JSAnalyze* in its simplest form, can merely automatically update the list of JavaScript links to be blocked by a browser extension.

The next challenge is to enable a central server to continuously analyze and generate the updated list. One of the ways to address this challenge is to rely on the users in the developed world to aid the simplification and the generation of this list. Users in developed countries can simply install the *JSAnalyze* browser plugin to analyze the pages they regularly visit and share the analysis reports with the central server. The central server can then be utilized by users in developing countries in order to improve their browsing experience.

6 CONCLUSION

JSAnalyze is an analysis environment that aims to optimize JavaScript usage in modern web pages. This analysis can aid web developers and content providers in making optimization decisions on their pages to create lighter versions for the mobile web. In this paper, we demonstrated how the optimized *JSAnalyze* pages outperform the original pages in multiple aspects, while maintaining the visual content and the interactive functionality of these pages. *JSAnalyze* is open-source, and the source code can be found at https://github.com/comnetsAD/JS_Analyzer.

REFERENCES

[1] 2018. webpagetest. https://www.webpagetest.org. Accessed: 2020-03-26.

[2] 2019. The 2019 Web Almanac. https://almanac.httparchive.org/en/2019. Accessed: 2019-12-14.

[3] 2019. Chrome DevTools Protocol Viewer. https://chromedevtools.github.io/devtools-protocol. Accessed: 2019-11-07.

[4] 2019. CITI Program - Collaborative Institutional Training Initiative. www.citiprogram.org. Accessed: 2019-10-10.

[5] 2019. HTTP Archive. https://httparchive.org. Accessed: 2019-09-10.

[6] 2019. JSQual. https://github.com/comnetsAD/autoComparisonTool. Accessed: 2019-11-27.

[7] 2019. Selenium. https://selenium.dev. Accessed: 2019-12-14.

[8] 2019. The State of Mobile Internet Connectivity 2019. https://www.gsma.com/mobilefordevelopment/wp-content/uploads/2019/07/GSMA-State-of-Mobile-Internet-Connectivity-Report-2019.pdf. Accessed: 2020-03-17.

[9] 2019. wxPython. https://wxpython.org. Accessed: 2019-12-14.

[10] 2020. JavaScript Beautifier. https://beautifier.io. Accessed: 2020-05-14.

[11] 2020. Lighthouse. https://developers.google.com/web/tools/lighthouse. Accessed: 2020-03-26.

[12] 2020. mitmproxy. https://mitmproxy.org. Accessed: 2020-05-14.

[13] 2020. The top 500 sites on the web. https://www.alexa.com/topsites. Accessed: 2020-01-03.

[14] Talal Ahmad, Yasir Zaki, Thomas Pötsh, Jay Chen, Arjuna Sathiaseelan, and Lakshminarayanan Subramanian. 2019. GAIUS: A New Mobile Content Creation and Diffusion Ecosystem for Emerging Regions. In *Proceedings of the Tenth International Conference on Information and Communication Technologies and Development* (Ahmedabad, India) *(ICTD '19)*. ACM, New York, NY, USA, Article 34, 5 pages. https://doi.org/10.1145/3287098.3287130

[15] Kayce Basques. 2020. Inspect Network Activity in Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools/network/#load. Accessed: 2020-05-14, Updated: 2020-05-07.

[16] Google Developers. 2019. Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools. Accessed: 2020-05-01.

[17] Maximiliano Firtman. 2016. *High Performance Mobile Web: Best Practices for Optimizing Mobile Web Apps*. "O'Reilly Media, Inc.".

[18] Google. 2019. AMP is a web component framework to easily create user-first web experiences - amp.dev. https://amp.dev. Accessed: 2019-05-05.

[19] Patrick Hulce. 2019. Third-Party Web. https://www.thirdpartyweb.today/about. Accessed: 2019-11-27.

[20] Byungjin Jun, Fabián E Bustamante, Sung Yoon Whang, and Zachary S Bischof. 2019. AMP up your Mobile Web Experience: Characterizing the Impact of Google's Accelerated Mobile Project. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–14.

[21] Mozilla and individual contributors. 2005. Firefox Developer Tools. https://developer.mozilla.org/en-US/docs/Tools. Accessed: 2020-05-01.

[22] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/netravali

[23] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. 2018. Vesper: Measuring Time-to-Interactivity for Web Pages. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 217–231. https://www.usenix.org/conference/nsdi18/presentation/netravali-vesper

[24] Addy Osmani. 2017. The Cost Of JavaScript - Dev Channel - Medium. https://medium.com/dev-channel/the-cost-of-javascript-84009f51e99e. Accessed: 2020-01-05.

[25] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 473–485. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/wang_xiao

[26] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up Web Page Loads with Shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 109–122. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/wang

[27] Yasir Zaki, Jay Chen, Thomas Pötsch, Talal Ahmad, and Lakshminarayanan Subramanian. 2014. Dissecting Web Latency in Ghana. In *Proceedings of the 2014 Conference on Internet Measurement Conference*. 241–248.