QLUE: A Computer Vision Tool for Uniform Qualitative Evaluation of Web Pages

Waleed Hashmi
Computer Science, NYUAD
waleedhashmi@nyu.edu

Advised by: Dr. Yasir Zaki, Dr. Moumena Chaqfeh

Reference Format:
Waleed Hashmi. 2020. QLUE: A Computer Vision Tool for Uniform Qualitative Evaluation of Web Pages. In *NYUAD Capstone Seminar Reports, Spring 2020, Abu Dhabi, UAE.* 19 pages.

EXTENDED ABSTRACT

The increasing complexity of web pages has attracted a number of solutions to offer simpler versions of these pages. These solutions have been quantitatively evaluated to show significant speed-ups in page load times and/or considerable savings in bandwidth and memory consumption. However, what these solutions fail to evaluate is the qualitative impact on their generated pages, i.e., how these solutions might lead to missing content and broken functionality—apart from only a few who relied on small-scaled user studies. Additionally, due to the lack of a unified qualitative metric, it is nearly impossible to fairly compare results obtained from different user studies campaigns, unless recruiting the exact same human evaluators. In this paper, we demonstrate the lack of qualitative evaluation metrics, and propose *QLUE* (QuaLitative Uniform Evaluation), a tool that automates the qualitative evaluation of web pages using computer vision. Our results show that *QLUE* computes comparable content and functional similarity scores to those provided by human evaluators. Specifically, for 90% of 100 pages, the human evaluators gave content similarity scores between 90% and 100%, while QLUE shows the same range of similarity scores for more than 75% of the pages. In addition, QLUE effectively evaluates the functionality of web pages, whereas this is proven to be a challenging task for humans given the functional dependencies in modern web pages. QLUE's time complexity evaluation results show that the tool is capable of generating the scores in a matter of few minutes. Finally, the usability and benefits of QLUE is assessed by an independent expert who co-authored one of the most recent web complexity solutions.

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 Seminar, Spring 2020, Abu Dhabi, UAE

1 INTRODUCTION

In order to tackle the web complexity challenge, many of the above solutions need to strike a balance between accelerating the pages and compromising some aspects of the pages either in terms of their appearance or functionality. This of course requires rigorous evaluation to highlight the impact of these solutions on both sides of the spectrum (i.e., speedups gains, vs. loss of content/functionality).

Conventionally, there are three high-level metrics that are considered as key performance indicators for these tools: (1) user experience timings metrics such as SpeedIndex and Time-To-Interactivity, (2) resource utilization measurements such as bandwidth, CPU, memory, energy consumption, and (3) overall page quality in terms of content completeness and retained functionality. The evaluations of the above solutions focus solely on either the first metric, or the combination of the first and the second, while ignoring the solution impact on the overall page quality. In few work, such as the one found in [12, 25, 30], the authors evaluated the impact on some aspects of the page quality using simple and small-scale user studies. Generally, to evaluate the quality of pages, authors rely on conducting user studies. Although, we agree that nothing can replace humans perception/intuition in evaluating the qualitative aspects of web pages, there are many constraints that hinders conducting large-scale user studies. These constraints can be of financial nature—due to the high associated cost, while others are of logistical nature—for example the shutdown that resulted from the COVID-19 pandemic.

To overcome the above challenges and to speed-up the development cycles of web acceleration solutions, we propose QLUE, a computer-vision tool that provides a standardized

qualitative score that assesses the content and functional similarity of the web pages generated by these solutions in comparison to their original versions. QLUE is designed to emulate the human perception of the pages content similarity and their behavior in assessing the interactive features of the pages. QLUE fills a very important gap in today's literature by providing researchers with a uniform scoring metrics that allows them to systematically compare their results against other solutions using a unified tool. Many of today's direct human evaluation strategies are neither standardized nor validated [39], making it nearly impossible to fairly compare results obtained from different user studies campaigns, unless using the exact same human evaluators. Thus, relying on QLUE's systematic scores can help alleviate the aforementioned challenge. To the best of our knowledge, QLUE might be the first attempt to assess the quality of web pages created by different solutions against each other through a unified and standardized metrics. Specifically, QLUE provides two modes of operation: a) standalone, or b) integrated in existing tools, both operating in a fully automated manner with minor user intervention, supporting both desktop and mobile versions of web pages. The main contributions of the paper are:

- Presenting a uniform approach to evaluate the quality of modern web pages using two different scoring metrics, namely content and functional similarity.
- Effectively computing the aforementioned metrics without biasing against any individual element type (in a matter of few minutes).
- Utilizing a *"bot"* that emulates human-like interactions with the page to evaluate the functionality preservation.
- Showing similar qualitative scores when compared to human evaluators.

We evaluated QLUE by comparing the content and functional similarity scores of 100 popular web pages to a conducted users study with 30 participants, each evaluating 20 pages. These pages, taken from Alexa top million sites [6], were simplified using an existing state-of-the-art web acceleration solution[1]. The results show that QLUE exhibits similar trends in the overall similarity scores compared to the user study for both the structural and functional similarity, achieving a structural similarity scores of $\geq$ 90% for more than 70% of the pages in contrast to 90% of the pages for the human evaluators. Similar results were observed in the functional similarity. This highlights that QLUE is less forgiving than the human evaluators. This can be explained by QLUE's systematic rules in penalizing the score for every missing component, regardless of it's size or relevance.

[1]We omitted the name of the solution so as not to violate the double-blind policy.

2 MOTIVATION

Over the past decade, several solutions have been proposed to tackle the web complexity problem from both the industry and academia standpoints. These solutions constitute a promising step towards realizing the United Nation's vision to ensure that digital technologies provide meaningful opportunities for all people and nations [3]. More specifically, the aim of these solutions revolves around accelerating the page load time and improving the overall experience for millions of users who solely rely on low-end mobile devices to access the world wide web. These solutions can broadly be categorized into three categories: blocking solutions, simplification solutions, and pre-processing solutions.

2.1 Web Complexity Solutions

Blocking solutions: Several content-blocking solutions can be used to transfer less-complex pages to web browsers. These solutions can be deployed in a from of in-browser plugins [8, 10, 15, 24, 33]. Recently, Brave [4] was released as a new mobile browser, with built-in ad-blocking features, for a more efficient and privacy-preserving user experience. Additionally, Percival [7] has shown promising results in blocking ads using deep learning, while JSCleaner [12] offers a proxy-based solution to block non-critical JavaScript elements in mobile pages to reduce the processing burdens imposed on low-end mobile devices.

Simplification solutions: These solutions focus on altering some of the page content by either elimination or compression. For example, Facebook-Lite eliminates the secondary features of Facebook [17] to offer a lighter version for users with limited connectivity and low-end phones. SpeedReader [18] is another attempt that is built as a Brave in-Browser tool designed to automatically convert pages that are suitable for the reader-mode into simpler reader-friendly pages. In a recent work [30], the authors proposed WebMedic to eliminate less-useful functions from web page to improve the memory consumption on low-end mobile devices. On the other hand, compression-based solutions focus on the data savings at the client-end. For instance, Flywheel [9] is a service that extends the life of mobile data plans by compressing responses between the servers and the browsers. Moreover, BrowseLite [25] is recent in-browser tool that aims to achieve data savings by applying different image compression techniques.

Pre-Processing solutions: Given the complexity of modern web pages, these solutions restructure the page load process to avoid bottlenecks at the client end [36]. This is achieved mostly by offloading complex processing from the browser to a proxy server. For instance, In the case of Opera Mini [1] browser, a proxy server renders web pages before sending them to the users. Similarly, both Shandian [37] and

Prophecy [32] pre-process web pages on a server and send modified versions to the users' browsers. In contrast, Polaris [31] modifies the sequence in which the different page components are loaded, which results in an overall reduction in page load time.

2.2 Impact of these Solutions on the Page Quality

In principle, web complexity solutions pose an impact on the quality of the pages, where an evaluation is required to ensure high-quality pages, and identifying the different circumstances that might lead to broken pages. In SpeedReader, the authors noted that they did not attempt any user evaluation on the quality of their generated pages, and left a deployment plan with subjective presentation evaluation for future work. Unfortunately, SpeedReader cannot handle around 78% of web pages [2]. One of the reasons might be the fact that it never fetches or executes JavaScript; and we believe that a comprehensive large-scale qualitative analysis can help in covering a larger portion of pages. Similarly, and despite significant speedups, the impacts of pre-processing in Shandian and Prophecy on the page quality have not been evaluated, and a qualitative evaluation is required to understand this impact. In contrast, the impact of BrowseLite on the appearance of the pages is limited to image quality, and no impact is expected on the page structure or functionality. Table 1 provides a summary comparison on the evaluation metrics used in web complexity solutions. As the table shows, while a qualitative evaluation might not be required in compression-based solutions, it is crucial for most of the other simplification and blocking approaches. The table clearly highlights the lack of qualitative evaluations in many of the web complexity solutions apart from very few small-scale user studies.

3 RELATED WORK

QLUE is a tool that can provide a rapid qualitative evaluation of modified web pages with respect to their corresponding reference versions. Despite that web complexity solutions are rapidly emerging, existing tools can only help developers quantitatively evaluate their pages, but not qualitatively in comparison to the existing original pages. For instance, Lighthouse [5] is an automated tool that runs in Chrome DevTools to generate a report on the evaluation of a given web page and how to improve its timing performance metrics such as Speed Index and Time-to-interactive. Similarly, Browsertime [26] tool collects performance timing metrics of a given web page, and records a video of the browser screen used to calculate visual timing metrics (such as Speed Index).

A common framework that assists web developers in creating light-weight mobile pages is Google's Accelerated Mobile Pages (AMP) [19]. To compare the performance of an AMP to the corresponding non-AMP page, the developer must ensure that both pages have similar content and functionality [20], and there is no way today to automate this comparison. Instead of creating new accelerated mobile pages, web developers commonly seek to learn from and reuse features in existing web pages. In Ply [27, 28], the authors proposed a CSS inspection tool to assist web developers in replicating visual features of existing complex pages. To automatically identify irrelevant properties of elements in these pages, Ply disables a given property, captures a screenshot of the resulting web page, and compares it to a reference screenshot with all properties enabled. It uses a simple pixel-level screenshot comparison to compute the visual regression between screenshots and remove the property with no visual impact. However, unlike QLUE, the main limitation of Ply is that it cannot consider interactive features that are driven by JavaScript.

Other existing developer tools [13, 16, 22, 29, 38] focus on debugging and tracking code snippets in web pages, or reproducing interactive content from existing pages. For instance, Chrome Developer Tools are extended in Unravel [22] to support reverse engineering of complex web pages, by enabling developers to track and visualize both: page changes and JavaScript function calls. On the other hand, Fusion [38] allows developers to borrow functionalities from existing web pages, by extracting components from these pages and turning them into self-contained widgets that can be embedded into other pages. In [13], a simple markup language is combined with reactive components to reduce the effort needed to produce interactive documents. In comparison to QLUE, these existing developer tools assume the intention of reusing content on different web pages, while QLUE aims to assist developers who intended to provide alternative pages that accommodate the needs of users relying on handheld mobile devices to access the web.

4 THE DESIGN OF QLUE

To evaluate the quality of web pages created by different web complexity solutions in comparison to their original counter part, we developed QLUE, a computer vision tool that assesses the content completeness and the functionality as two separate scoring metrics (0-100%). As explained earlier, most of today's state-of-the-art solutions that tackle the simplification of web pages either rely on small-scale user studies to evaluate the qualitative aspects of their created pages, or simply ignore the qualitative evaluations and focus more on other quantitative metrics. The QLUE design is inspired by how humans evaluate the quality of web pages in user studies. When it comes to evaluating the content similarity, QLUE tries to emulate the human perception when

Table 1: Evaluation metrics used in web complexity state-of-the-art solutions

Solution	Quantitative Metrics	Qualitative Metrics
SpeedReader [18]	Data Size, Memory Consumption, Page Load Time	Not evaluated
WProf [36]	Page Load Time	Not evaluated
Shandian [37]	Page Load Time, Page Size	Not evaluated
Polaris [31]	Page Load Time	Not evaluated
Vroom [34]	Page Load Time, Above the fold Time, SpeedIndex	Not evaluated
Prophecy [32]	Page Load Time, Bandwidth, Energy, SpeedIndex, ReadyIndex	Not evalauted
Flywheel [9]	Page Load Time, Time to first byte, Time to first paint, Page Size	Not required
BrowseLite [25]	Bandwidth Saving, Page Size, Speed Index	User Study, Visual Completeness
JSCleaner [12]	Page Load Time	User Study
Web Medic [30]	Memory consumption	User Study

comparing the web pages, by searching for missing components, and penalizing the score based on the importance of these components. To this end, QLUE approximates the importance of a component by measuring the area it occupies in the page—emulating humans behavior penalizing large missing components, while often overlooking small missing ones. On the other, when evaluating the functional completeness, QLUE utilizes a computer bot that interacts with all the functional elements of the page and assesses the missing functionality. In comparison to a human evaluator, the bot automatically extracts a complete list of event-listeners and then triggers each one individually, some of these events might not be easily identifiable (i.e., missed) by a human evaluator—since some are hidden under a number of consecutive dependant events.

QLUE compares a version of a page against its original counterpart, we refer to these pages as the *Modified page* and the *Reference page*, respectively. As stated earlier, QLUE performs two different comparisons of the modified page against its reference, these are: content completeness, which we technically refer to as *Structural Similarity*, and the degree to which the interactivity and the functional features are retained, which we refer to as *Functional Similarity*.

4.1 Structural Similarity

The structural similarity aims to compute a score on how content from the reference page is retained in the modified one. In order to compute the score, the structural similarity goes over various processes, these are: i) screenshots generation and processing, ii) components extraction, iii) components matching, and iv) computing the score. Figure 1 shows a high-level architecture of QLUE's structural similarity.

4.1.1 Screenshots Generation and Processing. To compute the structural similarity (i.e., content completeness), QLUE first requires to generate a screenshot for both rendered pages–the modified and the reference. A rendered page is created by the browser after the page is requested and its HTML representation is processed by the browser, which in turn requests and display all the page embedded objects (such as images, videos, and etc.). To assess the full page content completeness, the screenshots of the entire pages are required for the comparison. More specifically, the screenshots are generated while scrolling down till the end of the page. In other words, a screenshot not only considers what is usually refereed to as above-the-fold (which is the part of the page that is visible on the screen), but also the part that appears when scrolling down. These screenshots are represented as images in the form of 3D arrays: the first two dimensions are the X and Y coordinates for the pixels of the image and the 3rd dimension is the color space (the RGB values of the pixel). In order to find the interconnectedness of the pixels, we transformed this 3D array image into a 2D binary array (with zeros and ones), where the ones represent the presence of the components and the zeros represent their absence. In other words, we transform the screenshot into a black and white representation for the sake of speeding up the computation.

After generating the full page screenshots for both the reference page and the modified page, QLUE identifies the background color of the page by reading the Document Object Model (DOM) property *document.body.style.backgroundColor* and comparing it with the most frequent pixel-color on the page, given that sometimes web developers add an additional gradient background or an image background that overlays the background color. If different, the image is run through OpenCV's adaptive threshold filter. Otherwise, the image is run through a more efficient custom threshold filter that we built, which replaces background pixels with zeros and foreground pixels with ones. The reason why we do not use OpenCV's threshold filter in case the background color matches the most frequent pixel-color is that it breaks the images into several sub-components, which in turn increases the time complexity of the process. In other words, if there is an image that contains several objects, instead of taking

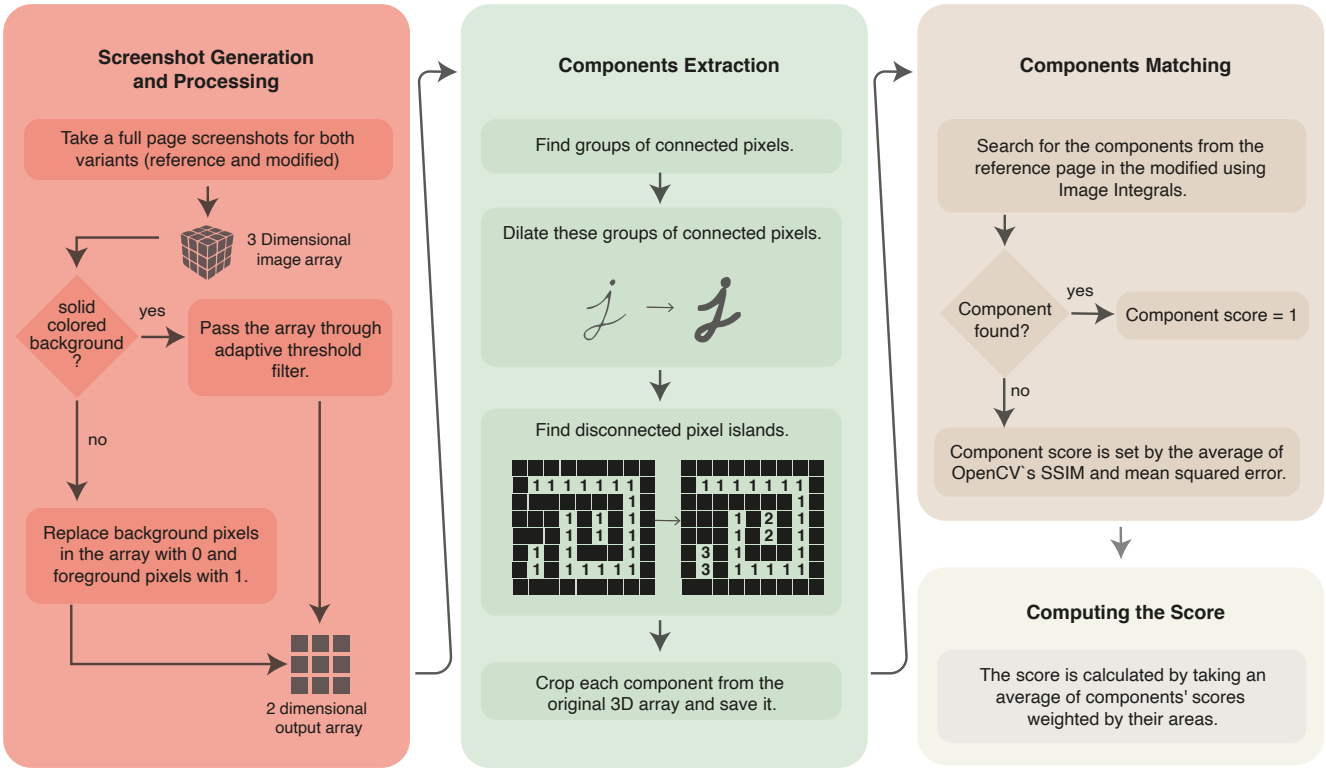


Figure 1: Content Similarity Evaluation

the entire image as one component, OpenCV's threshold filter splits all objects inside the image into separate components. Figure 2 shows an example comparing the outcome of OpenCV's adaptive threshold filter to the custom threshold filter. It's worth noting that for the purpose of identifying the individual components and their locations within the page, we do not require the individual objects within each image, instead we care about the entire image as a whole, since the modified version will either have the entire image or not (there won't be a case where only a subset of the image is retained within the modified page).

4.1.2 Components Extraction. Now that the screenshots for both the modified and the reference pages have been transformed into separate 2D binary arrays, the next step for QLUE is to split each array into a number of separate components. The rational behind the components splitting comes from the fact that a direct comparison using openCV's Structural Similarity Index Measure (SSIM) of the two arrays (representing the modified and the reference ones) might not yield an accurate similarity score. This is because the modified version may lack a number of elements, which consequently affects the position of other elements within the page (given the dynamic nature of HTML that reorders the components based on their relative positions). Figure 3 shows two examples of how the SSIM fails in providing an accurate score. In the first example, due to a missing banner, the modified version of the page has shifted the subsequent elements following the missing banner. As result, the SSIM method reported a 67% similarity between the two pages (which is clearly not accurate given that both pages look almost identical apart from the missing banner). On the other hand, the second example shows two different screenshots of different pages (where the modified version is not created from the same reference page), however the SSIM method reported a similarity score of 90% even though that the elements within the pages are completely different (due to the fact that there are many background pixels that match between the two pages).

To overcome the above limitation, QLUE breaks the pages into several components while eliminating the background pixels. The score can then be computed by matching these components individually (discussed later in Section 4.1.3). To identify the individual components, QLUE searches within the 2D binary array for groups of connected pixels (i.e., adjacent 1s in the array without 0s in between), we refer to

(a) Original screenshot (b) OpenCV's adaptive threshold filter (c) Custom threshold filter

Figure 2: An Example showing the impact of two different threshold filters on the same original screenshot. (a) shows the original screenshot, (b) shows the OpenCV's threshold filter which highlights that each image is split into multiple smaller components, and (c) shows the custom QLUE threshold filter where each image is a standalone component.

these groups as islands. These islands can consist of big elements such as images, or small elements such as alphabets in a sentence. For many of the small close-by islands, QLUE merges them into a bigger island for two main reasons: i) as the number of separate islands in the reference page increases, the time to search for them in the modified page will also increase, and b) islands that often repeat in a page (such as alphabets) are required to form unique combination (i.e., a sentence or a paragraph island) for easier matching with the modified page. This unique combination of forming larger islands is essential in making sure that the missing components in the modified page are correctly penalised and not falsely matched due to the presence of repetitive smaller islands.

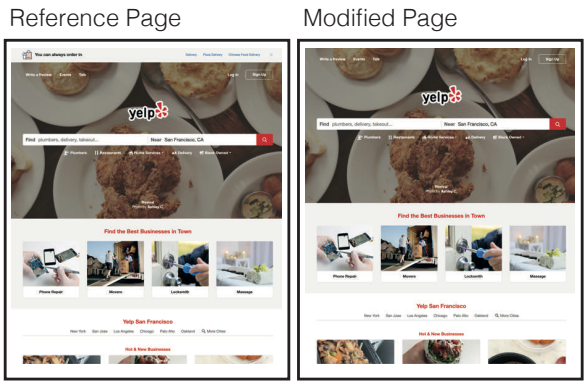
To join the small islands together, QLUE uses OpenCV's dilation function to "dilate" these pixel islands. Pixel Dilation is a morphological operation that traverses through the binary array and replaces all the 0s with 1s if one of the neighboring pixels is a 1. Figure 4 shows an example of the dilation process over a textual paragraph. The bounds of the neighborhood can be changed by modifying the kernel size. For example for a kernel size of $(2, 2)$ it will check the 2x2 squares around the pixel in question. A larger kernel size would result in merging farther islands together. For QLUE, it is crucial to have a kernel size that captures islands correctly—we don't want the entire screenshot to be a single island, nor every alphabet to be an island on its own (see Figure 5 on the impact of the kernel size). The initial kernel size values are set by multiplying a constant number with the ratio of the foreground and the background pixels in both the x and y directions. This constant is different depending on the type of the web page (mobile or desktop), as well as the display screen resolution.

The initial value, K of the kernel is defined as:

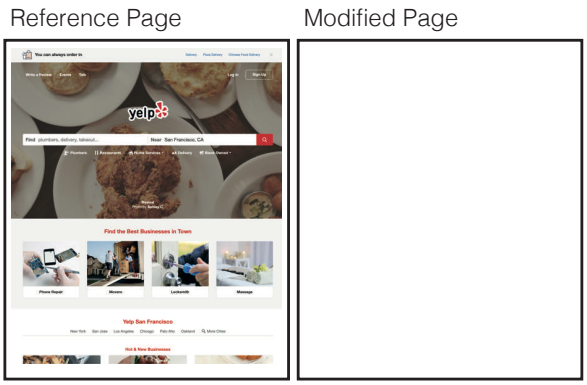
$$K = (K_x, K_y) = \begin{cases} K_x = \frac{C}{numRows} \sum_{i=1}^{\text{numRows}} \frac{\delta_i}{\gamma_i + \delta_i} \\ K_y = \frac{C}{numCols} \sum_{i=1}^{\text{numCols}} \frac{\delta_i}{\gamma_i + \delta_i} \end{cases} \quad (1)$$

where δ is the number of pixels with value 0 at the i^{th} row or column (depending on whether we are computing k_x or k_y), and γ is the number of pixels with value 1 at the i^{th} row or column (depending on whether we are computing k_x or k_y), and C is a constant value that inflates the probability of zero pixel into a larger dilation factor, given that the probability is between 0 and 1. C is chosen to be 15, which intuitively means that when the probability of the zero pixel per screenshot is low (i.e., the page is dense in components), the kernel size would be roughly around 1 to 2 pixels. On the other-hand, when the probability of the zero pixel is high (i.e., the page is sparse in components), the kernel size would need to be a higher value not exceeding 15 pixels.

After the initial value of K is computed for a given page using the above formula, QLUE dilates the pixel of the screenshot and computes the number of islands (i.e., disconnected components). The number of components is computed by using openCV's connected components method (see Figure 1 green process, where the pixels of each connected component are labeled with a unique number from 1 to N). Based on analysing 100 modern web pages, we have discovered that on average the number of components per page varies in area between 4000 and 12000 pixels. To verify the initial value of our kernel K, we compare the number of



(a) 61% SSIM similarity score, while intuitively a higher score is expected given that the pages look very similar



(b) 69% SSIM similarity score, while intuitively a score of 0% is expected given that the pages are completely different

Figure 3: An example highlighting two failure cases of OpenCV's SSIM in providing an accurate similarity score due to either missing certain elements that causes the rest of the page elements to shift (as in a), or due to the fact that OpenCV's SSIM considers the background pixels when computing the score (as in b).

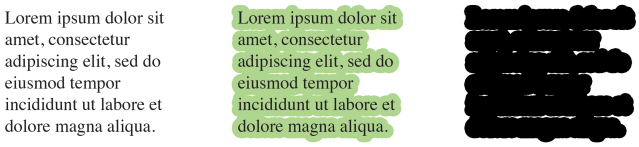


Figure 4: An example of the dilation process applied on a textual paragraph. Note that the intermediary figure is only displayed for illustrative purposes, to highlight the pixels that are being dilated (shown in green).

components computed earlier with the theoretical upper and lower bounds, which are compute by dividing the screenshot area by both bounds of the average component area. That is $\lfloor (x \times y)/12000 \rfloor$ and $\lfloor (x \times y)/4000 \rfloor$, where x and y are the width and height of the 2D array. If the number of components is outside this range, the kernel size will be automatically tweaked depending on whether the number of components is lower or higher than the range. In case it is lower it suggests that the kernel size is larger than required and should be reduced, whereas it should be increased if the number of components exceeds the upper range (which means that the kernel size is smaller than required). This kernel size is automatically increase/decreased by 1 pixel in each iteration and the dilation process is performed again until the number of components is within the range.

Once the number of components is computed and within the range, the bounding box of each component is computed by finding the minimum and maximum x and y coordinates across all pixels of that component, i.e. (x_{min}, y_{min}), (x_{max}, y_{min}), (x_{min}, y_{max}), (x_{max}, y_{max}).

QLUE stores the coordinates of each component, and uses them to extract the components images by cropping the original 3D array at the same coordinates of the bounding boxes. Figure 6 shows how the bounding boxes overlay over both the reference and the modified original screenshots.

4.1.3 Components Matching. Now that QLUE has extracted all the individual components from both the reference and the modified pages (in the form of images cropped from the 3D arrays of the reference and modified screenshots), its time to match these components in order to identify which ones are missing in the modified page. To achieve the latter, QLUE utilizes an image search algorithm called the *image integrals* [14] to match each component in the reference page with a component in the modified one. For all the successfully matched components, QLUE assigns a full similarity score of 100% for that component (Figure 6 shows an example of the matched components, highlighted by the green bounded-boxes). For the rest of the components that are not found in the modified page, QLUE uses OpenCV's similarity index and the mean squared error for each of the unmatched components from the reference page against each similarly-sized unmatched components from the modified page. Then, for a given unmatched component in the reference page, the component from the modified page with the highest score is considered a partial match with the score obtained from OpenCV's similarity index which ranges between 0 - 100%. For the reference page components with no matches, i.e., when QLUE runs out of components to match in the modified page, we consider a score of 0% for each of them (see Figure 6 for an example of such unmatched components

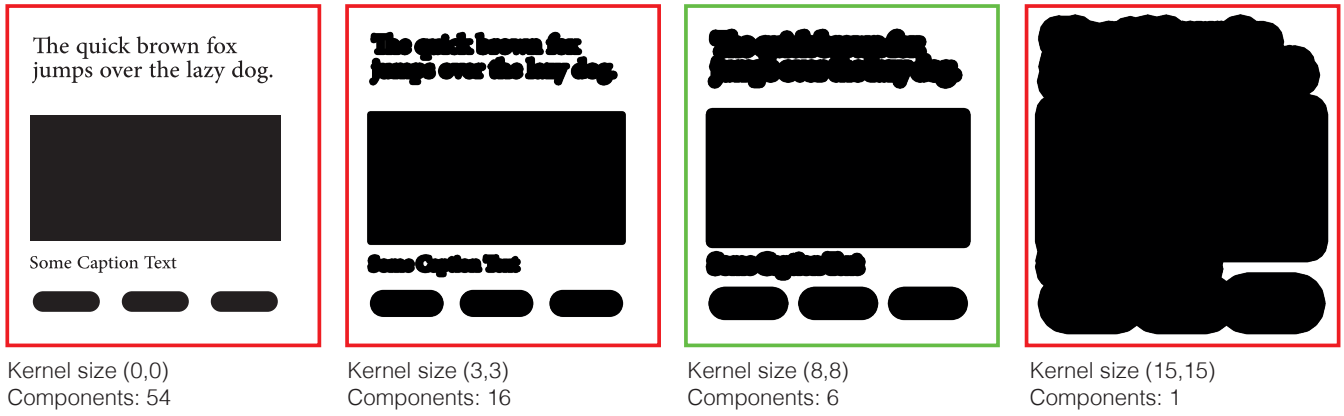


Figure 5: The impact of different kernel size on the pixels dilation process. When a small kernel size is used, e.g., (0,0), this results in having many small components (each alphabet is considered as an individual component). On the other-hand, when a large kernel size is used, e.g., (15,15), the entire page is represented by a single component. In this example, kernel size (8,8) shows the best components' splitting results.

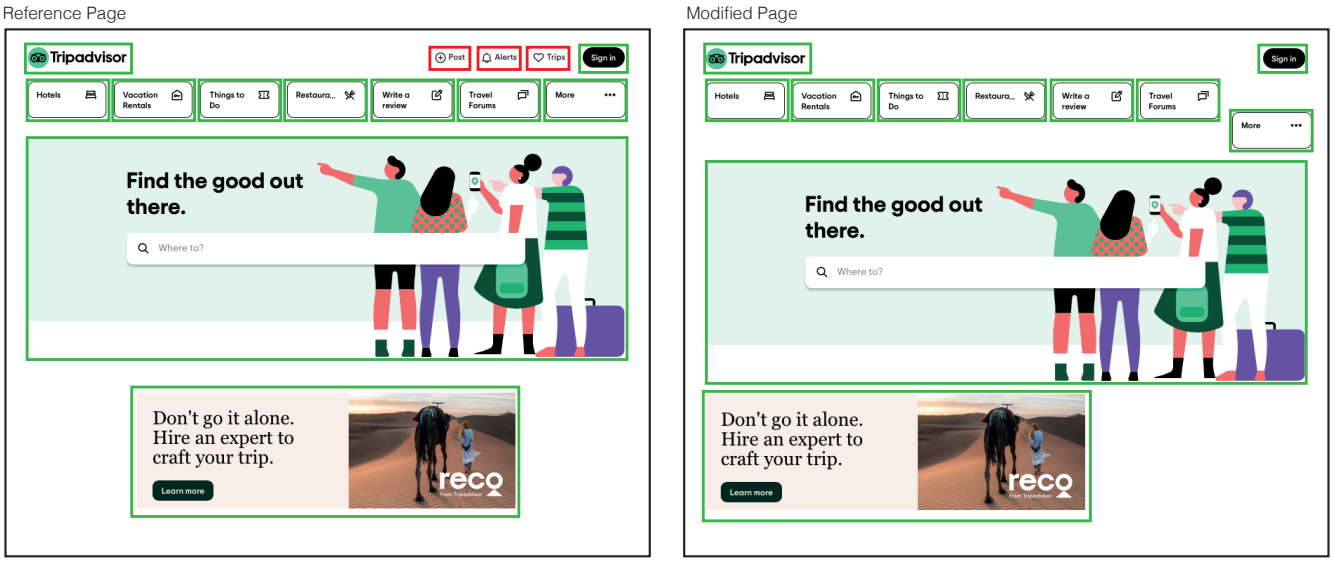


Figure 6: An example of QLUE's components matching process, where green bounded-boxes highlight the fully matched components between the two pages, and the red bounded-boxes highlight the missing components from the modified page. Note that regardless of the actual components position, QLUE can still easily match them. For example, the ad displayed at the bottom of the page has been left-aligned in the modified page in comparison to the center alignment in the reference page. Additionally, the *more* item at the end of the navigation bar within the modified page has been shifted below it's actual position in the reference page.

highlighted by the red bounded-boxes). Upon matching completion, the result would be in the form of a list of matched components each with an individual score between 0 - 100%.

4.1.4 Computing the Final Score. From the previous step, QLUE computed an individual score for each component in the reference page. Now, these scores must be combined into a unique score that represents the overall page similarity. One naive approach to compute the overall similarity score is to simply average over all of the individual components' scores. However, such an approach might not yield a correct

score that represents the content completeness between the two pages. For instance, if one of the components that has relatively a low score turned out to be a small missing icon or thumbnail within the page, the weight of this element would unfairly penalize the overall similarity score given that when the same task is given to humans, they generally tend to penalize for missing large elements, whereas they might forgive or even overlook smaller missing elements within the page. Inspired by this, QLUE averages the scores weighing each element score by its relative area. That is, the scores are multiplied by the area of each component and divided by the overall area of all components. This way, each component contributes to the final score in a proportion relative to its area. Consequently, smaller components would have marginal impact on the similarity score. This gives a Structural Similarity Score between 0 and 100%, where a score of 100% affirms that the pages are identical (lack of missing content in the modified page).

$$\text{Structural Similarity Score} = \frac{\sum_{i=0}^{C} a_i \times s_i}{\sum_{i=0}^{C} a_i} \quad (2)$$

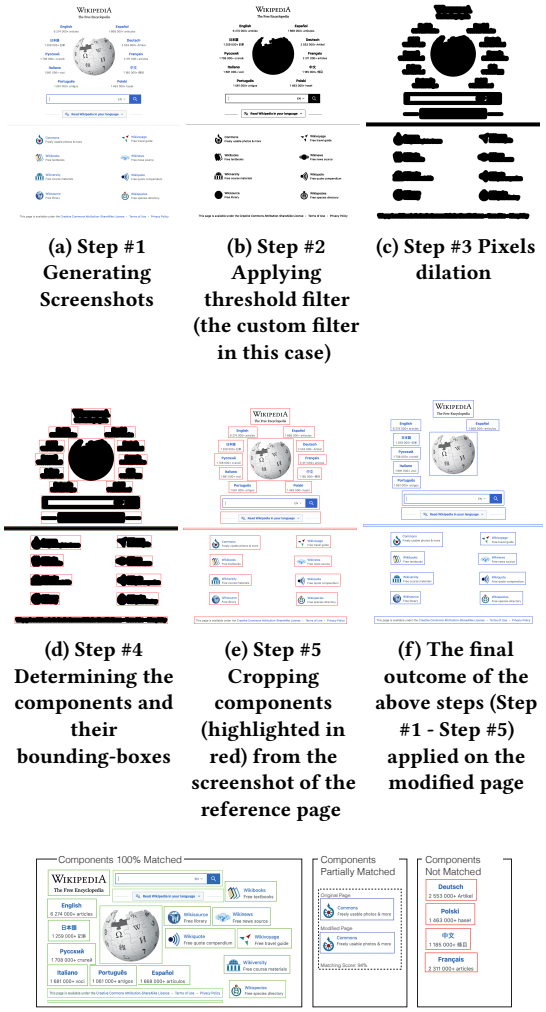
where C is the number of components, s_i is the score of the component i, and a_i is the area of component i.

4.1.5 Walk-through example. This subsection illustrates a step-by-step walk-through example of the QLUE's structural similarity evaluation. For this example, we have chosen the *Wikipedia* mobile page due to its simplicity to highlight the individual processes.

- Step #1 Generating Screenshots: First, QLUE obtains a screenshots for both the reference and the modified pages, that is the 3D image array (Figure 7a)
- Step #2 Applying threshold filter: Next, QLUE applies the threshold filter, as seen in Figure 7b, to convert the 3D image array into a 2D binary array (representing black and white pixels). In this example, QLUE uses the custom threshold method to convert the image since the background color of the page taken from the document object model attributes matches the most frequently used pixel in the page (that is the white color pixel).
- Step #3 Pixels dilation: Once the 2D binary array is created, QLUE uses the dilation technique explained earlier to build the islands' representation of the connected pixels so as to determine and separate all the individual components within the page. As can be seen in Figure 7c, each of the Wikipedia components is grouped into a single island, for example each of the sections around the Wikipedia globe (at the center of the page) forms an independent component, merging both the title and the underlying text, e.g., merging the "English" title with the text underneath it.
- Step #4 Determining the components and their bounding-boxes: QLUE uses openCV's connected components methods, each of the pixels islands is identified, and each of its bounding-boxes is determined as seen by the red colored boxes in Figure 7d.
- Step #5 Cropping components from the original screenshots: applying the same bounding-boxes coordinates obtained in the previous step, QLUE crops the components' images from the original screenshots as seen in Figure 7e. These images would serve as the basis for components matching between both versions of the page.
- Step #6 Matching components across the two versions: QLUE iterates over all identified components in the reference page and searches for possible matches in the modified page. Figure 7g shows three different categories of components: a) components that are fully matched with a score of 100% (highlighted in green), such as the Wikipedia globe, the Wikipedia title, etc., b) partially matched components (highlighted in blue). Note that in this example we only have one such component where the text associated with the logo is shifted from its original location in the reference page (leading to a similarity score of 94%), and c) missing components (highlighted in red), these components were not present in the modified page, hence their score is set to 0%.
- Step #7 Computing the final score: this step simply averages the scores of all the components present in the reference page, while weighing each component's score proportional to its area. As such, the overall page structural similarity score is 89%.

4.2 Functional Similarity

Functional similarity refers to the assessment of the modified page in comparison to the reference page in terms of retaining the functional elements and their proper interactivity features. For example, a drop-down interactive menu in a reference page should: a) be present in the corresponding modified page and b) its functionality is preserved in the modified page. To compute the functional similarity score, the user interactivity events are emulated, through an *interaction bot*, by extracting all of the event listeners from both the reference and the modified pages, triggering each of the events, and taking screenshot whenever an event is triggered. Then, the functional similarity score is computed based on the retained functionality in the modified page with respect to the reference page.



(a) Step #1 Generating Screenshots

(b) Step #2 Applying threshold filter (the custom filter in this case)

(c) Step #3 Pixels dilation

(d) Step #4 Determining the components and their bounding-boxes

(e) Step #5 Cropping components (highlighted in red) from the screenshot of the reference page

(f) The final outcome of the above steps (Step #1 - Step #5) applied on the modified page

(g) Step #6 Matching components obtained in (e) and (f), showing three categories of matched components: 100% matched (green), partially matched (blue), missing (red)

Figure 7: A walk-though step-by-step example to evaluate the structural similarity. Note that sub-figures a-d show the steps applied on the reference page only, given that the process is the same for the modified page.

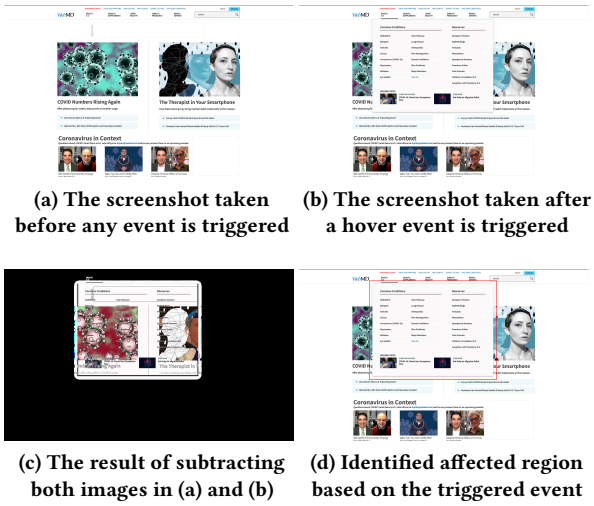
4.2.1 Emulating User Interactivity. To emulate the user interactivity with a given web page, we leverage the browser's built-in functionality to identify all the event listeners in the reference page, and map them to their corresponding elements. These events can come in various forms including but not limited to: *mousedown*, *mouseup*, *mouseover*, *mouseout*, *keydown*, *keypress*, *keyup*, *dblclick*, *drag*, *dragstart*, and *dragend*.

Elements are identified using their *XPaths* that facilitates navigating through a given web page and accessing all elements and their attributes to construct an event-dependency graph, such that a group of dependent events can be triggered in a given order. For example, an event that closes a menu cannot be triggered unless the open event corresponding to that menu is triggered first. An *XPath* of an element is constructed using the nearest parent with an "id" or "class" attribute as the root. In an HTML document representing a given web page, elements are structured internally using different $< div >$ tags. Each $< div >$ tag has a unique "id", and can reference an pre-defined "class" of attributes from the accompanying Cascading Styling Sheets (CSS) files, which set the different visual attributes for that given $< div >$ tag. Position-based indexing is then used to identify descendants. For elements with no ancestor with an "id" or "class" attribute, the "body" element is used as the root, while position-based indexing is initiated from the body. Using XPath element identification allows us to identify elements across reloads.

When the events of a given reference page are identified and mapped to the corresponding page elements, they are traversed in a depth-first order. For each element, an automated browser is used to trigger each of the events associated with that element. In general, triggering an event leads to changes in the page appearance. Figure 8 shows a sample snapshot of a hover event over a drop-down menu.

In order to assess if the functionality is retained, a screenshot is taken and stored whenever an event is triggered, Figure 9 shows examples of such events. However, any screenshot that renders no visible change, in comparison to the master screenshot of the page before any event is triggered, is not stored in order to increase the efficiency of computing the functional similarity score. This is confirmed by a pixel-by-pixel comparison, such that an event that does not change the appearance of the page (when triggered) is dropped from the list of events. The final set of event listeners identified in the reference page along with their screenshots showing the impact of each on the page are then used to assess the retained functionality in the modified page. For these screenshots, QLUE does not save the entire screenshot area but rather only the area that shows a visual change in comparison to the master screenshots. This is achieved by subtracting both pixel by pixel and eliminating the zero value pixels (i.e., the ones that were identical). QLUE searches for each of these event listeners in the modified page, and triggers each of them to generate and store similar screenshots from the modified page for each of those events. Similar to above, only the part of the screenshot that shows a visible change is saved.

4.2.2 Computing the Score. QLUE iterates over all screenshots generated from the reference page that showed a visual



(a) The screenshot taken before any event is triggered

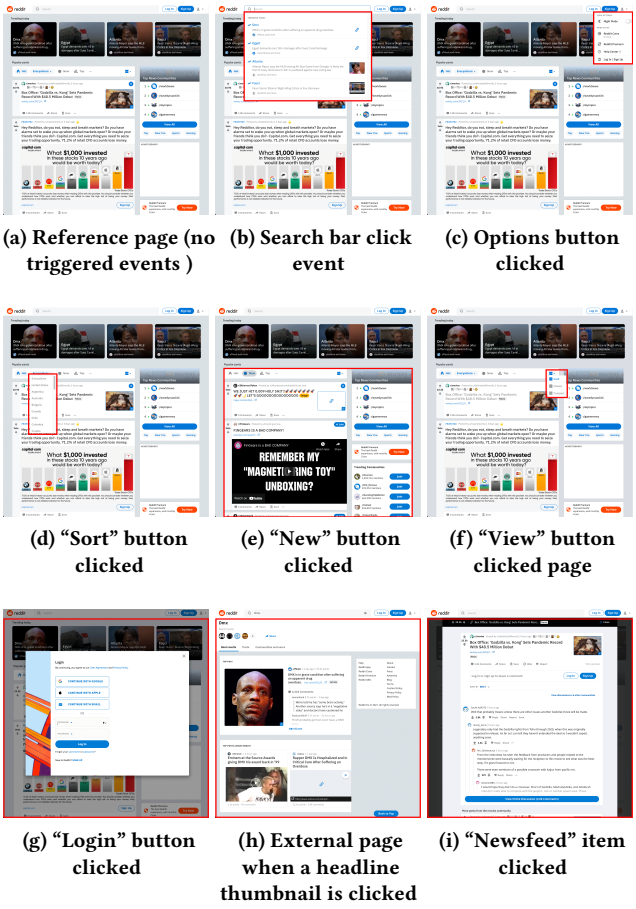
(b) The screenshot taken after a hover event is triggered

(c) The result of subtracting both images in (a) and (b)

(d) Identified affected region based on the triggered event

Figure 8: An example of an interactive functional element, corresponding to a hover event-listener that triggers a drop-down menu. The figure also shows how QLUE eliminates the unaffected portions of the page by subtracting the screenshots (sub-figure (c) is the result of subtracting both images in sub-figure (c) and (a)). This is done to effectively only compute the score for the region that showed a visible change (highlighted by the red bounding-box)

change in the page appearance upon triggering an event, and compares each to its counterpart screenshot taken from the modified page. QLUE has two different approaches when comparing these screenshots: a) using the same structural similarity approach taken earlier for the content completeness (see Section 4.1, or b) simply using openCV's SSIM method for a quick similarity score computation. The choice between the two is left for the QLUE user, since this strikes a trade-off between accurately computing the score (option a), and the time complexity of computing the functional score (where option b is tentatively faster than option a). As shown earlier, SSIM might not be accurate when it comes to measuring similarity scores especially when missing elements can lead to the reordering of page elements (see examples shown in Figure 3). However, in the case of a functional element screenshots, the possibility of having a missing element is rare, given the fact that an event element functionality is either fully retained or completely missing. An event that exists in the reference page and is not found in the corresponding modified page is given a score of zero. On the other-hand, for a matched event, a score is between zero and one is computed to assess the similarity of the screenshot taken when that event is triggered in the modified page with respect to the corresponding screenshot taken when the same event is triggered in the reference page (using image subtraction). The average of the scores given to each of the event listeners is computed to represent the overall functional similarity score of the modified page in comparison to the associated reference page. The functions of a given page are considered equally important and consequently, no weights are given to different types of event listeners (unlike the case of the structural similarity score).



(a) Reference page (no triggered events)

(b) Search bar click event

(c) Options button clicked

(d) "Sort" button clicked

(e) "New" button clicked

(f) "View" button clicked page

(g) "Login" button clicked

(h) External page when a headline thumbnail is clicked

(i) "Newsfeed" item clicked

Figure 9: Examples of QLUE's *interaction bot* triggering different types of event listeners. Red boxes highlight the impacted area of the page when each of the events is triggered.

5 IMPLEMENTATION

QLUE provides a standalone implementation, where the users can locally install the tool on their machine. The implementation is split into two modules: structural similarity, and functional similarity. These modules are all available

on GitHub as open source libraries[2]. QLUE's modules are combinations of both Python and Java code, relying on a number of openCV's functions.

5.1 Structural Similarity Implementation

For the structural similarity implementation, QLUE provides a flexible approach with three different modes of operation depending on the usage scenario, these are:

- *Proxy mode*: here, the users configure QLUE with two different proxies, one serving the reference pages, while the other serves the modified pages. The use of proxies is a common approach for many of the web acceleration solutions, where the solutions' developers cache cloned versions of these pages to guarantee reproducible results.
- *URL mode*: in this mode, QLUE can be used to evaluate live web pages against each other without the need for caching. A usage example of this mode can be evaluating live production web pages where web developers host two different live versions of the same page.
- *Screenshots mode*: this mode is used to directly provide the pages screenshots (i.e., reference and modified). In contrast to the above two approaches, where QLUE generates the screenshots internally.

For the screenshot generation, QLUE uses Selenium Chrome Web-driver [23], where depending on the need of the users, their solution, and their selected mode, QLUE can be configured to either emulate a desktop or a mobile phone chrome browser when generating the screenshots (valid for the first two modes of operation). It also runs as a headless browser, and automatically scrolls through the entire page in an iterative manner while waiting in each iteration for the content to be fully displayed. The maximum number of iterations is configurable (and is set to 20 iterations by default). This number is introduced, given that certain web pages can virtually display endless content when scrolling.

Following the screenshot generation, QLUE automatically, without the intervention of the user, passes these screenshots to the following subsequent processes: pixel dilation, components extraction, components matching, and computing the final score. These are all implemented in Python using openCV [11], Scikit-Image [35], and numPy [21].

5.2 Functional Similarity Implementation

The functional similarity implementation has two main modules: an *interactivity bot*, and a *score generation* module. The *interactivity bot* emulates user actions on the web page by first extracting all the event-listeners from the DOM structure of the page. Given that all of the pages interactivity/functionality are triggered using an event-listener. The *interactivity bot* is implemented in Java, relying internally on Selenium functions. The bot can be configured to operate using two different modes (similar to the first two modes of the structural similarity mentioned above): *Proxy mode*, and *URL mode*. The *score generation* module is implemented in Python. The initial comparison to see if the events rendered any change, image subtraction and image matching are performed using openCV [11] and Scikit-Image [35].

6 EVALUATIONS

6.1 User Study

To evaluate the effectiveness of QLUE on how well it emulates the human perception when comparing web pages against each other, we conducted a user study with 30 participants to compare 100 modified web pages[3] with respect to their original pages. We split the 100 pages across the participants, and ask each to evaluate 20 unique pages. In total each page was evaluated by 6 participants. The participants were recruited from an international University campus, and were trained to manually evaluate the quality of the pages with respect to their original counterpart pages in terms of structural similarity (i.e., content completeness) and their retained functionality. We met with each participant online to explain the purpose of the evaluation and how to use the evaluation tool.

6.1.1 Evaluation Tool and Metrics. We designed an evaluation tool that automates the comparison process for the human evaluators. That is, the tool automatically picks a URL from the list of URLs the evaluator is supposed to assess, and displays side-by-side both versions of the page (the reference and the modified) by opening two instances of the Chrome browser. Figure 10 shows the evaluation tool interface with an example web page, *BBC.com*.

The left window connects to a proxy that servers the reference pages, and the right window connects to a different proxy that serves the modified pages. The reason behind using proxy servers is to serve the same cloned versions of the pages for all users (to avoid the page regular updates over time). The evaluation tool randomly selects a web page for evaluation. The user is asked to compare the two pages and fill in a form with the following considerations:

- Content Similarity score: where the evaluator is required to rate her/his perceived content similarity of the modified page in comparison to the reference page using a slider with a 0-to-100% scale. A score of 0% is

[2]The link is omitted so as to respect the double-blind review policy.

[3]These pages were created using one of the state-of-the-art web complexity solutions. We omitted the name of the solution so as not to violate the double-blind policy.

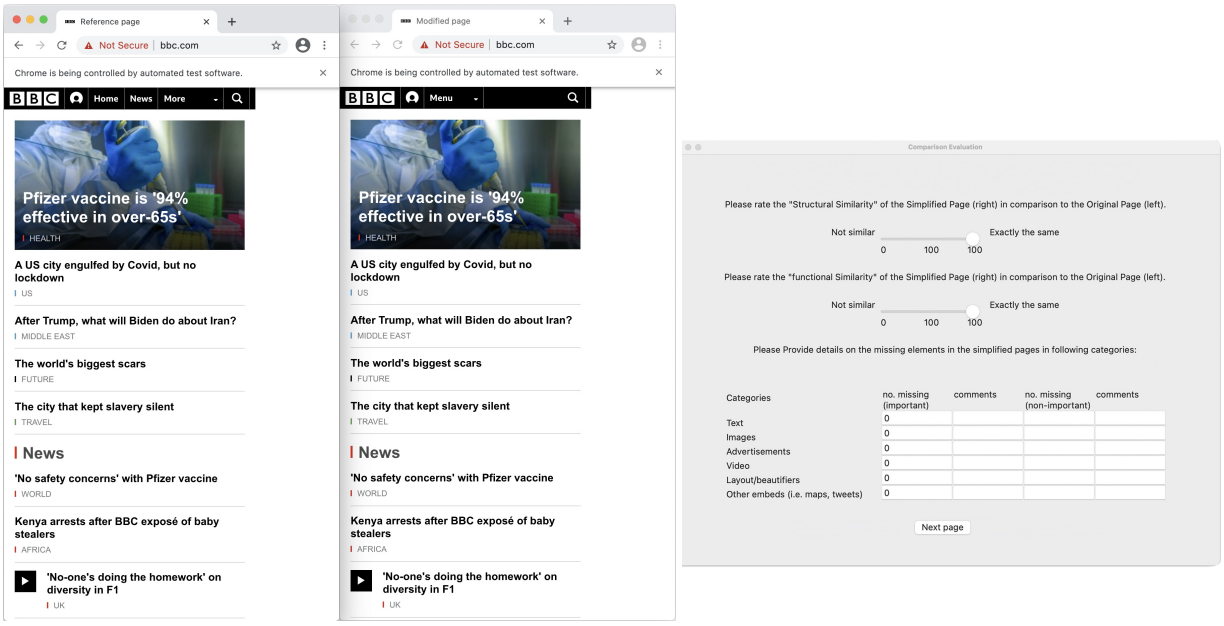


Figure 10: The Evaluation Tool showing an example web page to evaluate (BBC.com)

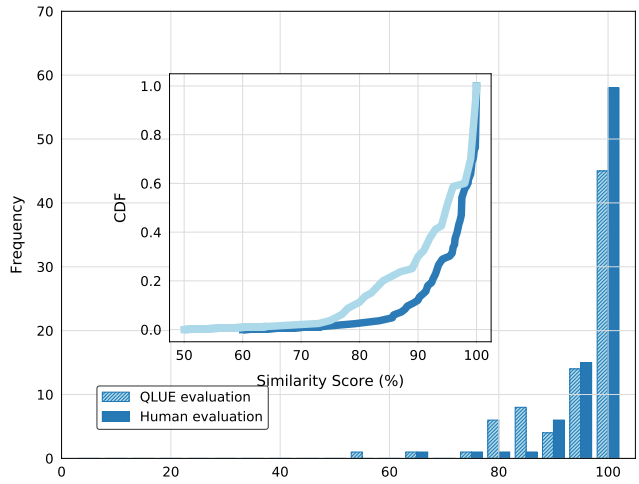
interpreted as the two pages being completely different, whereas a score of 100% means that the two pages are identical. A score between 0% and 100% refers to a partial similarity that matches the given score.

- Functional Similarity score: which refers to evaluator perceived similarity score of the modified page compared to the reference one from the functional completeness perspective. Participants are asked to rate their perceived functionality score in a 0-to-100% scale by manually assessing the presence as well as the operation of all functional elements found in the reference page within the modified one. These elements includes but not limited to: interactive menus, navigational elements, search bars, and image scrollers, etc.
- Missing Content: where the participant is requested to quantify the number of missing elements based on their importance, categorised into 6 categories (see Figure 10), within the modified page. Additionally, the participants are also asked to describe why have they split these elements into important vs. non-important elements.

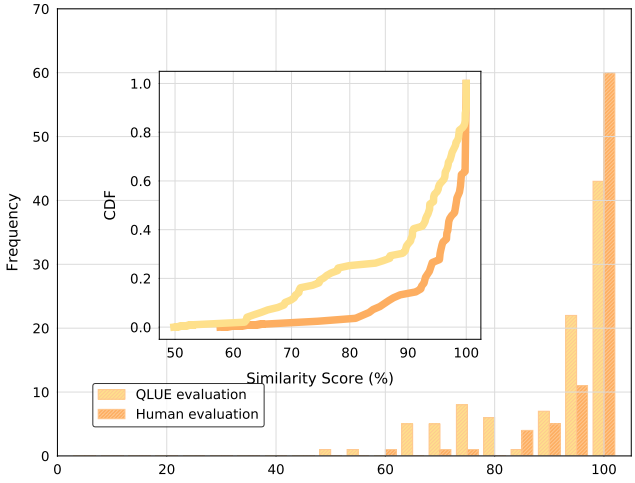
6.1.2 Evaluation Results. Figure 11 shows the histograms and the cumulative distribution functions (CDFs) of the structural (Figure 11a) and the functional similarity (Figure 11b) of the modified pages in comparison to the corresponding reference pages for both the user study and QLUE.

The structural similarity results, shown in Figure 11a, compares the automated QLUE results highlighted by the light blue curve in comparison to the manual human evaluations highlighted by the dark blue curve. For the user study results, it can been seen that for 90% of the pages, the human evaluators gave a score $\geq$ 90%, whereas for the rest of the 10% of the pages, almost all (with the exception of two outliers) have a score $\geq$ 80%. In comparison, QLUE results show more conservative scores, where 75% of the pages have a score of $\geq$ 90%, while the rest (apart from 3 outliers) scores between 75%-90%. This highlights that QLUE is less forgiving than the human evaluators, evident by the smooth and gradual increase of the scores. This can be explained by the fact that QLUE systematic rules in penalizing the score for every missing component no matter how small it is, or how important is the component to the page main content. In summary, QLUE's structural similarity score can be considered as the lower bound of the page content evaluation.

Similar observations to the above can be viewed in QLUE's functional comparison shown in Figure 11b. That is, QLUE scores follows a similar trend compared to the scores given by human evaluators with slightly lower values. This can be explained by the fact that human evaluators tend to overlook minute differences, and that they are more forgiving in their assessment when major elements in the two pages are matching. In addition, human evaluators tend to miss evaluating certain functional elements, especially when they are triggered after triggering a series of previous dependant events.



(a) Structural similarity



(b) Functional similarity

Figure 11: QLUE vs. human evaluation results

6.1.3 How QLUE compares to human evaluators. In Figure 12, we illustrate three examples on how the structural similarity scores computed by QLUE are comparable to the scores given by the human evaluators.

In Figure 12a, we show that QLUE computes approximately the same score given by the human evaluators. The 2% reduction in QLUE's score reflects the two missing menu elements (highlighted in red at the upper right corner of the reference page). QLUE identifies exactly the same missing images in the reference page as the human evaluators. Similarly, Figure 12b shows that QLUE identifies the same missing elements as the human evaluators, however, QLUE penalize the similarity score more accurately, since the size of the missing images spans a large area in the reference page. The reason why human evaluators gave a higher score than QLUE is that they attributed the missing images to advertisements (reported as non-important missing elements in the form shown in Figures 10). We believe that it is crucial to consider all missing elements equally regardless of their perceived category (e.g., advertisements) when measuring the qualitative score of the pages modified by web complexity solutions. This is because QLUE aims at providing a unified qualitative scoring metric to compare different web complexity solutions in a uniform and unbiased manner. Finally, Figure 12c shows the comparison of the *BBC.com* web page. Here, QLUE reports a similarity score that is 10% lower than the score given by the human evaluators due to the fact that it correctly identifies many missing navigation elements at the end of the page—that were overlooked by the human evaluators. This example highlights how easy it is for human evaluators to overlook many elements because it is very hard to recognize them when they have similar structural appearance.

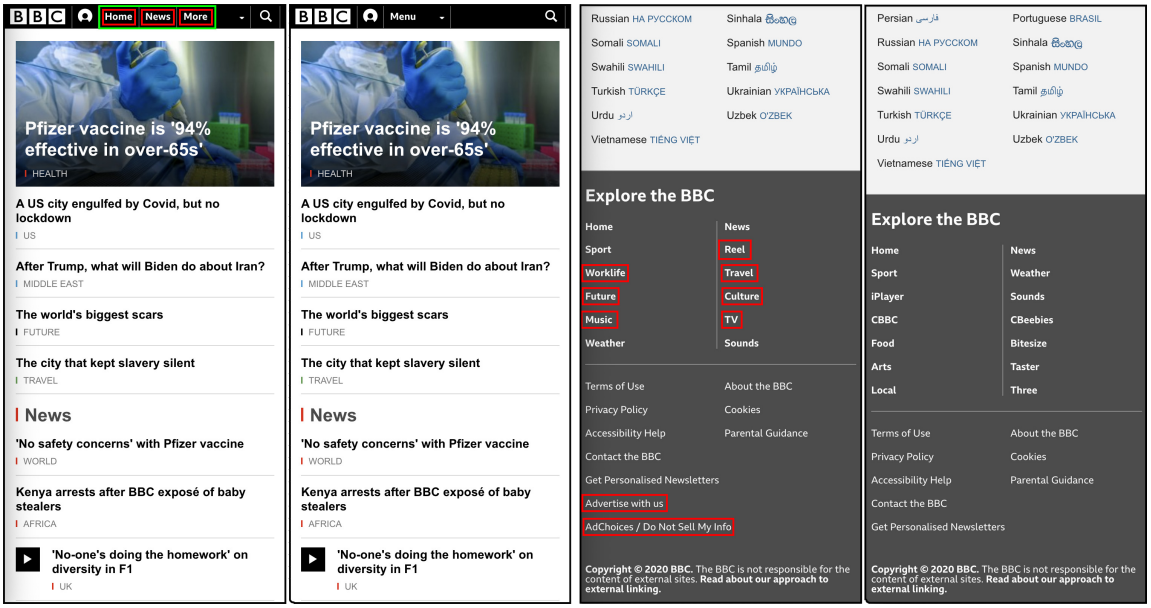
6.2 Time Complexity

To evaluate the performance of QLUE in terms of time complexity, we compared the different timing metrics on per-process basis, using the same set of 100 pages considered in the user study. These metrics represent the timings of the most time-consuming processes in computing the structural similarity score: threshold filter, components extraction, components matching, and the total required time (structural similarity total time). In Figure 13a, we show the CDFs of the QLUE's timing metrics measured in seconds, whereas in Figure 13b, we show these timing metrics as a function of the number of components in a web page. Figure 13a clearly shows that the threshold filter does not impact the overall time given that it is completed in a matter of milliseconds (hence the straight blue line in the figure), in comparison to the maximum total time of around 220 seconds in the worse case scenario within the 100 evaluated pages. The Figure also shows that components matching (highlighted in orange) is the most time-consuming process in computing the structural similarity score—taking around a minute at the median. In contrast, the components extraction process (highlighted in green) is relatively quick, with a maximum time of less than a minutes (i.e., around 49 seconds).

To better understand the relationship between the page complexity in terms of the number of components it contains and the timings metrics, we plotted these different timing metrics as a function of the number of components in each page, shown in Figure 13b. The results reveal that the relationship between QLUE's overall time and the number of components in the page follows a linear trend. Additionally, for the threshold filter timings (highlighted in blue),

(a) An example showing missing image thumbnails identified by both QLUE and the human evaluators, while missing menu elements identified by QLUE are overlooked by the human evaluators. Reported scores are 87% and 85% for human evaluators and QLUE, respectively.

(b) An example showing missing advertisements, all identified by both QLUE and the human evaluators. Additionally, QLUE recognized a missing text that is missed by the human evaluators. Reported scores are 91% and 64% for human evaluators and QLUE, respectively.

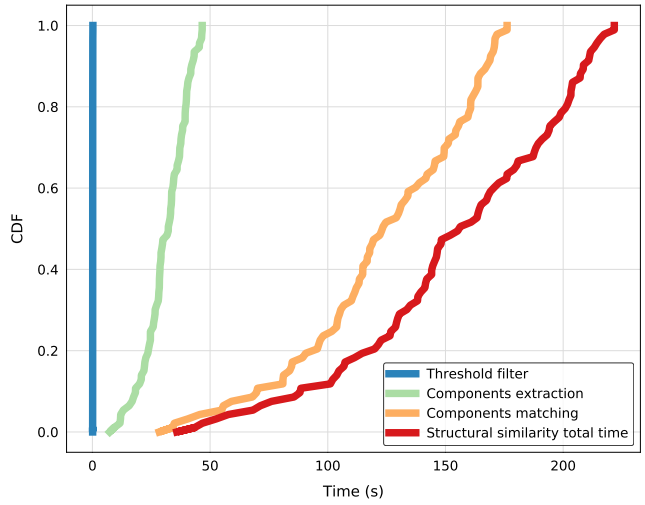


(c) An example showing missing navigation-bar elements identified by both QLUE and the human evaluators, and missing navigation elements at the end of the page that are overlooked by the human evaluators. Reported scores are 97% and 87% for human evaluators and QLUE, respectively.

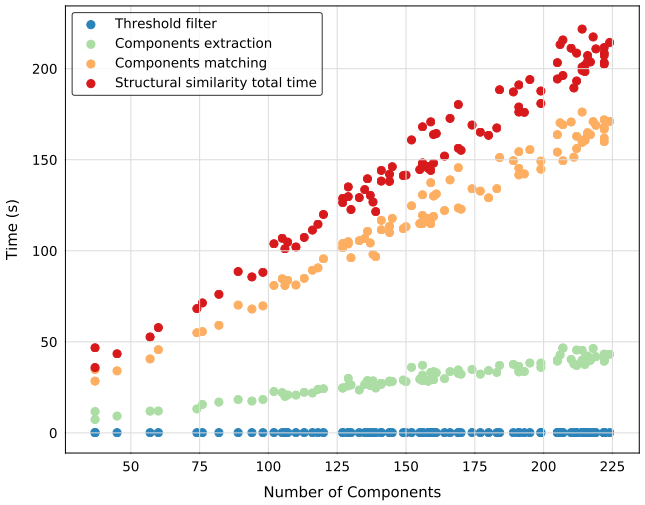
Figure 12: Three examples showing QLUE in action, illustrating how it perceives structural similarity in comparison to humans. The left side in (a) and (b) shows the reference page, whereas the right side shows the modified page. In (c), two screenshots are shown for both the reference and the modified page (to display the parts of BBC.com with missing elements). Missing elements recognized by humans (in each modified page) are highlighted in the corresponding reference page by the green rectangles. In contrast, missing elements recognized by QLUE in each modified page are highlighted by the red rectangles in the corresponding reference page. We relied on the comments provided by the human evaluators to identify the reported missing elements.

the results show that the total time required to perform the threshold filter is almost constant (i.e., pixels dilation), that is it does not depend on the number of components present in a page.

Similarly, we also evaluated the time complexity of the functional similarity in QLUE. Figure 14 shows the different time complexity results for each of the functional similarity

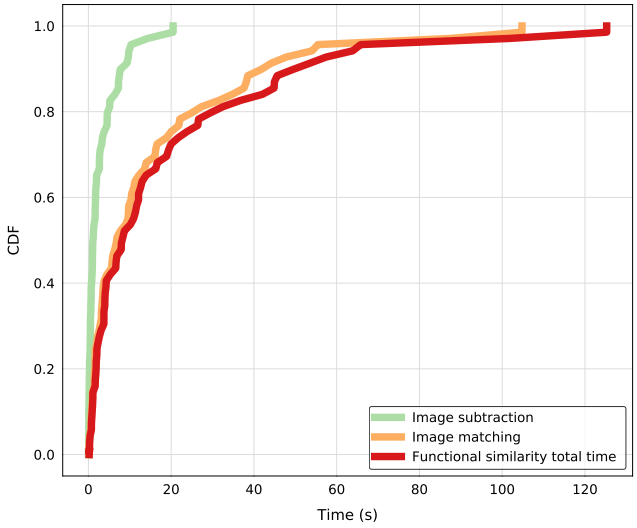


(a) QLUE processing times

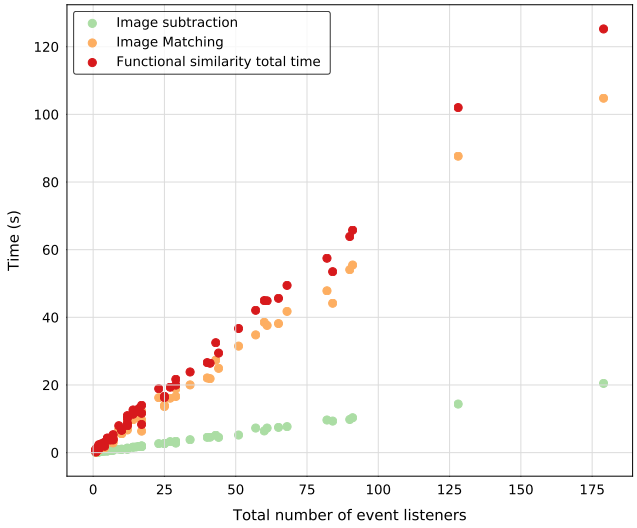


(b) QLUE processing times as a function of the number of components

Figure 13: Structural similarity time complexity evaluation



(a) QLUE processing times



(b) QLUE processing times as a function of the number of event listeners

Figure 14: Functional similarity time complexity evaluation

sub-processes, namely: image subtraction, and image matching. Figure 14a shows the CDFs of the time spent in each of the aforementioned sub-processes, in addition to the overall time to compute the functional similarity score. It can be observed that the *image subtraction* process (highlighted by the green curve) is significantly faster than the *image matching* (highlighted by the orange curve), where the *image subtraction* process completes its execution in less than a second at the median (with a maximum of 20 seconds in the worst-case scenario). In contrast, the *image matching* process has a median value of ~10 seconds, whereas in the worst-case scenario it consumes around 105 seconds. Next, Figure 14b shows the time spent in each sub-process as a function of the number of event listeners present in a given web page. The results show a linear relationship between the timing metrics and the number of event listeners.

7 DISCUSSIONS

This paper presented QLUE, a uniform approach to evaluate the quality of modern web pages using computer vision, to provide two different scoring metrics: content and functional similarity. We show that these metrics can be computed uniformly in a systematic way, without discriminating against any individual element, or increasing its weight. QLUE provides a novel solution to assess the preservation of the page functionality by implementing an *"interactivity bot"*. The bot is responsible for quantifying and triggering all of the page event listeners (similar to a human evaluator).

We envision that QLUE can be used in two different scenarios: a) uniformly comparing the quality of different web complexity solutions against each other as a standalone tool, b) as a built-in module within some of the aforementioned solutions that involve an iterative approach in simplifying web pages, such as the one used in WebMedic [30]. To seek an independent expert assessment of the latter usage scenario, we reached out to the WebMedic authors. The following subsection is a reflection of the discussion we had with one of the authors.

7.1 An Independent Expert Assessment on QLUE's Usage

Given the specificity of QLUE, and to independently assess its usage potential, we sought the opinion of experts in the field, mainly authors of existing state-of-the-art web complexity solutions. Here, we present WebMedic's authors [30] point of view on QLUE's usability, not only as a final scoring mechanism of the overall page quality, but also as part of WebMedic's internal algorithm—to be used for measuring the appearance metric of their page utility, instead of relying on the simple *pHash* algorithm. We shared the conceptual and technical details of QLUE with the authors offline and then conducted an interview over zoom with one of them. Below is a record of the interview transcript.

Question: *Do you see the value of using QLUE as a module within the WebMedic framework?*

WebMedic author: *"Given that QLUE can accurately identify the user-centric importance of web page components, and by extension, the JavaScript that interacts with the given components, QLUE can be integrated into WebMedic as system module to automatically identify the key JavaScript functions without the need of conducting user studies. As web pages frequently evolve their content over time, it can be challenging to generalize the results of user-studies from one version to another, and QLUE can fill the gap here by providing an alternative to user-studies that unifies the scoring approach."*

Question: *To what extent do you think that QLUE can speedup simplifying web pages using WebMedic?*

WebMedic author: *"The previous version of WebMedic relied on brute-force exploration through different versions of a web page to generate memory/utility weights. If we had the same framework, then yes I do see some value. It can also help in speeding up the computations, since QLUE can replace the measure of functionality/appearance change for this version. We are moving away from the brute-force approach in the next version and rely on a single run to profile memory/utility of the web page. I see QLUE as the target metric that can help in judging how good a certain cut was (especially from a user-perspective). Though the value of using QLUE for generating memory/utility profiles for the JS functions is not quite clear to me, it definitely has value for translating the utility impact to a user-perspective number."*

Question: *How would the functional comparison in QLUE improve WebMedic accuracy in avoiding page breakage?*

WebMedic author: *"There are two cases for page breakage: a) a JavaScript function accessing some non-existent state, e.g., $foo()$ defines array, $bar()$ accesses the array, and if we cut $foo()$ without cutting $bar()$ then the array accessed in $bar()$ no longer exists, or b) an event listener attached to an element (e.g., button) gets removed, thereby breaking the button. We currently instrument JavaScript to track every event listener added to the web page and can thereby track if an event listener is missing. In this case QLUE can help, since WebMedic only checks if the event listener exists or not. My only concern is that it might be very time consuming to run for too many pages, due to the fact that WebMedic creates JavaScript cuts at functional level, and given that a web page might have a high number of functions (i.e., 1000 functions or above). If we make a different version of the web page for every missing function, we'll have a large number of page variants. It is definitely useful but maybe beyond the initial operation of WebMedic, given that we have to test on permutations of all the JavaScript functions. However, we are on the evaluation phase of WebMedic after creating the candidate pages, QLUE's functional comparison will be very helpful, given that QLUE takes it a step further from just checking if the event listeners exist or not."*

7.2 Limitations

Here, we show two three corner cases that represent the current limitations of QLUE. The first case occurs when a web page displays different images upon re-load. For instance, a web page that utilizes an image slider component may display a different image every time the page is loaded (either in the next visit or when the page is refreshed). In this case, there is a high chance that the screenshot captured by QLUE for the modified page would have a different image in the slider component than the image shown on the corresponding

reference page. While this difference should not penalize the structural similarity score, the current implementation of QLUE does not recognize such cases, hence, the final score will be unnecessarily impacted. An extended version of QLUE can overcome this limitation by predicting components with changeable content, such as image sliders and advertising containers, where QLUE can take multiple screenshots of the page in order to collect all possible images.

The second corner case that QLUE does not automatically handle is a web page that uses a floating banner which always appears as the user scrolls. This poses a challenge in the screenshot generation process because the banner would appear multiple times in the full screenshot captured for the page, unless the user fixes the banner location to appear only at the top of the page. This can manually be handled by checking for such case and modifying the CSS styling of these banners before proceeding with the screenshot generation. In our future work, we plan to extend QLUE screenshot generation to automatically detect such floating elements and modifying their CSS styling accordingly.

QLUE is capable of evaluating search bars functionality in web pages, by filling the search bar with a search query and triggering the search event. Given that the page would return a valid visual response to the search query that QLUE can compare between the two versions of the page (i.e., the modified and the reference page). However, general web forms, although similar in spirit to search bars, are not handled by the current implementation of QLUE's functional comparison. The reason behind this is the fact that most of the responses triggered by submitting a form do not necessarily reveal whether the filled data were properly sent to the server or not (apart from a simple thank you message that is usually displayed as a default response).

8 CONCLUSION

In this paper, we presented QLUE, a tool that aims to provide a unified approach for performing qualitative evaluations of web pages using computer vision. A user study of 30 participants has shown that QLUE computes comparable similarity scores to those provided by humans, and effectively assesses the retainment of web pages functionality. Additionally, an interview with the authors of one of the web complexity solutions served as a usage scenario highlighting the usability and benefits of QLUE as a qualitative evaluation tool in today's web.

9 ACKNOWLEDGMENTS

We would like to thank Usama Naseer and the rest of the WebMedic authors for their valuable feedback and suggestions serving as independent expert evaluators for QLUE usability.

REFERENCES

[1] [n.d.]. Opera Mini for Android. https://www.opera.com/mobile/mini. Accessed: 2021-01-11.
[2] 2018. SpeedReader: Fast and Private Reader Mode for the Web. https://brave.com/speed-reader/. Accessed: 2021-02-15.
[3] 2019. The age of digital interdependence. https://digitalcooperation.org/wp-content/uploads/2019/06/DigitalCooperation-report-web-FINAL-1.pdf. Accessed: 2020-10-04.
[4] 2020. Brave: the Privacy Preserving Browser. https://brave.com/. Accessed: 2021-01-12.
[5] 2020. Lighthouse. https://developers.google.com/web/tools/lighthouse. Accessed: 2020-03-26.
[6] 2020. The top 500 sites on the web. https://www.alexa.com/topsites Accessed: 2020-01-03.
[7] Zainul Abi Din, Panagiotis Tigas, Samuel T King, and Benjamin Livshits. 2020. PERCIVAL: Making in-browser perceptual ad blocking practical with deep learning. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 387–400.
[8] AdBlock. 2009. Surf the web without annoying pop ups and ads. https://getadblock.com/. Accessed: 2020-05-02.
[9] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. 2015. Flywheel: Google's data compression proxy for the mobile web. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 367–380.
[10] Mark Bauman and Ray Bonander. 2017. Advertisement blocker circumvention system. US Patent App. 15/166,217.
[11] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).
[12] 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.
[13] Matthew Conlen and Jeffrey Heer. 2018. Idyll: A markup language for authoring and publishing interactive articles on the web. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 977–989.
[14] Franklin C Crow. 1984. Summed-area tables for texture mapping. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. 207–212.
[15] Sybu Data. 2016. Sybu JavaScript Blocker – Google Chrome Extension. https://sybu.co.za/wp/projects/js-blocker/. Accessed: 2020-05-02.
[16] Google Developers. 2019. Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools. Accessed: 2020-05-01.
[17] Facebook. 2015. Instant Articles | Facebook. https://instantarticles.fb.com/ Accessed: 2020-03-21.
[18] 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.
[19] Google. 2019. AMP is a web component framework to easily create user-first web experiences - amp.dev. https://amp.dev. Accessed: 2019-05-05.
[20] Google. 2021. AMP Reporting guide. https://support.google.com/analytics/answer/9264222?hl=en. Accessed: 2021-04-04.
[21] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'10, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362.

https://doi.org/10.1038/s41586-020-2649-2

[22] Joshua Hibschman and Haoqi Zhang. 2015. Unravel: Rapid web application reverse engineering via interaction recording, source tracing, and library detection. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 270–279.

[23] Jason Huggins. 2019. Selenium WebDriver. Browser Automation. https://www.seleniumhq.org/projects/webdriver/. Accessed: 2019-05-14.

[24] Cliqz International. 2009. Ghostery Makes the Web Cleaner, Faster and Safer. https://www.ghostery.com/. Accessed: 2020-05-2.

[25] Conor Kelton, Matteo Varvello, Andrius Aucinas, and Benjamin Livshits. 2021. Browselite: A Private Data Saving Solution for the Web. *arXiv preprint arXiv:2102.07864* (2021).

[26] Jonathan Lee, Tobias Lidskog, and Peter Hedenskog. 2020. Browsertime. https://www.sitespeed.io/documentation/browsertime/introduction/. Accessed: 2020-02-6.

[27] Sarah Lim, Joshua Hibschman, Haoqi Zhang, and Eleanor O'Rourke. 2018. Ply: A visual web inspector for learning from professional webpages. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 991–1002.

[28] Sarah Lim, Joshua Hibschman, Haoqi Zhang, and Eleanor O'Rourke. 2018. Ply: A visual web inspector for learning from professional webpages. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 991–1002.

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

[30] Usama Naseer, Theophilus A Benson, and Ravi Netravali. 2021. WebMedic: Disentangling the Memory-Functionality Tension for the Next Billion Mobile Web Users. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*. 71–77.

[31] 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

[32] Ravi Netravali and James Mickens. 2018. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 249–266. https://www.usenix.org/conference/nsdi18/presentation/netravali-prophecy

[33] Travis Roman. 2018. JS Blocker. https://jsblocker.toggleable.com/. Accessed: 2020-05-02.

[34] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V Madhyastha. 2017. Vroom: Accelerating the mobile web with server-aided dependency resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 390–403.

[35] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. 2014. scikit-image: image processing in Python. *PeerJ* 2 (2014), e453.

[36] 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

[37] 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

[38] Xiong Zhang and Philip J Guo. 2018. Fusion: Opportunistic web prototyping with ui mashups. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 951–962.

[39] Sharon Zhou, Mitchell L Gordon, Ranjay Krishna, Austin Narcomey, Li Fei-Fei, and Michael S Bernstein. 2019. Hype: A benchmark for human eye perceptual evaluation of generative models. *arXiv preprint arXiv:1904.01121* (2019).