JSCleaner: Firefox Plugin

Manesha Ramesh
Computer Science, NYUAD
mr4684@nyu.edu

Advised by: Yasir Zaki, Moumena Chaqfeh

ABSTRACT

JavaScript constitutes a disproportionate percentage of the web while also being a dominant factor in slow page loads and poor browsing experience. However, JSModel proposes a machine learning based approach to classify JavaScript code into 14 categories; based on these classes, JavaScript can be deemed essential or non-essential to a web page load. This capstone realizes a use case scenario for this classification model in the form of a client-oriented Firefox plugin. Results show that across multiple implementations of the plugin (Single Request Implementation and Batch Request Implementation), a 31.6% and 66% reduction in Full Page Load Time can be achieved respectively by blocking JS code of classes Advertising, Analytics, Social and first party non-critical scripts.

KEYWORDS

computer-networks, javascript-simplification, user-interface, page-load-time

Reference Format:
Manesha Ramesh. 2020. JSCleaner: Firefox Plugin. In *NYUAD Capstone Seminar Reports, Spring 2020, Abu Dhabi, UAE.* 9 pages.

1 INTRODUCTION

A study by Google reveals that web pages that do not load within 5 seconds lose about 90% of their visitors [3]. Nevertheless, according to http archive, the number of requests per page has increased by 29% from 2010 to 2019. Moreover, the average download size per request has increased by 320% in the same time duration [3]. With increasing size and complexity of web pages, the web is experiencing what is colloquially known as "web bloat" [12], resulting in a very "inefficient use of network traffic and computation resources" [12]. In regions with low-connectivity, the cost of loading heavy web resources manifests itself in longer page loads and poor browsing experience.

Many steps are involved in the loading of a web page: making network requests, downloading web objects, issuing DNS requests, processing JavaScript, generating the DOM, CSS and render trees. Two primary factors that influence the Page Load Times (PLT) are the costs of making network requests and the processing time of the resources at the end-user's browser. JavaScript is proven to be the most expensive resource for web browsers to process [10], hence contributing the most to Page Load Time.

JavaScript operates on the main thread of the browser and therefore long execution times can block the thread and delay the completion of the web page load [13]. Moreover, several requests are made to the network to fetch these JavaScript files as external resources. According to the Web Almanac 2019 report, web pages are sending a median of 19 JavaScript requests [5]. Taking www.bbc.com as an example for web page testing, JavaScript takes up 36.6% of the number of requests and 53.8% of the download size [8].

According to Stack Overflow's annual developer Survey, over 67% of respondents have selected JavaScript to be their favorite programming language making it the most popular programming language for 7 years in a row [2] and constituting 95.2% of today's web, despite being a costly resource. The motivation behind using JavaScript is to enhance the interactivity of web pages. To do so, developers are reliant on bulky JS libraries and frameworks to speed up the development process. JQuery, ReactJS and AngularJS are among the three most popular web frameworks used in web development; JQuery is the most popular JS library constituting 85.03% of web pages accessed by desktops [2]. In most cases, developers insert the entire library despite only needing a single function from it. Smaller JS bundles on a web page improves performance in terms of not only PLT but also in bandwidth and energy consumption (since the transfer size is low) [5]. Such libraries are third party scripts which constitute a significant portion of JavaScript in the web. A median of 89% more third party code is used than the first party code, being one of the biggest contributors to the web bloat problem.

2 RELATED WORK

In view of the problems that come with the pervasiveness of JavaScript in the web, the research community has been involved in producing strategies to simplify the web on a page level and decrease PLTs. Some utilize server-side technology to decrease PLT, some have focused on simplifying web pages to be "readable", while others analyze dependencies on a very granular level [6]. SpeedReader upgrades the implementation of reader modes (used to decrease page clutter and extract the core content) by operating on a web pages before page rendering as opposed to after the webpage has been loaded. This decreases the number of network requests and saves the processing power from rendering many unnecessary media files. However, its benefits are bound by the number of "readable" pages it can process, which is only 22% of the sample in the SpeedReader work [6].

On the other hand, Shandian relies on server-side technology to reduce Page Load Times [14]. It simplifies client-side loading by splitting the process between a proxy server and the end user. The pre-processing of the web page in the proxy server (with more computation power), reduces computation costs on the end user's device [14]. Although, Shandian's restructured architecture of the loading process reduces the PLT, it does not attempt to simplify the content of the webpages. Solutions for improving caching technology (edge caching) as done so by xCache is another approach to the issue at hand [11]. It also relies on a proxy server referred to as a "Cloud Controller" that optimizes web content, maximizes edge cache hit rates while also minimizing the bandwidth usage for data updates [11].

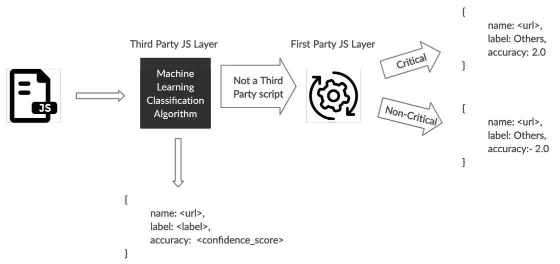
Extensive research has been conducted regarding web object dependencies on a granular level which have encouraged the making of algorithms that can conduct critical path analyses for web pages. Polaris is an implementation of such dependency graphs. It harnesses server-side technology (Scout) to produce dependency graphs for web pages [9]. Then, a client-side scheduler uses this dependency graph while adjusting to network conditions to produce optimized dynamic critical paths [9].

Web Simplification has been central to many kinds of research in academia; because of its commercial benefits, industry has attempted to intercept the challenge as well. Facebook Lite is an example of an initiative by Facebook which simplifies the web app for users in poor network conditions [?]. However, it is only designed for Facebook applications. Google Amp is another project that is undertaken by Google that enforces the use of a framework that curbs the use of JavaScript by only allowing the use of Google JavaScript [1]. However this can be restrictive as there are certain front-end elements that cannot be implemented within the boundaries of Google Amp API. Also, this initiative only focuses on improving the Page Load Times of upcoming applications and does little to simplify the clutter of the existing web.

The JSCleaner Project [4] is a novel ongoing approach to web and JS simplification in which the content is examined and classified into different buckets that are expected to describe the entire JavaScript content of the world wide web [7]. This paper examines a client-oriented use case scenario of this technology in the form of a Firefox plugin that harnesses this technology of JS classification and allows the user to achieve desired/customizable reduction of JavaScript content. It implements the JSModel [7]which utilizes a combination of a Machine Learning Model and rule-based model to classify JavaScript.

3 JSMODEL

The JSModel is an engine that assigns a class to each JS resource loaded by a given web page [7]. This classification of JavaScript into 14 categories caters to the customizable simplification process in the plugin in which a lighter version of the web page can be generated according to the user's preferences. The JSModel follows two layers of classification: Third-party JS Model and First-Party JS Model. Figure 1 illustrates the full architecture.



```
{
    name: <url>,
    label: Others,
    accuracy: 2.0
}
```

```
{
    name: <url>,
    label: Others,
    accuracy:- 2.0
}
```

```
{
    name: <url>,
    label: <label>,
    accuracy:  <confidence_score>
}
```

Figure 1: The JS Model

Third-Party JS Model: A Third party JavaScript is a library that can be hosted outside the server but can also live in the web server as an imported external library. The first layer of the classification model detects third party scripts using an ML model that uses a Random Forest classification algorithm to classify JavaScript elements into 12 categories as proposed by Web Almanac:

(1) Advertising: Refers to scripts that embed ads in a web page
(2) Analytics: Scripts that track or record user interactions with web page

(3) Social: Refers to JS elements that enable social features in a web page
(4) Video: Scripts that embed video players and manage streaming functionality
(5) Utilities: Web developer utilities, site monitoring utilities or fraud detectors
(6) Hosting: JS elements brought by web hosting platforms (such as WordPress or Wix)
(7) Marketing: Scripts brought by marketing tools (sales and email marketing)
(8) Customer Success: Libraries brought from customer support/marketing providers that offer chat and contact solutions
(9) Content: Libraries brought by content providers or publishing-specific tracking
(10) CDN: Publicly hosted open source libraries (JQuery), served over public and private CDNs
(11) Tag Management: JS scripts that load other JS scripts
(12) Other: Refers to scripts that don't belong to any of the other above categories and are most likely first party JavaScript files.

The ML model analyzes the content of the scripts in terms of various JS API calls and generates a label. Although, this model can technically label any type of JavaScript, it is trained using Third Party JS Scripts. Therefore, it recognizes third party scripts with a higher confidence score. Any script that is labelled with a score that is lower than a threshold of 60% is considered a first party script and passed to the second layer of the model.

First Party JS Model: This is the second layer of classification provided by the JSModel. Unlike the ML classifier in the previous layer, this layer uses a rule-based model that classifies JS elements that are found in the "Other" bucket of the Third-Party JS model [7].

The "Other" class consists of scripts that were not accurately classified by the previous layer and are most likely first Party JS elements. This layer further classifies such elements into critical and non-critical scripts. The JSCleaner [4] identifies three basic types of features: (1) Event Features (2) Write features (3) Read features [7]. Based on these features the following two classes can be assumed:

(1) Critical (Class 1): where event features and writing features can be extracted and hence the JS element should be preserved
(2) Non-critical (Class 2): where neither events or writing features can be extracted from the JS code.

The user plugin implements this JSModel with the two layers of classification using a combination of server-side and client-side technology. This is described in detail in the following section.

4 METHODOLOGY

4.1 Plugin Implementation

The JSCleaner user plugin labels JavaScript files, stores it locally and blocks requests based on the user's desired settings. The labelling of the scripts occur in the server and the blocking of unnecessary JavaScript takes place at the browser. The Apache web server receives the unlabelled url(s) in the form of a GET request from the plugin, uses the JSModel to label them and then returns a JSON object. The browser receives the JSON object, updates its local database and blocks requests accordingly upon page loads. Figure 2 illustrates the architecture of the system.

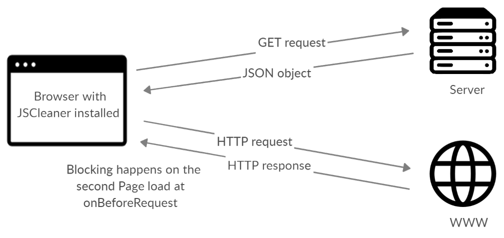

GET request
JSON object
Server
Browser with JSCleaner installed
HTTP request
HTTP response
Blocking happens on the second Page load at onBeforeRequest
WWW

Figure 2: The architecture of the JSCleaner Plugin

Web Server: The server that is used in this user plugin is an Apache web server that implements the JSModel. When it receives a url or a list of urls from the browser, it checks its local database for the script. If it is not there, it makes a request to the web to retrieve the content of the script. Information about the combination of different JavaScript features in the script is extracted and vectorized. This generated vector is injected into the ML model which produces a label and a confidence score. If this confidence score falls below the threshold of 60%, the script is placed in the "Other" bucket and passed onto the second layer of the classification. The second layer further classifies the scripts in the "Other" bucket into critical and non-critical JavaScript using the rule-based model. After the labels are generated, the database is updated and the JSON object containing the name(s), the label(s) and the confidence score(s) is sent back to the browser.

Plugin: Before a request for a certain JS file is sent to the web, the plugin checks if the script is labelled. If it is, it blocks the script depending on the user's settings. If it is not labelled, a GET request is sent to the Apache web server with the url(s).

Blocking: Harnessing the webRequest API, blocking happens when the *onBeforeRequest* event is fired. The plugin is only capable of blocking already labelled scripts. Therefore, blocking of JS files does not happen in the first page load; it happens in the Second Page Load (SPL). The plugin implements two kinds of blocking:

(1) Cross-web page blocking
(2) Page-specific blocking

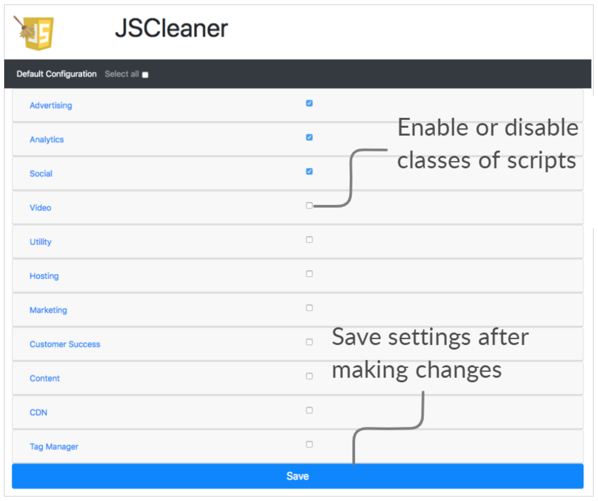
The user can block scripts based on classes on the settings page of the plugin. However, if the user wishes to relax the blocking of certain scripts, he may use the interface to enable or disable specific scripts on a web page.

Storage: There are two levels of storage in this system; one is managed by the server and the other is managed by the plugin at the client's end. As the server receives GET requests from many browsers for JavaScript labelling, it is generating a shared database of labelled scripts that can be shared across all users of the plugin. As the database grows in size, the response time of the web server is expected to be faster as it will no longer need to rely on the ML model for every request. The local storage mechanism at the client's end stores scripts names and their labels requested by the user's most frequented websites. The local storage of a browser is limited to 5MB. To increase the storage capacity, the plugin uses an API for client-side storage called IndexedDB - a noSQL database - that can hold and manage data exceeding 5MB (limit is 50MB). The database contains two tables. One table stores the url to scripts, their labels and the confidence score. The other table stores the url of web pages that have page-specific settings.

Interface: The JSCleaner plugin has provided the user an interface to achieve a desired and customizable reduction in page requests and page size. In figure 3, the user may select a combination of classes that he/wishes to disable across all websites loaded by the browser.

It may be the case that the user may want to enable a certain script for a specific website that may conflict with the all-browser setting,he/she may do so by managing the scripts of a specific web page by navigating to it and enabling or disabling specific scripts. This feature is in place to account for websites that may break as a result of blocking classes of scripts (Figure 4). For example www.instagram.com does not load any content when a certain script of type Analytics is blocked. This plugin provides an interface and mechanism to create exceptions for such sites.

The plugin has been implemented in two different ways. In Figure 2, it can be seen that for every request for an unlabelled JS resource, the plugin sends a GET request to the proxy. This is problematic because first page loads of various sites can flood the network with requests. As an attempt to reduce the number of requests, the plugin has been implemented such that requests are sent to the proxy in batches. From this point onward, the Batch Request Implementation and Single Request Implementation will be referred to as BRI ans SRI respectively. This paper will


JSCleaner

Default Configuration Select all

Advertising	✓
Analytics	✓
Social	✓
Video	☐
Utility	☐
Hosting	☐
Marketing	☐
Customer Success	☐
Content	☐
CDN	☐
Tag Manager	☐

Enable or disable classes of scripts
Save settings after making changes
Save

Figure 3: Interface to enable or disable classes of JavaScript across all sites

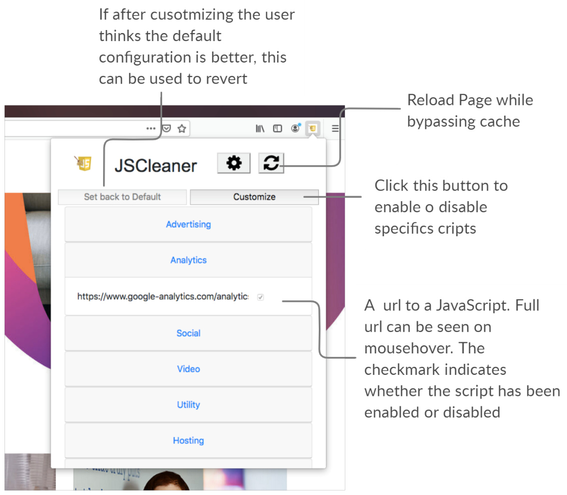

If after cusotmizing the user thinks the default configuration is better, this can be used to revert
Reload Page while bypassing cache
JSCleaner
Set back to Default
Customize
Click this button to enable o disable specifics cripts
Advertising
Analytics
https://www.google-analytics.com/analytic
A url to a JavaScript. Full url can be seen on mousehover. The checkmark indicates whether the script has been enabled or disabled
Social
Video
Utility
Hosting

Figure 4: Interface to enable or disable scripts for a specific site

evaluate and compare both implementations in terms of: Page Load Time, JS requests and Transfer size.

BRI: In this implementation, the plugin collects urls of requested JS resources and sends them to the Apache web server in batches. The plugin does so by maintaining an array of scripts which upon reaching a threshold of size 5, sends the following GET request to the server:

<IP_address_of_the_server:port_number>?url=<encoded_url>,<encoded_url>,<encoded_url>,<encoded_url>,<encoded_url>

In case the threshold is not met, a timeout mechanism sends the urls and clears out the array regularly.

SRI: In the first Page load of a given web page, for every request for a JS resource, a GET request containing an encoded url as a parameter is sent to the web server as follows:

<IP_address_of_the_server:port_number>?url=<encoded_url>

The web server returns a JSON object with the label and confidence score for that script.

4.2 Challenges in the implementation:

When factoring in the limitless possibilities in the architecture of various websites, the plugin does fail under some circumstances. After the implementation, some manual tests were performed on a list of 10 websites and the following challenges were encountered:

(1) *Scripts are requested from different domains on each page load:* This plugin distinguishes one script from another based on its url. It does not have the ability to check for similarity across scripts. For instance, when loading www.sina.com/cn, two JS elements (toutiaobao.js and sinaere.js) are requested from a different domain on every page load:
First Page Load:
-> https://d2.sina.com.cn/litong/zhitou/sinaads/test/ e -recommendation/release/sinaere.js,
-> https://d5.sina.com.cn/litong/zhitou/sinaads/demo /jiliang/toutiaobao/toutiaobao.js
Second Page Load:
-> https://d9.sina.com.cn/litong/zhitou/sinaads/test/ e -recommendation/release/sinaere.js
-> https://d6.sina.com.cn/litong/zhitou/sinaads/demo /jiliang/toutiaobao/toutiaobao.js
The plugin treats requests for the same script from two different domains separately. So, the same script will be treated as a first encounter and labelled separately each time without getting to the blocking. Therefore, even after the second, third or fourth page load, sinaere.js and toutiaobao.js are not guaranteed to be blocked.

(2) *JavaScript file recognition:* Before a request is sent, the plugin only has access to information provided by the request headers. The plugin recognizes JavaScript files if the request headers categorize the element as a resource of type ‘script or has “js”. Some web pages request JavaScript files that are categorized as a resource of a different type. For instance, https://outlook.live.com /owa/ requests some JavaScript files (e boot.worldwide.0 .mouse.js, boot.worldwide.1.mouse.js,boot.worldwide.2 .mouse.js, boot.worldwide.3.mouse.js) that are of type stylesheets. The plugin is able to recognize them as scripts because the name of the scripts include a “.js” file extension. However, this is not the most reliable way of recognizing scripts because some urls to JS resources may not have that identifier. For example, when loading *http://sohu.com/*, a request is made to *https://txt.go.sohu.com/ip/soip* which is a JS file but does not have “.js” identifier. Fortunately, the request header marks it as a resource of type script and as a result the plugin manages to label the script returned by the url. However, if a request that is neither formally of type “script” and does not have a “*.js*” identifier may be ignored by the plugin.

(3) *Server Response Time:* Sometimes the response time of the server might be slow (especially if the urls are sent in batches) and so some scripts may not be labelled in time for the reload of the web page.

5 EVALUATION AND RESULTS

The goal of this plugin is to reduce the PLT, by curbing a web page of unnecessary JavaScript. In order to evaluate the plugin's performance, the study takes Alexa's top 50 websites and records metrics pertaining to load time, number of requests and transfer size. In this experiment, the following should ne noted:

(1) The most recommended setting for users is to disable scripts of class Analytics, Advertising, and Social. This is the setting used for the tests.
(2) To best show the performance of the plugin, the browser cache has been disabled.
(3) The performance of the plugin will be compared to the base case - page load without the plugin.

Page Load Time: To evaluate the performance of the plugin based on Page Load Time, this study uses three different metrics:

(1) Time to Interactive - Time it takes for a web page to be interactive
(2) Time to DOMContentLoaded - Time to the point where the DOM tree has been constructed and no stylesheets are blocking JavaScript execution.
(3) Full Page Load Time - Time to the completion of the web page load

Figure 5 illustrates an almost 31.6% reduction in the average Time to DOMContentLoaded and Full Page Load Time in the SPL. However, reduction in Time to Interactive is < 1%. The First Page Load has a higher average load time than the base case because of the computational overhead cost that comes with delaying the requests that are being sent from the browser to check if the script is unlabelled or labelled. Figure 6 plots the cumulative distribution of the Full Page Load for SRI. The median page load time for SPL and the

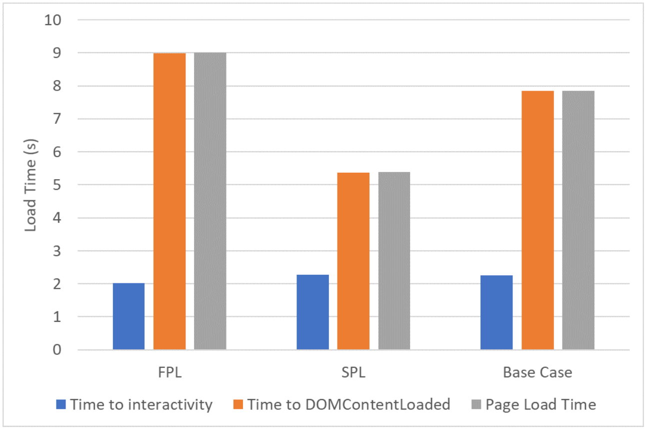


Figure 5: Average Page Load Times in the First Page Load (FPL) and Second Page Load(SPL) for the Single Requests Implementation and the base case

Base Case is 3.787s and 2.117s respectively indicating a 44% reduction in load time.

Number of Requests and Transfer size: The decreased Page Load Time is justified by figure 7 and 8 which describes the reduction in page content (JS requests and page size). The number of requests to JS resources and the transfer size in the FPL are similar to the base case because blocking does not occur then. Nevertheless, in the SPL, with the blocking of non-essential JS scripts, the median number of JS requests is 2 which is a 50% reduction from the base case. The transfer size too decreases with reduced JS requests. In the base case, page size can go over 6MB while in the SPL of the SRI the page size does not exceed 4.21 MB. Moreover, with the plugin 50% of the pages can have a size of no more then 0.38MB which is a 26% reduction from the base case (0.522MB). hence, blocking non-essential JavaScript decreases the page size and JS requests along with page load time.

5.1 Batch Requests

The underlying issue with the Single Request Implementation is the doubling of requests. For each request to a JS resource another request is sent to the Apache web server. Therefore, a second implementation strategy for the plugin was designed in which one GET request to the web server is sent for every five JS scripts. The results of this evaluation are illustrated in the figures 9, 10, 11 and 12.

Page Load Time: It is clear that BRI outperforms SRI in terms of PLT. On average, the second page load of BRI witnesses a 66% reduction in Time to DOMcontentLoaded and Full Page Load Time compared to the 31.5% reduction in SRI (Figure 9). Unlike in the former implementation, there is a 36% decrease in the average Time to interactivity as well. 50% of pages have a page load time of up to 1.633s which indicates a

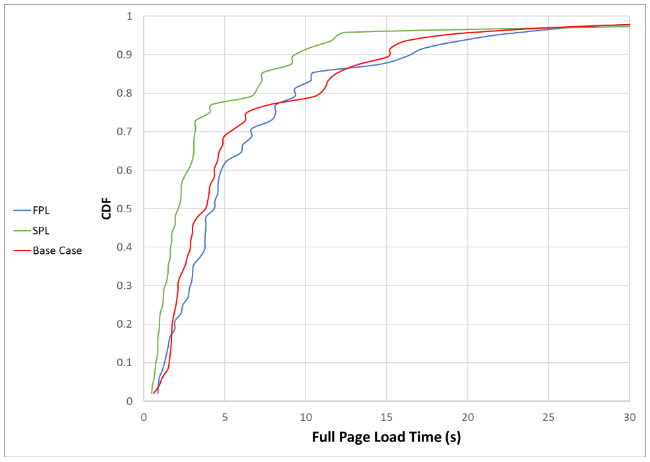


Figure 6: Cumulative Distribution of Full Page Time the of SPL, FPL ans the base case

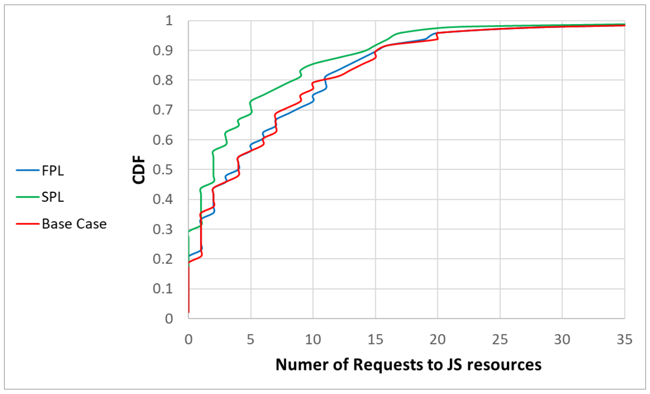


Figure 7: Cumulative Distribution of the number of requests to JS resources in the First Page Load (FPL), Second Page Load (SPL) and base case

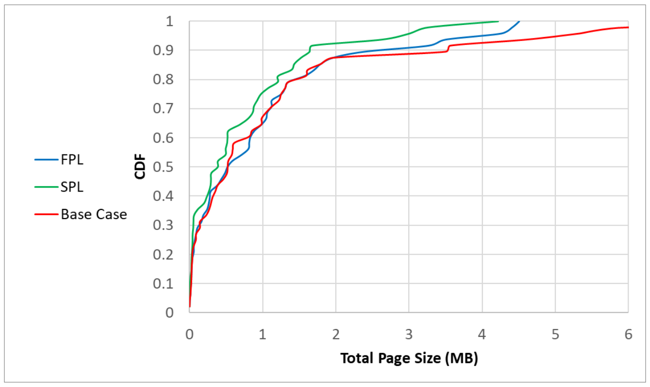


Figure 8: Cumulative Distribution of the total transfer size in the First Page Load (FPL), Second Page Load (SPL) and base case

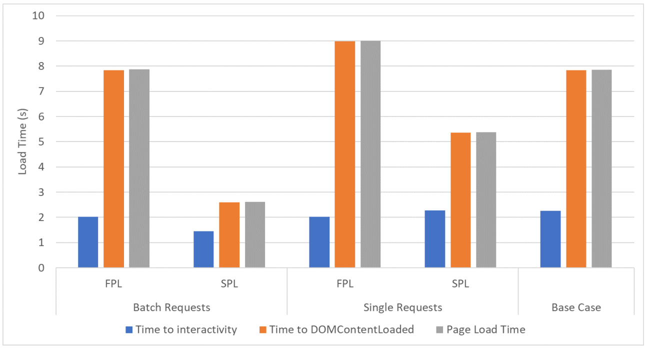


Figure 9: Averages of Time to Interactive, Time to DOM ConentLoaded ans full page load time for BRI, SRI and base case

reduction of 56% from the base case as opposed to the 44% in the SRI. Notice that the maximum PLT is 10.171s whereas SRI and the base case exceed 30 seconds. This improvement in performance can be due to the reduced network requests; this avoids network flooding and improves server response time by not eating up the network bandwidth. Another viable reason is that in SRI the process of checking whether a script is labelled occurs when the *onBeforeSendHeaders* event is fired in the browser which is followed by the *onBeforeRequest* event. Blocking can only occur in the latter stage. In BRI, GET requests to the web server for unlabelled scripts are sent when the former event is fired and the labelled scripts are recognized and blocked in the latter stage. This process is non-blocking and occurs synchronously. However, in the SRI implementation, both the GET requests for labelling and the blocking happen on the *onBeforeRequest* stage and thus both operations occur asynchronously resulting in a longer delay for each request to a JS resource. In other words, the BRI takes advantage of browser's synchronous nature while SRI does not and incurs lower computational costs for the former.

Number of requests and Transfer Size: Surprisingly there is a difference in JS requests and page size between BRI and SRI in the SPL. Although, the median number of requests for BRI is equal to the SRI,the average for BRI, 4.65 requests, is lower than that of SRI, 5.38 requests. The difference can be visually observed in Figure 12. The median transfer size for BRI is 0.30MB which is 22% lower than that of SRI and 43% lower than the base case. The difference may be because of BRI's better performance (in terms of lower computational overhead) which allows for more labelling to occur in time for the second page load (detailed explanation under *Page Load Time*) and hence more requests are blocked.

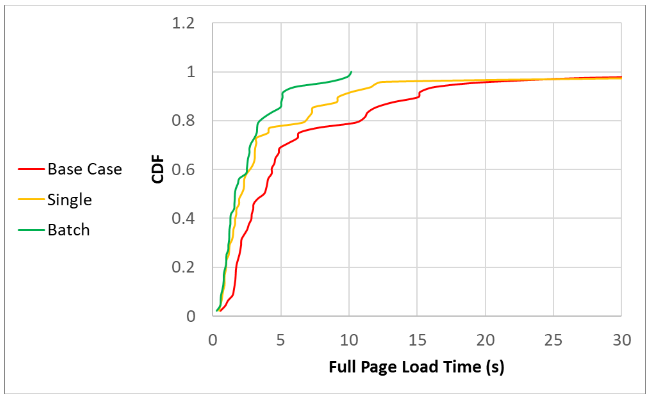


Figure 10: Cumulative Distribution of full page load time in the Second Page Load for BRI, SRI and Base Case

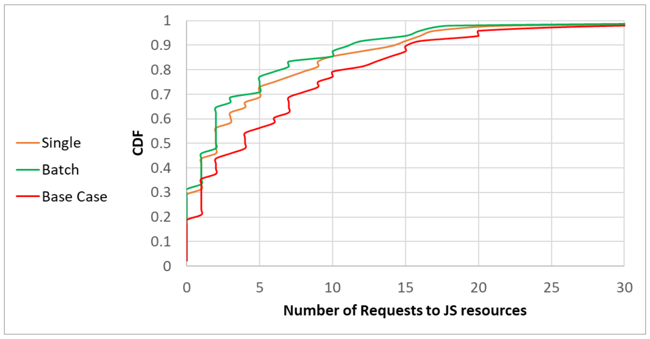


Figure 11: Cumulative Distribution of number of JS requests in the second page load for BRI, SRI and Base Case

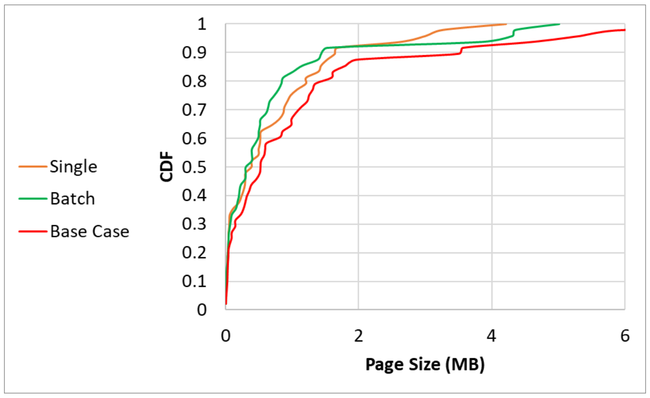


Figure 12: Cumulative Distribution of page size in the second page load for BRI, SRI and Base Case

5.2 Distribution of JavaScript Classes

This section is a commentary on the labelling process of the JSModel. From the list of Alexa's top 50 websites, a total of 2416 labelled scripts were gathered in the server's database. Figure 13 and 14 portray the distribution of classes in this data set.

About 80% of scripts are labelled as "Other" and are passed to the First Party JS Model. That means only about 20% of the scripts are recognized as a class of Third Party Scripts with a confidence of over 0.60. The performance of the plugin depends heavily on the ability of the ML model to classify JavaScript with a high confidence. Perhaps, the plugin would be able to perform better with a wider distribution of labels and a larger cluster of recognized Third Party JS scripts. Moreover, the following figure shows the distribution of classes in the second layer of the JSModel. According to our model, 45% of scripts are non-critical and hence blocked when requested by the user plugin. This means, scripts that don't perform any significant operation on the DOM tree of a web page take up a massive percentage of first party JavaScript. This demonstrates the need to curb the web of JavaScript that reduce the bloat and as a result decrease page load time.

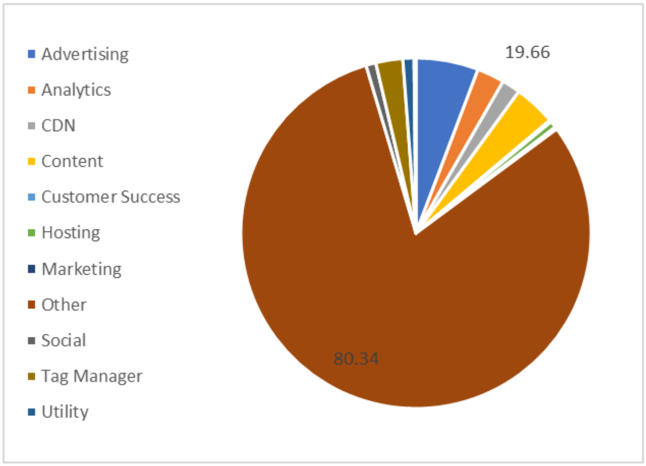


Figure 13: Distribution of Labels in the Third Party JS Model

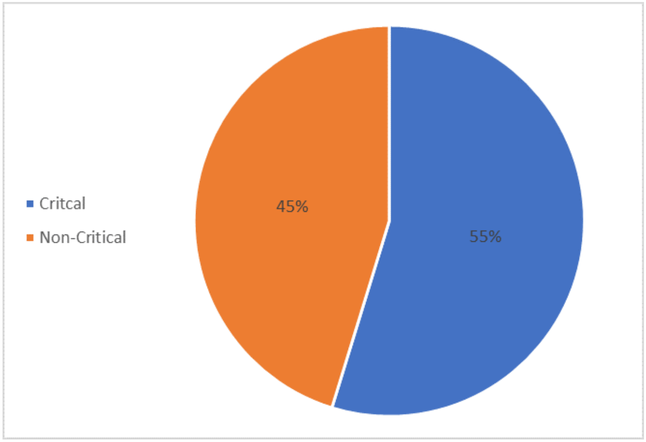


Figure 14: Distribution of Labels in the First Party JS Model

6 CONCLUSION

The JSLCleaner user plugin demonstrates the potential of the JSCleaner project as a client-oriented use case scenario. This plugin provides an interface for normal users to this classification-based JS simplification algorithm, encouraging them to make more well-informed decisions regarding the blocking of scripts and giving them the agency to decide which scripts are necessary and which are not. Moreover, a large-scale deployment of this plugin would also generate a comprehensive database of labelled scripts. While users in developed regions populate the database with their use of the web, users in developing regions can benefit from its growing classification and hence web simplification capacity.

As of now the plugin labels scripts in the first page load and blocks scripts in the second page load. A possible implementation in which the scripts can be blocked in the first page load deserves some attention. The labelling component part of this project classifies based on content without checking for similarity with other scripts. Hence, this results in redundant analysis and labelling. The efficiency of the labelling can be upgraded if another layer is added to the JSModel architecture that allows for similarity checks.

Because the tests were performed semi-manually, this study uses a small size of 50 websites to analyze the effect of the plugin's blocking on Page Load Time, number of JS requests and Page size. For more precise results and smoother cumulative distributions, future tests should be performed on a larger data set.

The evaluation surfaced the difference in performance between the BRI and SRI because of the computational overhead that comes with asynchronous computing. Due to time limitations, this capstone was unable to change the SRI to a more synchronous approach and evaluate it. However, any future work should re-evaluate the performance difference between BRI ans SRI while both operate asynchronously

Although the quantitative analysis of this user plugin's performance proves a reduction in Page Load Time and JS transfer size, a qualitative analysis of this plugin is also required to check if the web pages before and after blocking are similar and functional in terms of interactivity . Due to the limitations of the corona crisis outbreak, this was not possible. However, any future efforts to improve this plugin must include a qualitative evaluation.

7 ACKNOWLEDGEMENTS

A heartful gratitude is due to both Yasir Zaki and Moumena Chaqfeh for their counsel and guidance. I would also like to thank my friends Shivani Mishra, Armaghan Khan, Febin Thapa Magar and Manson Tung for their moral support.

REFERENCES

[1] [n.d.]. AMP - a web component framework to easily create user-first web experiences. https://amp.dev/

[2] [n.d.]. Stack Overflow Developer Survey 2019. https://insights.stackoverflow.com/survey/2019

[3] 2017. Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed. https://www.thinkwithgoogle.com/intl/en-aunz/advertising-channels/mobile/au-mobile-page-speed-new-industry-benchmarks/

[4] Moumena Chaqfeh, Yasir Zaki, Jacinta Hu, and Lakshmi Subramanian. 2020. JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup. In *Proceedings of The Web Conference 2020*. 763–773.

[5] Houssein Djirdeh. 2019. JavaScript: 2019: The Web Almanac by HTTP Archive. https://almanac.httparchive.org/en/2019/javascript

[6] Mohammad Ghasemisharif, Peter Snyder, Andrius Aucinas, and Benjamin Livshits. 2019. SpeedReader: Reader Mode Made Fast and Private. In *The World Wide Web Conference*. 526–537.

[7] Muhammad Haseeb, Moumena Chaqfeh, Manesha Ramesh, Vladyslav Cherevkov, Gabriel Garcia Leyva, Fareed Zaffar, and Yasir Zaki. 2020. JSModel: Analyzing JavaScript in Today's Web Using Machine Learning. In *ACM IMC 2020*. Under Submission.

[8] Patrick Meenan. [n.d.]. WebPageTest. https://www.webpagetest.org/

[9] 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)*.

[10] Addy Osmani. 2019. The Cost Of JavaScript In 2018. https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4

[11] Ali Raza, Yasir Zaki, Thomas Pötsch, Jay Chen, and Lakshmi Subramanian. 2017. xcache: Rethinking edge caching for developing regions. In *Proceedings of the Ninth International Conference on Information and Communication Technologies and Development*. 1–11.

[12] M. Selakovic and M. Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 61–72.

[13] 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)*. 473–485.

[14] 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)*. 109–122.