Near-Duplicate Detection in Web App Model Inference

Rahulkrishna Yandrapally
University of British Columbia
Vancouver, BC, Canada
rahulky@ece.ubc.ca

Andrea Stocco
University of British Columbia
Vancouver, BC, Canada
astocco@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

ABSTRACT

Automated web testing techniques infer models from a given web app, which are used for test generation. From a testing viewpoint, such an inferred model should contain the minimal set of states that are distinct, yet, adequately cover the app's main functionalities. In practice, models inferred automatically are affected by near-duplicates, i.e., replicas of the same functional webpage differing only by small insignificant changes. We present the first study of near-duplicate detection algorithms used in within app model inference. We first characterize functional near-duplicates by classifying a random sample of state-pairs, from 493*k* pairs of webpages obtained from over 6,000 websites, into three categories, namely clone, near-duplicate, and distinct. We systematically compute thresholds that define the boundaries of these categories for each detection technique. We then use these thresholds to evaluate 10 near-duplicate detection techniques from three different domains, namely, information retrieval, web testing, and computer vision on nine open-source web apps. Our study highlights the challenges posed in automatically inferring a model for any given web app. Our findings show that even with the best thresholds, no algorithm is able to accurately detect all functional near-duplicates within apps, without sacrificing coverage.

ACM Reference Format:
Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. 2020. Near-Duplicate Detection in Web App Model Inference . In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380416

1 INTRODUCTION

Automated techniques such as web app crawlers are widely used to reverse-engineer state-based models as a viable vehicle for various analysis and testing tasks such as automated test generation. The *state* in such models represents the dynamic webpage of the app, as represented by the Document Object Model (DOM) in the browser. Crawlers are capable of efficiently exploring a large state space of any given web app. However, an adequate model should contain only the minimal set of *distinct* states that represent the web app functionalities, while discarding insignificant states that do not contribute to exposing new functionality to the end user. Instances of such states are pages only differing by small cosmetic changes, which are also referred to as *near-duplicates* in the literature [23, 24, 29, 34]. To discard such near-duplicate webpages, crawlers have adopted state abstraction functions over the DOM [26, 36, 37, 45] as a proxy for the similarity of webpages. The downside of these abstractions is that minimal changes to the DOM can result in duplicate states in the model, even if such DOM changes are not reflected on the final UI visually, and therefore might not be representative of a new webpage functionality. From an end-to-end (E2E) testing perspective, clone and near-duplicate states in web app models negatively impact their accuracy and completeness, undermining the quality of the test suites generated from such models in terms of size, runtime, and coverage.

Clone and near-duplicate detection *across* different web apps has been an active research topic in many fields [23, 24, 29, 34]. In information retrieval, the content of a webpage has been the primary focus, because the purpose of web search engines is to index and retrieve information from webpages through search queries. Computer vision techniques have been employed to detect visually similar webpages, for instance in phishing detection [2, 21]. Other approaches leverage state abstractions based on the similarity of URLs, textual content and the DOM [17, 44, 53]. Detecting near-duplicate pages is a challenging problem as there is no generally accepted definition of near-duplicate states and there is no unified standard against which a technique can be assessed [28, 29]. A second challenge pertains to the selection of similarity *thresholds* that such techniques need as input to determine when two pages are similar. These thresholds are usually educated guesses, as no systematic means have been proposed so far to estimate them automatically.

In this work, we are interested in detecting distinct states in web app models in the context of functional E2E web testing. Our aim is to study the nature of duplicate states occurring *within* a web app, and provide a systematic approach to selecting thresholds for inferring an optimal model, i.e., having the lowest number of (near-)duplicate states. To this end, we evaluate the capability of 10 near-duplicate detection algorithms in identifying clone, near-duplicate, and distinct web app states. We adopt techniques from three different domains—information retrieval, web testing, and computer vision—where the textual content, the DOM tree, and the visual screenshot of the page are used to measure the similarity between states. Our goal is to assess whether textual, structural, or visual features are related with semantic properties of webpages and provide meaningful means to understanding their degree of functional relatedness from an E2E testing perspective.

To select the similarity thresholds for fine-tuning such techniques, we first crawled 6*k* websites randomly selected from Alexa's top million URLs. We retrieved 493*k* pairs of states belonging to the same application, and computed the similarity distance between

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380416

these pairs using each near-duplicate algorithm. We then manually classified $1k$ random state-pairs into three categories of clone, near-duplicate, or distinct. We used our empirical data of distances to choose thresholds for each algorithm through statistical and optimization search methods. We evaluated their accuracy in automatically classifying clones and near-duplicates in the remaining unlabelled portion of the dataset. Further, we evaluated these configured algorithms on a subject set of nine unseen web apps, for which manual ground truth models were previously created.

Our work makes the following novel contributions:

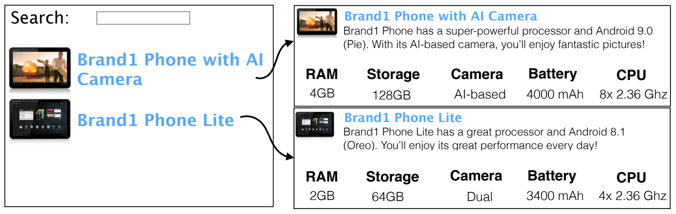
- The first study of 10 different near-duplicate detection techniques applied in the context of web app model inference.
- A classification of different categories of near-duplicates occurring within a web app.
- Systematic ways of threshold selection for near-duplicate detection as well as an empirical evaluation of their effectiveness in test models.
- The toolset comprising the 10 near-duplicate detection algorithms, which is available for download [5].
- A dataset of $99k$ manually classified pairs of webpages, of which (1) $1{,}5k$ pairs are randomly sampled from 6k websites, and (2) $97.5k$ from nine real-size web apps. Our dataset can be used by others to conduct similar near-duplicate detection studies and is also available for download publicly [5].

Our results show that even with the best thresholds, no algorithm is able to accurately detect all functional near-duplicates within apps. In practice, existing near-duplicate detection techniques are *not* designed to find functional similarity in a way that human testers regularly assess while testing web apps. For certain types of near-duplicates, we observed that the model deteriorates over time as the crawl progresses. For instance, although RTED was able to achieve a high accuracy F_1 score of 0.95 initially, the final produced model had only an F_1 of 0.45. This deterioration is due to the accumulation of numerous near-duplicates to the model, which decreases precision. Our results underline the need for further research in devising techniques geared specifically toward web test models, i.e., that can distinguish between different types of near-duplicates such as those found in our study.

2 REDUNDANCIES IN WEB APP MODELS

In practice, web testing is often performed in an end-to-end (E2E) fashion, by verifying the correctness of the web app state in response to user events and interactions with the GUI (e.g., clicks, and forms submissions). This task is performed either manually by testers, or by writing test scripts with test automation tools such as Selenium [46].

Automated techniques, on the other hand, generate web test cases from models that are inferred through reverse-engineering techniques. A popular method to model construction for modern web apps is automated state exploration, also known as web app crawling [35, 54]. Such techniques dynamically analyze the web app under test by automatically firing events and checking the webpage for changes. When new state changes are detected, the model is updated to reflect the event causing the new state. Generated models can be represented in various formats such as UML state diagrams, Finite State Machines (FSM), or State-Flow Graphs (SFG) [35, 42, 54].



Brand1 Phone has a super-powerful processor and Android 9.0 (Pie). With its AI-based camera, you'll enjoy fantastic pictures!

RAM	Storage	Camera	Battery	CPU
4GB	128GB	AI-based	4000 mAh	8x 2.36 Ghz

Brand1 Phone Lite has a great processor and Android 8.1 (Oreo). You'll enjoy its great performance every day!

RAM	Storage	Camera	Battery	CPU
2GB	64GB	Dual	3400 mAh	4x 2.36 Ghz

Figure 1: Example of near-duplicate web pages.

To avoid redundancies in the model, states that are identical or highly similar to previously encountered states should be discarded. For instance, let us consider Figure 1, a web app in which the homepage shows a list of phones. When the user clicks on any of the phones in that list, she is redirected to another web page displaying the detailed characteristics of the selected phone. From a functional testing viewpoint, however, a page containing a list of 20 phones is *conceptually* the same as one listing the same 20 phones plus one extra phone.

The problem of detecting already visited states can be cast as an *equivalence problem*: given two web page states p_1 and p_2 explored by the crawler, a state abstraction function determines whether $p_1 \simeq p_2$. More formally:

Definition 1 (State Abstraction Function). *A state abstraction function (SAF) $\mathcal{A}$ is a pair $(dist, t)$, where $dist$ is a similarity function, and t is a threshold defined over the values of $dist$. Given two web pages, p_1, p_2, $\mathcal{A}$ determines whether the distance between p_1 and p_2 falls below t.*

$$\mathcal{A}(dist, p_1, p_2, t) \begin{cases} true & : dist(p_1, p_2) < t \\ false & : otherwise \end{cases}$$

In practice, $\mathcal{A}$ is defined based on the similarity of some abstracted notion of the web pages such as their URLs, textual content, DOM structure, or screenshot image. However, the amount and nature of changes occurring in a web page with respect to the functionality of the app is not always directly proportional to the amount of changes in the DOM tree, textual content, or visual aspects of the page.

Let us consider using a crawler equipped with a SAF based on DOM content similarity on our sample web app of Figure 1. This SAF is less tolerant to content (textual) changes occurring in web pages. Therefore, each page displaying a new phone's characteristics might be considered a different state and many functionally similar occurrences of already modelled pages (i.e., near-duplicates) would be included in the model. If we use this *inflated model* to generate test cases, the overall functional coverage does not change when the generated tests exercise the phone details page multiple times, thus potentially wasting precious testing time and resources.

On the other hand, another "better" SAF, for instance based on the DOM tree similarity with a proper threshold value, would consider all such phone detail pages as the same, providing a more concise model for the web application of our example. However, a high threshold value might cause other relevant functionality to be abstracted away as well, resulting in an incomplete model.

Table 1: Near-Duplicate detection algorithms included in our study.

Domain	Algorithm	Input	Description	Distance Output
Information Retrieval				
Web Search	simhash [20]	DOM (content)	64 bit fingerprinting technique which uses features extracted from the web page content	Hamming distance of two 64 bit digests
Malware detection	TLSH [38]	DOM (content)	Locality-sensitive 256 bit hashing scheme that is robust to minor variations of the input	Hamming distance of two 256 bit digests
Web Testing				
	RTED [37, 39]	DOM (Tree)	Minimum-cost sequence of node edit operations that transform one DOM tree into another	Tree edit distance value normalized by the sum of nodes in the two trees
	Levenshtein [30, 36]	DOM (String)	Minimum number of single-character edits required to transform one string into another	Edit distance value normalized by the sum of the string lengths
	String Equality (baseline)	DOM (String)	String equality comparison	Boolean value
Computer Vision				
Image Hashing	PHash [59]	Screenshot	128 bit perceptual hash that represent the lowest frequencies of pixel brightness, to which discrete cosine transform (DCT) is applied to retrieve a brightness matrix	Hamming distance of two 128 bit digests
	Block-mean [57]	Screenshot	256 bit perceptual hash obtained by dividing the image into non-overlapping blocks, which are encrypted with a secret key and normalized median value is calculated	Hamming distance of two 256 bit digests
Whole Image Comparison	Histogram [52]	Screenshot	Color distribution of a digital image	χ^2 distance between two color histograms
	PDiff [58]	Screenshot	Adopts a human-like concept of similarity that uses spatial, luminance, and color sensitivity	Number of different pixels normalized by the maximum number of pixels in the two images
Structural Similarity	SSIM [4]	Screenshot	Simulates the high sensitivity of human visual system to structural distortions while compensating for non-structural distortions	Normalized structural distortion value
Feature Detection	SIFT [32]	Screenshot	Computes local feature vectors and image descriptors which are invariant to geometric affine transformations like scaling/ rotation	Number of different key-points normalized by the maximum number of key-points in both images

Near-duplicate detection techniques have been studied for reducing the occurrence of redundant similar pages *across* web apps, e.g., in web search engines [20] or phishing detection [38]. An understanding of whether such techniques apply also in detecting functional near-duplicates *within* the same web app is missing in the literature. Despite its prevalence and importance, this problem is understudied, because it is hard to define a notion of equivalence for two arbitrary webpages. Moreover, in the general case, deciding *a priori* which abstraction function and which threshold would work best for a given web app is a challenging task, as it requires substantial domain-specific knowledge of the web app under test.

3 NEAR-DUPLICATE ALGORITHMS

In this work, we study 10 near-duplicate detection algorithms from three different domains, namely, information retrieval, web testing, and computer vision. Table 1 presents the techniques, along with the domain they belong to, the input types, a short description, and their distance output.

3.1 Information Retrieval

Near-duplicate detection has been applied to index the massive volume of web pages continuously retrieved by web crawlers for search engines. The overall goal is to select only a relevant set of pages based on the provided user search string. In this setting, performance is the most important factor, therefore hashing mechanisms have been adopted due to their design simplicity and speed of comparison. As an input, the web page content is typically the primary focus when designing algorithms used in this domain.

We chose two content hashing algorithms from this domain: (1) `simhash` [20], a popular and effective web page fingerprinting technique adopted by Google to index web pages [29], and (2) Trend Locality Sensitive Hash (`TLSH`) [38], a hashing technique for fingerprinting source code, employed for malware detection [55].

3.2 Web Testing

In the web testing domain, researchers have studied DOM-based abstractions to compare webpages during the crawling of the application under test. The assumption is that two web pages sharing similarities among their DOMs are likely to represent pages having analogous functionalities, hence it is worthwhile to consider them the same. The DOM can be treated either as a tree-like structure, or as a simple string of characters.

We chose three different similarity algorithms over the DOM that have been employed as state abstraction functions in prior web testing research [36, 37, 49]: (1) tree edit distance with the RTED algorithm [39], (2) Levenshtein distance [30] over the string represented by the DOM, and (3) string equality between two DOM strings, which we use as baseline.

3.3 Computer Vision

Image similarity is one of the main topics in computer vision. Many techniques have been proposed and studied, at different levels of granularity, ranging from low-level pixel matching up to high-level feature-based matching. These techniques are applied in indexing and searching, summarization, object detection and tracking, facial recognition, and also copyright image detection. We consider different classes of image-based algorithms.

Image hashing techniques map visually identical or nearly-identical images to the same (or similar) digest called image hash. We chose two image hashing algorithms: (1) block-mean hash [57] and (2) perceptual hash (PHash) [59], which have been used in multimedia security for image retrieval, authentication, indexing and copy detection. *Whole image matching* techniques focus instead on individual pixels composing the image. Color-Histogram [43] and Perceptual Diff (PDiff) [58] have been successfully applied in previous web testing work for detecting cross-browser incompatibilities [33]. A downside of those techniques is that they are affected by changes in coordinates of web elements common in responsive web layouts. *Structural similarity* techniques quantify image quality degradation. For instance, Structural Similarity Index (SSIM) [4] has been shown to be effective due to the highly structured nature of web apps' GUIs [21]. Lastly, *feature detection* techniques have been widely employed for near-duplicate image detection. For instance, Scale Invariant Feature Transform (SIFT) [32] has been applied to aid web test repair [51] and phishing detection [2].

To the best of our knowledge, this work is the first to consider and evaluate visual image similarity as a near-duplicate detection technique for web application crawling.

4 EMPIRICAL STUDY DESIGN

The goal of our study is to determine how existing near-duplicate detection techniques can be employed to obtain an optimal model of a web application that can be used for E2E testing.

RQ$_1$: *What type of functional near-duplicates exist within apps?*
RQ$_2$: *How well can functional near-duplicates be detected?*
RQ$_3$: *What is the impact of near-duplicates and detection techniques in inferring a web-app model?*

First, in Section 5, we randomly sample 1,000 within-app state-pairs from a dataset created by crawling 6*k* randomly selected URLs. We characterize the changes occurring between states within an app and identify how they lead to different classes of functional near-duplicates (RQ$_1$). We label these 1,000 pairs as either clones, near-duplicates or distinct states, and compute the distance between them for all the ten near duplicate techniques described in Section 3.

In Section 6, using these labelled pairs, we compute statistical and optimal thresholds to fine-tune each near duplicate technique. Through this, we aim to determine whether such randomly sampled distances from a large dataset can be used to automatically classify state-pairs in unseen web apps and detect near-duplicates (RQ$_2$).

In Section 7, we determine the best near-duplicate detection techniques and application-specific thresholds to infer web app models for nine open-source web apps covering the different near-duplicate categories. Finally, we analyze these models to determine how different kinds of near-duplicates impact model inference (RQ$_3$).

5 RQ$_1$: NEAR-DUPLICATES IN WEB APPS

In order to determine what kinds of functional near-duplicates occur within apps, we first create a dataset of *within app state-pairs* and their calculated distances for each near-duplicate detection algorithm. Then, we manually characterize the nature of differences between pairs of pages and classify them in a random sample.

5.1 Dataset Creation

First, we crawl randomly selected website URLs from the top one million as provided by Alexa,[1] a popular website that ranks sites based on their global popularity for a *week* using CRAWLJAX [36], an event-driven crawler for exploring highly dynamic web apps. We configured CRAWLJAX to run using the Chrome browser, with its default simple state abstraction function, namely string equality (see Table 1), and a runtime limit of five minutes for each crawl.

To account for network communication errors and the tool's exploration limitations, e.g., on sites that require login credentials, we filtered out sites for which the crawl models obtained contained less than 10 states. After this filtering stage, we retained 1,064 different sites accounting for 30,202 states from the original 6,359 web crawls. We then created all possible 677,415 pair-wise combinations of states *within each crawl*, which we call *state-pairs*.

5.1.1 Computing Distances. We computed the distance for each state-pair using each of the 10 algorithms presented in Table 1. We discarded the state-pairs for which the distance could not be computed correctly, such as the case of DOM-based tree edit distance of malformed HTML trees. The final dataset, called $\mathcal{DS}$, contained 1,031 sites and 29,704 states, from which 493,088 state-pairs with properly computed distances were obtained.

5.1.2 Distance Normalization. The raw distances which quantify the difference between two given pages have different output spaces based on the page characteristic used by the technique. As an example, given a state-pair of web pages, PDiff outputs the number of perceptually different pixels between their screenshots, whereas BlockHash returns the hamming distance between image hashes. For the sake of comprehensibility, we normalized all distances computed by each algorithm, as described in the *Distance Output* column of Table 1, but we never compare outputs of different techniques.

5.2 Classification of Changes

To gain a better understanding of what changes within web pages characterize near-duplicates, we classify the differences of the state-pairs in our dataset from the point of view of a human tester who is interested in functionality coverage.

5.2.1 Procedure. Manually examining state-pairs is a time consuming task requiring familiarity with the functionality of the application. Therefore, we randomly sampled a set, called $\mathcal{RS}$, of 1,000 state-pairs from our final dataset of 493,088 state-pairs, which allows us to have a confidence level of 99% with a 4% margin of error in deriving a representative statistic. For each state-pair $(p_i, p_j) \in \mathcal{S}$, the authors of the paper visually analyzed, in isolation, the screenshot images (and the original web pages where necessary) of the two web app states from a functional testing perspective, to obtain a set $\mathcal{D}$ of differences. Each difference in $\mathcal{D}$ is defined as $\Delta(p_i, p_j) = \{\delta(e_i, e_j)\}$ where $\delta(e_i, e_j)$ is a pair of non-identical web elements in which $e_i \in p_i$ and $e_j \in p_j$. Finally, each author assigned a descriptive label to each detected difference.

5.2.2 Difference Categorization. After enumerating all differences across the 1,000 state-pairs in $\mathcal{RS}$, the authors reviewed them together to resolve conflicts and reached consensus on equivalence classes of differences. Our study revealed the following categories.

Definition 2 (Unrelated (U)). Given a difference $\delta(e_i, e_j)$, neither of e_i or e_j are related to any functionality offered by the web app.

[1] http://www.alexa.com

Examples of these differences include changes in background images, or GUI widgets related to advertisement (see red ovals in Figure 2a).

Definition 3 (Duplicated (D)). Given a difference $\delta(e_i, e_j)$, e_i and e_j replace each other in the original pages p_i and p_j without adding any new functionality to either page.

Two distinct subcategories of duplicated differences emerged:

- *Replacement* (D_1): $D_1 : e_i \equiv e_j$ meaning the difference represents a functionality or content that is equivalent. For instance, in Figure 2b, the red ovals highlight equivalent content.
- *Addition* (D_2): $D_2 : e_i = \emptyset \wedge \exists e_i' \in p_i : e_i' = e_j \vee \delta(e_i', e_j) \models D_1$ meaning the non-empty e_x in δ has a duplicate e_y in the same page, and therefore its addition does not affect the overall functionality of the page. For example, in Figure 2c, the oval identifies a duplication of an existing functionality.

Definition 4 (New (N)). Given a difference $\delta(e_i, e_j)$, δ represents a new functionality or a semantically different content, i.e.: $\delta(e_i, e_j) \models N : (e_i = \emptyset) \wedge (\nexists e_i' \in p_1 s.t. e_i' = e_j \vee \delta(e_i', e_j) \models D)$.

For example, the search box in Figure 1 is absent in phone description pages and is an example of new functionality.

5.2.3 State-Pair Classification. Following the classification of differences described above, we classified state-pairs from a functional point of view, in three distinct categories defined as follows.

Definition 5 (Functional Clone (Cl)). Given two web pages p_1 and p_2, the state-pair (p_1,p_2) is a functional clone (Cl) if there are no semantic, functional or perceptible differences between them, defined as $Cl : \Delta(p_1, p_2) = \emptyset$.

Definition 6 (Functional Distinct (Di)). Given two web pages p_1 and p_2, p_1 is functionally distinct from p_2 if there is any semantic or functional difference between the two pages, $Di : \exists\delta(e_i, e_2) \models N$.

Definition 7 (Functional Near-Duplicate (Nd)). Given two web pages p_1 and p_2, p_1 is a functional near-duplicate of p_2 if the changes between the states do not change the overall functionality being exposed: $Nd : \Delta \not\models Cl \wedge \nexists(\delta(e_1, e_2) \models N) \in \Delta$.

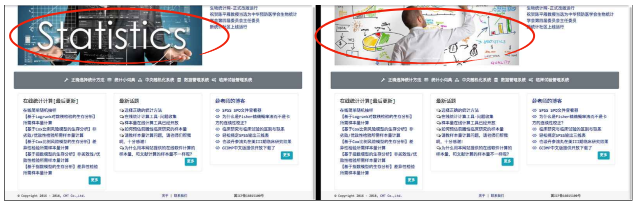
We further observed three fine-grained subclasses of near-duplicates in our dataset.

Cosmetic (Nd_1) when changes related to the aesthetics of the webpage such as advertisements or background images occur, which leave the functionalities unaltered (see Figure 2a): $Nd_1 : \Delta(p_1, p_2) \ni \delta(e_1, e_2) \models U$

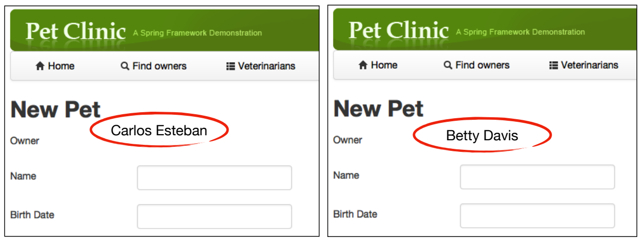
Dynamic data (Nd_2) when both states of the pair are generated from the same template and populated with dynamic data, according to a user query or app business logic (see Figure 2b): $Nd_2 : \Delta(p_1, p_2) \ni \delta(e_1, e_2) \models D_1 \vee U$

Duplication (Nd_3) when there are additional web elements in a page the functionality and semantics of content of which is entirely represented within the other page (see Figure 2c): $Nd_3 : \exists\delta(e_1, e_2) \models D_2 \in \Delta(p_1, p_2)$

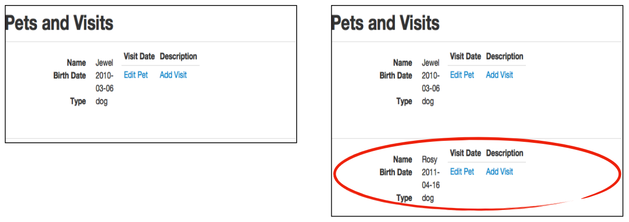
Following these definitions, we manually labelled the 1,000 state-pairs in $\mathcal{RS}$, and found 441 clones, 275 near-duplicates (45 Nd_1, 219 Nd_2, 11 Nd_3), and 284 distinct pairs.



(a) *Near-Duplicate (Nd_1): Background Image Changes*



(b) *Near-Duplicate (Nd_2): Dynamic Data*



(c) *Near-Duplicate (Nd_3): Duplicated Functionality*

Figure 2: Different subclasses of near-duplicate state-pairs.

6 RQ$_2$: CLASSIFICATION OF STATE-PAIRS

6.1 Subject Systems

To address RQ$_2$ (and later RQ$_3$), we need to infer models with different algorithms and thresholds numerous times, which requires web apps with deterministic behaviours. To this aim, we selected nine open-source web apps (Table 2) used in previous research of web testing [15, 16, 49, 50], as *subjects*: Claroline (*v. 1.11.5*) [7], Addressbook (*v. 8.2.5*) [40], PPMA (*v. 0.6.0*) [11], MRBS (*v. 1.4.9*) [12] and MantisBT (*v. 1.1.8*) [13] are open-source PHP-based applications while Dimeshift (*commit 261166d*) [8], Pagekit (*v. 1.0.16*) [9], Phoenix (*v. 1.1.0*) [10] and PetClinic (*commit 6010d5*) [6] are web apps that cover popular JavaScript frameworks *Backbone.js*, *Vue.js*, *Phoenix/React* and *AngularJS*, respectively.

6.2 Manual Classification (Ground Truth)

We ought to create manually labelled models for each subject, which we can use as ground truths for comparison of techniques.

First, we use Crawljax to create a master crawl model with default depth-first exploration strategy, default state abstraction function based on DOM string equality, and a maximum time budget of one hour, which allow us to capture a large portion of each app's state space.

Table 2: Subject Set with Manual Classification

	Bins	States	Pairs	Clones	Near-Duplicates			Distinct
					Nd_2	Nd_3	*Total*	
Addressbook	25	131	8,515	26	52	2,295	2,347	6,142
PetClinic	14	149	11,175	2	1,433	180	1,613	9,411
Claroline	36	189	17,766	2,707	71	0	71	14,988
Dimeshift	21	153	11,628	375	570	0	570	10,683
PageKit	20	140	9,730	0	904	3,044	3,948	5,782
Phoenix	10	150	11,175	1	25	4,580	4,605	6,569
PPMA	23	99	4851	64	467	0	467	4,320
MRBS	14	151	11,325	27	4,044	0	4,044	7,254
MantisBT	53	151	11,325	2	1,117	0	1,17	10,206
Total	216	1,313	97,490	3,204	8,683	10,099	18,782	75,355

Table 3: Average webpage characteristics state (DOM and Screenshot) across the two datasets

	DOM			Image
	Tree (# nodes)	**Source** (length)	**Content** (length)	**Pixels** (#)
Dataset ($\mathcal{DS}$)	810	105,445	45,575	3,575,837
Subjects ($\mathcal{SS}$)	290	17,655	6,216	1,190,230

Next, we created state-pairs from the states in each model, as follows. The authors of this paper *manually classified* each state-pair into a clone, near-duplicate (with subcategories) or distinct category, following the same procedure described in Section 5.2. In addition, we also assigned each state to a *bin* that represents a part of the application's state space devoted to a certain functionality. As such, each bin is a logical container for all dynamically generated concrete webpages upon crawling (e.g., all webpages related to *login*). We consider the first concrete instance of a bin B to be a *coverage of* B by that crawl model. Additional concrete instances of a bin are considered clones or near-duplicates of the bin B.

Table 2 shows the master crawl characteristics for each web app as well as our classification outcome. In the rest of the paper, we refer to the nine master crawls with manually classified $97.5k$ state-pairs of the nine apps as *subject set* ($\mathcal{SS}$), and to our manual classification and identified bins as *ground truth*. Our classification of the subject-set did not find any near-duplicates of category Nd_1 in $\mathcal{SS}$ as the subjects did not feature unrelated changes (U) such as advertisements, commonly found in other kind of websites. MantisBT has the most bins (53), representing a state-space five times bigger than that of Phoenix, which is the smallest number of bins (10). Addressbook, PageKit and Phoenix have a high number of near-duplicates of category Nd_3, differently from the other six. To study how different near-duplicate categories impact web-app model inference, we group these three subjects referring to them as *Nd3-Apps* and the other six as *Nd2-Apps*.

Table 3 compares the subjects webpage characteristics in terms of DOM size, complexity, and image size to *DS*. For example, the content of a web page in $\mathcal{DS}$ on an average is almost eight times that of the web pages in $\mathcal{SS}$.

6.3 Threshold-Based Classification

We aim to evaluate the effectiveness of the near-duplicate detection algorithms in classifying a given pair as either clone, near-duplicate, or distinct. Essentially, this is a multi-class classification problem, which we propose to solve using a classification function Γ. Function Γ takes as inputs a near-duplicate detection algorithm f and computes the distance between two given states in a state-pair (p_1, p_2), classifying the pair to a category according to a threshold-pair (t_c, t_n), as follows:

$$\Gamma(p_1, p_2, f, t_c, t_n) \begin{cases} Cl & : f(p_1, p_2) < t_c \\ D & : f(p_1, p_2) > t_n \\ Nd & : otherwise \end{cases}$$

To evaluate Γ, we need to find appropriate threshold values for each algorithm that maximize the classification scores.

6.3.1 Threshold Determination. We employ two different approaches, namely, *statistical* and *optimization*, to find a suitable threshold-pair (t_c, t_n) for each algorithm. In the statistical approach, we follow a data-based approach in which we use the distance distributions of different classes (Figure 3). In the optimization approach, instead, we determine the thresholds that maximize the classification score on a given labelled set, a commonly adopted strategy in machine learning for hyper-parameters selection of predictive models [47].

Definition 8 (Statistical Threshold Pair (St_c, St_n)). Threshold St_c is the 3rd quartile (Q_3) of the distances calculated by a technique on a given set of clone state-pairs, whereas, threshold St_n is the *median* distance on a given set of near-duplicate state-pairs.

Definition 9 (Optimal Threshold Pair (O_c, O_n)). Given a labelled set of clones, near-duplicates and distinct state-pairs, the optimal thresholds O_c and O_n are retrieved by a Bayesian optimization search that maximizes the average F_1 classification score for Γ over all three classes.

Figure 3 shows the distribution of distance values among the three classes, for each considered algorithm. As the box-plots show, a clear separation between distance values among classes emerged upon statistical analysis (despite some overlaps caused by outliers), which motivates using this data to determine statistical thresholds on $\mathcal{RS}$. For instance, clones (left-most plot for all techniques) have low distances, whereas distinct pairs have high distance scores. Near-duplicates, as expected, lie in between those two categories for all 10 techniques considered in our study. We use quartile data for choosing thresholds since prior work [31] has shown that the median value is a better estimator of the central tendency than mean in such cases.

We refer to the four thresholds $\{St_{c\_DS}, St_{n\_DS}, O_{c\_DS}, O_{n\_DS}\}$ as *universal thresholds*, as the state-pairs in $\mathcal{DS}$ represent a large set of randomly selected real-world webpages (see Section 5.1).

6.3.2 Classification Accuracy. To address RQ$_2$, we evaluate the algorithms by comparing the effectiveness of Γ (Section 6.3.1) with corresponding state-pair inputs. We evaluate the effectiveness of Γ using the F_1 measure, which is the harmonic mean of precision

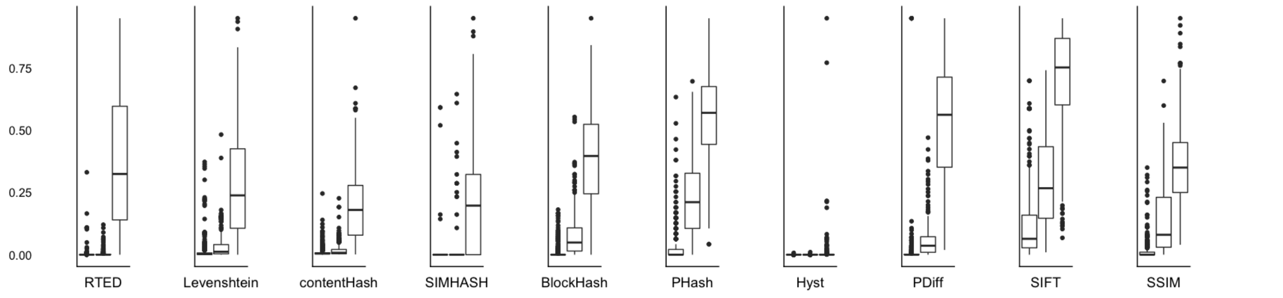


Figure 3: Normalized Distance distribution of labelled pairs in the dataset $\mathcal{DS}$. Within each box-plot, from left to right: clone, near-duplicate and distinct pairs.

Table 4: Estimated statistical (St) and optimal (O) thresholds for clone (c) and near-duplicate (n) bounds, in dataset $\mathcal{DS}$

	St_{c_DS}	St_{n_DS}	O_{c_DS}	O_{n_DS}
TLSH	0.00794	0.00794	0.01742	0.07052
Levenshtein	0.00638	0.01089	0.00704	0.07029
RTED	0.00000	0.00000	0.00007	0.04099
SimHash	0.00000	0.00000	0.00044	0.00108
BlockHash	0.00000	0.04082	0.00301	0.13371
HYST	6.52E-11	1.29E-09	1.15E-09	1.49E-08
PDIFF	0.00160	0.03800	0.00120	0.20080
PHASH	0.01754	0.17544	0.04018	0.32232
SIFT	0.16691	0.27993	0.10192	0.61876
SSIM	0.01000	0.08000	0.02020	0.15560

Table 5: F_1 Measure for Statistical and Optimal threshold sets

Algorithm	statistical (St_{c_DS},St_{n_DS})			optimal (O_{c_DS},O_{n_DS})			All		
	$\mathcal{TS}$	$\mathcal{SS}$	Avg	$\mathcal{TS}$	$\mathcal{SS}$	Avg	$\mathcal{TS}$	$\mathcal{SS}$	**Avg**
TLSH	0.50	0.40	0.45	0.56	0.44	0.50	0.53	0.42	**0.48**
Levenshtein	0.54	0.46	0.50	0.59	0.48	0.54	0.57	0.47	**0.52**
RTED	0.50	0.45	0.47	0.57	0.50	0.54	0.53	0.48	**0.50**
SIMHash	0.48	0.17	0.33	0.48	0.17	0.33	0.48	0.17	**0.33**
BlockHash	0.62	0.54	0.58	0.66	0.50	0.58	0.64	0.52	**0.58**
HYST	0.52	0.37	0.44	0.57	0.31	0.44	0.55	0.34	**0.44**
PDIFF	0.63	0.57	0.60	0.67	0.53	0.60	0.65	0.55	**0.60**
PHASH	0.59	0.43	0.51	0.63	0.40	0.52	0.61	0.41	**0.51**
SIFT	0.59	0.44	0.52	0.61	0.47	0.54	0.60	0.45	**0.53**
SSIM	0.62	0.53	0.57	0.65	0.48	0.56	0.64	0.50	**0.57**
Average	**0.56**	**0.44**	**0.50**	**0.60**	**0.43**	**0.51**	**0.58**	**0.43**	**0.51**
Random	0.32	0.32	0.32	0.32	0.32	0.32	0.32	0.32	**0.32**

Pr (ratio of correctly classified pairs to total number of classified pairs in each class), and recall Re (ratio of correctly classified pairs to the actual number of pairs that belong to the class).

Since we have more than two classes, we treat it as a multi-class classification problem, and obtain the average F_1 over the scores of all three classes (Cl, Nd, D). However, the datasets are unbalanced, i.e., the ratio of state-pairs of the classes are not equal; hence, we employ macro-averaging, to avoid favouring classes with higher representation [48]. We calculate the F_1 score of each algorithm using Γ with the universal thresholds (see Table 4) on two disjoint inputs: 1) a manually labelled random sample of 500 state-pairs, $\mathcal{TS}$, from the dataset $\mathcal{DS}$, and 2) the *97.5k* labelled pairs from $\mathcal{SS}$.

While the scores on $\mathcal{TS}$ can validate these thresholds, scores on $\mathcal{SS}$ assess the viability of discovering universal thresholds for a near-duplicate detection algorithm for unseen web apps.

6.3.3 Findings (RQ$_2$). Table 5 shows the F_1 classification scores for all techniques on the two labelled sets, $\mathcal{TS}$ and $\mathcal{SS}$. As a baseline to compare the techniques, we use a *stratified-random-classifier* [1] that classifies each state-pair randomly based on proportions of classes in the labelled set.

All evaluated techniques perform better on $\mathcal{TS}$ than $\mathcal{SS}$ when *universal thresholds* are used (+15% on average). This result is not surprising as $\mathcal{TS}$ is sampled from $\mathcal{DS}$, as well as $\mathcal{RS}$ from which we derived these thresholds. $\mathcal{SS}$, on the other hand, is completely disjoint and different from $\mathcal{DS}$ (Table 3).

Although *statistical* and *optimal* thresholds have similar overall average F_1 scores (*0.50*, *0.51*), it is important to notice that optimal thresholds perform worse than statistical thresholds on $\mathcal{SS}$, contrary to expectation.

These two findings essentially indicate that *universal thresholds may not necessarily be feasible*, and that the characteristics of web apps cannot be ignored while tuning thresholds.

Amongst the techniques, SimHash has the lowest average F_1 score (*0.17*) on $\mathcal{SS}$, almost 90% worse than the random baseline. The results concur with findings of a previous study [29], which points to the fact that the algorithm is poor at distinguishing states that belong to the same app.

On average, *five out of top six* techniques belong to the computer vision domain. *PDIFF* is the best with a classification F_1 score of *0.60*, **>85%** better than the baseline and **>13%**, **>20%** better than Levenshtein and TLSH, the best techniques in DOM and IR categories, respectively. On average, most visual techniques outperform DOM and IR techniques (with the exception of *PHash* and *color-histogram*). On $\mathcal{SS}$, *PDIFF* again outperforms all techniques while *BlockHash* and *SSIM*, both visual, are the only other techniques that have an F_1 score of more than *0.50*.

7 RQ$_3$: IMPACT ON INFERRED MODELS

With RQ$_3$, we evaluate the impact of the near-duplicate detection algorithms in automated web app model inference.

Specifically, we evaluate the quality of crawl models inferred using each of the near-duplicate detection algorithms as *state abstract function (SAF)* (see Definition 1) along with the determined thresholds. Crawljax already includes all DOM-based algorithms described in Section 3.2; we added the computer vision and information retrieval near-duplicate algorithms within Crawljax as SAFs. More specifically, we integrated the implementations of PDiff, SIFT, and SSIM from the open-source computer-vision library OpenCV, and the publicly available versions of TLSH[2] and `simhash`.[3]

Since we need to run and analyze many crawl sessions (i.e., nine apps, 10 algorithms, different threshold sets), we limit the crawl session with a *maximum runtime* of five minutes.

Model Quality. We measure the quality of a generated model through its F_1 score, the harmonic mean of Pr and Re. Lower precision (Pr) denotes a greater redundancy in the model and is computed as the ratio of unique states (*bins*) covered by the model to the total number of states in the model. Recall (Re) quantifies the application state coverage achieved in the model and is computed as the number of *bins* covered by the model to the total number of *bins* identified by humans, for the corresponding app, in the *ground truth* (see Section 6.2).

The recall Re of a crawl model is highly dependent on the ability of the SAF to reliably distinguish the distinct state-pairs and its precision Pr on its ability to exclude near-duplicates and clones of states already present from the model. Crawlers, however, typically expect one single similarity threshold for deciding if a state is new to be added to the model; i.e., they do not distinguish between clone/near-duplicate. Therefore, we frame the problem of finding optimal thresholds for a SAF as maximizing the F_1 score of its *distinct-pair detection.* Therefore, from the thresholds we derived in Section 6.3.1, we use the near-duplicate thresholds, which are designed to distinguish distinct pairs from near-duplicates. As the clone thresholds are lower than the near-duplicate thresholds, a near-duplicate threshold should be able to distinguish distinct pairs from clones as well.

Performing multiple crawls using the near-duplicate techniques as SAFs, and evaluating the generated models is a manually expensive process. Thus, we first assess the techniques and the *universal* thresholds automatically, based on the F_1 score of the distinct-pair detection, which indicates their applicability as SAFs.

Findings (RQ$_3$): Distinct-pair Detection. In the distinct state-pair detection scores from RQ$_2$ shown in Table 6, scores on $\mathcal{TS}$ allow us to assess the ability of a technique to distinguish distinct state-pairs in the wild, while $\mathcal{SS}$ lets us simulate each technique as a SAF on generated models captured in our *subject-set.* In contrast to the RQ$_2$ results, where both the threshold sets had better average classification F_1 on $\mathcal{TS}$ compared to $\mathcal{SS}$, Table 6 shows that *statistical threshold* had better distinct state-pair detection F_1 of 0.78 on $\mathcal{SS}$ than 0.73 in $\mathcal{TS}$. Optimal threshold O_{n_DS}, which is higher/stricter than St_{n_DS}, in terms of actual threshold value, as shown in Table 4, has a poor *recall* on $\mathcal{SS}$ (53%) compared to $\mathcal{TS}$ (81%). Also in $\mathcal{TS}$ statistical threshold has the highest recall, but by sacrificing precision; the optimal threshold emerges with a better overall F_1 score through a 25% better precision on $\mathcal{TS}$. The same threshold, however, could not improve precision in $\mathcal{SS}$ but has 50% lower recall.

Table 6: Distinct pair (Pr, Re, F_1) on existing datasets

	$\mathcal{TS}$			$\mathcal{SS}$			Average		
	Pr	Re	F_1	Pr	Re	F_1	Pr	Re	F_1
O_{n_DS}	0.81	0.81	0.80	0.89	0.53	0.64	0.85	0.67	0.72
St_{n_DS}	0.63	0.90	0.73	0.87	0.76	0.78	0.75	0.83	0.76

As we optimized our threshold to be stricter to fit the distribution in $\mathcal{DS}$, we ended up misclassifying distinct pairs to be near-duplicates in $\mathcal{SS}$ because of the differences in the distributions between the two data-sets. As we pointed out in RQ$_2$, *these results shows the infeasibility of finding universal thresholds as the distances for state-pairs are highly influenced by the intrinsic characteristics of the web app they belong to.*

Application Knowledge for Obtaining Thresholds. These results for universal thresholds prompted us to investigate whether having knowledge of the web app characteristics helps in selecting better thresholds to improve the detection rates of the techniques.

We use the manually labelled models (see Section 6.2) in the subject-set ($\mathcal{SS}$) for each app to represent application knowledge. In order to use this application knowledge, we apply the near-duplicate threshold definitions in Definition 8 and Definition 9 to each subject in $\mathcal{SS}$ to derive St_{n_SS} and O_{n_SS} respectively. In addition to these two thresholds, through initial experiments, we have observed that category Nd_3 near-duplicates overlaps with distinct (Di) pairs and it is not possible to design a threshold that can distinguish them. We therefore created a new threshold definition that sacrifices the precision of distinct pair detection by allowing misclassification of Nd_3 near-duplicates as Di for better recall (Re).

Definition 10. St_{n3} is defined as the *median* of the data distribution of manually labelled near-duplicates $\{Nd_1 \lor Nd_2\}$. In other words, St_{n3} is St_n computed after excluding Nd_3 near-duplicates.

We refer to these thresholds obtained by applying application knowledge in $\mathcal{SS}$ for each algorithm as *app-specific thresholds.* We crawled each of our subjects with two universal and three app-specific thresholds with each technique as a SAF, separately, and assess the quality of the generated models.

Findings (RQ3): Impact on Generated Models. Table 7 shows the average F_1 of crawls for all algorithms for each threshold. Overall, as expected, the universal optimal near-duplicate threshold O_{n_DS} has the worst score of *0.24*; only half of the 0.42 scored by the best threshold O_{n_SS}, the optimal threshold derived with application knowledge. On average, app-specific thresholds improve the model quality by *34%* compared to universal thresholds underlining the need to *consider app characteristics to choose thresholds.* For *Nd3-Apps*, it can be seen that St_{n3_SS} derived using the statistical Definition 10 significantly (90%) improves the F_1 score over the St_{n_SS}, showing that *threshold design needs to consider fine-grained near-duplicate categories prevalent in the app under test.* Overall, *app-specific thresholds produce better models.*

[2]https://github.com/idealista/tlsh
[3]https://github.com/albertjuhe/charikars_algorithm

Table 7: Inferred model F_1 score

	Universal			App-Specific			
	St_{n_DS}	O_{n_DS}	Avg	St_{n_SS}	St_{n3_SS}	O_{n_SS}	Avg
AddressBook	0.33	0.27	0.30	0.17	0.46	0.41	0.34
PetClinic	0.36	0.25	0.31	0.50	0.50	0.52	0.51
Claroline	0.30	0.18	0.24	0.42	0.42	0.44	0.43
DimeShift	0.31	0.22	0.26	0.33	0.33	0.38	0.34
PageKit	0.30	0.27	0.29	0.27	0.39	0.37	0.34
Phoenix	0.44	0.29	0.37	0.24	0.47	0.42	0.38
PPMA	0.31	0.19	0.25	0.49	0.49	0.51	0.49
MRBS	0.37	0.35	0.36	0.43	0.43	0.46	0.44
MantisBT	0.24	0.18	0.21	0.26	0.26	0.27	0.26
Average	0.33	0.24	0.29	0.34	0.41	0.42	0.39
Nd2-Apps	0.32	0.23	0.27	0.40	0.40	**0.43**	0.41
Nd3-Apps	0.36	0.28	0.32	0.23	**0.44**	0.40	0.35

Table 8: Inferred model F_1 for each algorithm for selected thresholds

Thresholds	Apps	TLSH	SIMHash	Levenshtein	RTED	BlockHash	PHASH	HYST	PDIFF	SIFT	SSIM	Average
All Five	All	0.10	0.05	0.47	**0.62**	0.46	0.39	0.41	0.34	0.34	0.35	0.35
	Nd2	0.10	0.04	0.48	0.62	0.47	0.39	0.41	0.36	0.31	0.39	0.36
	Nd3	0.10	0.06	0.43	0.62	0.43	0.40	0.41	0.29	0.39	0.28	0.34
O_{n_SS}	All	0.15	0.08	0.48	0.55	0.54	0.49	0.54	0.45	0.42	0.51	0.42
	Nd2	0.17	0.08	0.53	0.61	0.52	0.49	0.58	0.46	0.37	0.52	**0.43**
	Nd3	0.10	0.10	0.37	**0.43**	0.58	0.49	0.45	0.42	0.52	0.51	0.40
St_{n3_SS}	All	0.09	0.03	0.46	0.67	0.57	0.50	0.55	0.43	0.36	0.48	0.41
	Nd2	0.08	0.02	0.47	0.62	0.55	0.50	0.53	0.44	0.35	0.46	0.40
	Nd3	0.10	0.07	0.44	**0.76**	0.60	0.51	0.60	0.42	0.37	0.51	**0.44**

Table 8 shows the average F_1 scores for each algorithm for five minute crawls on our subjects. *RTED* consistently outperforms other techniques with an F_1 score of 0.62 averaged over all five thresholds. it is 29% better than Levenshtein, the next best algorithm.

The results for visual techniques in Table 8 contradict our expectations given that, in RQ$_2$, they convincingly outperformed the DOM and IR techniques in state-pair classification using Γ.

An analysis of visited states per minute or *speed* of the algorithms, shown in Table 9, seems to suggest that faster algorithms such as *RTED* (25 states per minute) could explore more states in a given crawl time and improve its *Re*. On the other hand, slower algorithms such as *PDiff*, which could only explore *four* states per minute on average, are at a clear disadvantage. Also, visual techniques, unlike DOM based algorithms such as RTED, do not rely on characteristics that can directly capture differences *corresponding to web elements* (e.g., SIFT keypoints), essential to be able to classify states similar to a human tester.

In IR techniques, SimHash is not able to distinguish even two completely different states in our subject-set as already seen in RQ$_2$. TLSH on the other hand, fails to calculate digests for app states of our subjects due to lack of enough complexity as shown in Table 3—the content in our subjects is 1/9th of the content size

Table 9: Techniques *Speed* and Inferred model (Re, Pr, F_1) for best 5-minute crawls

	Levenshtein	RTED	BlockHash	PHASH	HYST	PDIFF	SIFT	SSIM
Speed	11	**25**	17	16	16	**4**	5	8
Recall	0.42	**0.61**	0.49	0.49	0.55	0.30	0.28	0.39
Precision	**0.84**	0.79	0.75	0.79	0.72	**0.91**	0.71	**0.85**
F1	**0.54**	**0.66**	**0.54**	0.52	**0.58**	0.44	0.39	0.51

Table 10: Inferred model F_1 for 30-Minute crawls

Apps	BlockHash	Hyst	Levenshtein	PDiff	RTED	SSIM
All	0.51	0.57	0.53	0.52	0.62	0.56
Nd2	0.57	0.62	0.59	0.51	0.66	0.52
Nd3	0.39	0.47	0.42	**0.56**	0.52	**0.64**

in the wild. Therefore, we exclude SimHash and TLSH from further analysis.

Comparison of Optimally Configured Techniques. Table 8 shows that for all remaining eight techniques with the exception of SIFT, O_{n_SS} for Nd2-Apps and St_{n3_SS} for Nd3-Apps is the best threshold configuration.

Table 9 shows the statistics of the 5-min crawls for each technique with their best threshold configuration. Coverage (Re) data suggests that 5 minutes was not enough to cover all of the app state-space. In our next experiment, therefore, we use longer crawl time of 30 minutes. Given the exponential nature of increase in manual effort to analyze larger crawl models, we limit this experiment to the best performing techniques tuned with thresholds from best 5-minute crawls presented in Table 9. We select the top four techniques based on F_1 scores, however, as discussed before, since the slower algorithms were placed at a disadvantage in the 5-minute crawls, we also include PDiff and SSIM that produced models with the best precision (Pr) scores 0.91 and 0.85 (respectively 12% and 6% better than RTED which has the best F_1 score of 0.66).

Findings (RQ$_3$): Technique Comparison in 30-min crawls. Average F_1 scores shown in Table 10 for 30 minute crawls indicate that, when tuned correctly and given enough time, Histogram, BlockHash, RTED and Levenshtein can all perform well on Nd2-Apps, i.e., they managed to discard near-duplicates of type Nd_2 reasonably well. However, it is surprising to see that PDiff and SSIM score higher than all of them on Nd3-Apps. Thus, we decided to analyze how F_1 has changed over the 30 minutes for Nd3-Apps as opposed to the Nd2-Apps.

A plot of F_1 of the model over its states percentage for RTED crawls is shown in Figure 4. The figure highlights that for *Nd3-Apps, model deteriorates as states being added are near-duplicates, mostly of type Nd_3*, while, the models of Nd2-Apps seem to stabilize as Nd_2 near-duplicates are being detected and discarded. During the manual analyses of models, we observed that the Nd_3 near-duplicates

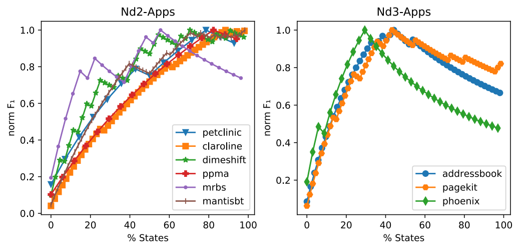


Figure 4: Normalized F_1 over %(states in model) during 30-minute crawls of RTED

are dynamically created, typically through user-interactions that result in addition/removal of web elements whose functionality already exists in the state (e.g., addition/deletion of new rows in a table). Not only is this newly created state a near-duplicate that will waster into precious testing time, but each time the crawler back-tracks to reach this state, it may invoke the same creation path adding even more duplicates resulting in a never-ending loop.

Given that RTED is the best algorithm and was fine-tuned to produce best model for each application, this surprising revelation points to the limitation of existing crawlers and threshold based SAFs and shows that *threshold based crawling may never produce an accurate and complete model of modern web apps with dynamic Nd3 near-duplicates*. We therefore think that future SAFs should incorporate characteristics that represent functionality and crawlers should be designed to utilize near-duplicate detection to establish the nature of duplication instead of quantifying the computed differences to actively guide the exploration to discover newer functionality.

8 THREATS TO VALIDITY

External validity threats concern the generalization of our findings. We considered only nine web apps and experiments with other subject systems are necessary to fully confirm the generalizability of our results, and corroborate our findings. We tried to mitigate this threat selecting real-world web apps with different sizes, pertaining to different domains, and adopted in previous web testing work [15, 16, 49]. Another threat concerns the selection of thresholds for near-duplicate detection techniques, whose results may not generalize to other algorithms. We mitigated this threat by selecting 10 techniques from three different domains: web testing, computer vision and information retrieval. *Internal validity* threats concern uncontrolled factors that may have affected our results. A possible threat is represented by the manually created ground truth, which was unavoidable because no automated method could provide us with the ideal classification of web pages. To minimize this threat, the authors of this paper created, in isolation, a ground truth. Then, the two established a discussion to produce a single ground truth for each web app.

For reproducibility of the results, we made our tool, datasets and used subject systems available [5], along with required instructions.

9 RELATED WORK

A large body of research has addressed the analysis of web sites structure via clustering for clone detection and duplicate removal of web pages [18, 19, 22–25, 29, 34, 41, 56].

Henzinger [29] performed an evaluation of two near-duplicate detection algorithms based on shingling on a large dataset of 1.6B web pages. Their results show that neither algorithm works well in finding near-duplicate pairs within the same site, while both achieve high precision for near-duplicate pairs from different sites. Manku et al. [34] followed up on the work using `simhash` to detect near-duplicates for web information retrieval, data extraction, plagiarism and spam detection with promising results. Fetterly et al. [23] study the evolution of near-duplicate web pages over time. Their results show that near-duplicates have little variability over time, as two pages that have been found to be near-duplicates of one another will continue to be so for the foreseeable future.

Our study is different from the above work as we aim to detect near-duplicates *within* web apps and not across different web apps. Regarding detection of within app near-duplicates, Calefato et al. [19] propose a method to identify near-duplicates as well as functional clone web pages based on a manual visual inspection of the GUI. Crescenzi et al. [22] propose a structural abstraction for web pages as well as a clustering algorithm that groups web pages based on this abstraction. Di Lucca et al. [24, 25] evaluate the Levenshtein distance and the tag frequency methods for detecting near-duplicate web pages. Eyk et al. apply simhash and broders near-duplicate detection within Crawljax [27].

In the broader research on GUI similarity detection, researchers [3, 14] have attempted to use GUI widget hierarchies specific to mobile applications in order to design optimal state abstractions. Our study did not consider such techniques as they are not directly applicable for web applications.

To the best of our knowledge, our work is the first one to study different near-duplication detection algorithms (from different fields) as SAFs in a web crawler. This paper is the first to propose a systematic categorization of near-duplicates in web apps, from a functional E2E testing perspective and to study the impact of near-duplicate detection on generated web application models and web testing. Moreover, our paper is the first to discuss selection of thresholds for near-duplicate detection, an important first step.

10 CONCLUSIONS AND FUTURE WORK

Automatically asserting the equality of two complex web pages is a difficult problem which the state abstraction function of a crawler needs to solve at runtime during the exploration. The problem is further complicated by the presence of near-duplicates which need to be detected and mapped to the logical pages in order to produce meaningful crawl models.

We study ten existing near-duplicate detection techniques from three different domains for the purpose and compare their effectiveness as SAFs in a crawler. Our results show that near-duplicates of Nd_2 kind are detectable by most techniques when configured with optimal thresholds found by using application knowledge. However, no technique is able to detect Nd_3 near-duplicates leading to poor inferred models.

Future work includes devising novel types of SAFs, incorporating both DOM and visual characteristics in a single hybrid solution to detect different kinds of near-duplicates while the crawler also needs to be improved to utilize the knowledge of duplication seen in detected near-duplicates to guide the exploration.

REFERENCES

[1] [n.d.]. Stratified Random Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html. Package: scikit-learn.

[2] S. Afroz and R. Greenstadt. 2011. PhishZoo: Detecting Phishing Websites by Looking at Them. In *2011 IEEE Fifth International Conference on Semantic Computing*. 368–375. https://doi.org/10.1109/ICSC.2011.52

[3] D. Amalfitano, A. R. Fasolino, and P. Tramontana. 2011. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 252–261. https://doi.org/10.1109/ICSTW.2011.77

[4] and A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (April 2004), 600–612. https://doi.org/10.1109/TIP.2003.819861

[5] anon. 2019. Near-Duplicate Study Tools and DataSet For Replication. https://github.com/NDStudyICSE2019/NDStudy. GitHub Repository.

[6] app1 2018. Angular version of the Spring PetClinic web application. https://github.com/spring-petclinic/spring-petclinic-angular.

[7] app3 2015. Claroline. Open Source Learning Management System. https://sourceforge.net/projects/claroline/.

[8] app4 2018. DimeShift: easiest way to track your expenses. https://github.com/jeka-kiselyov/dimeshift.

[9] app5 2018. Pagekit: modular and lightweight CMS. . https://github.com/pagekit/pagekit.

[10] app6 2018. Phoenix: Trello tribute done in Elixir, Phoenix Framework, React and Redux. https://github.com/bigardone/phoenix-trello.

[11] app7 2018. PHP Password Manager. https://github.com/pklink/ppma.

[12] app8 2018. Meeting Room Booking System. https://mrbs.sourceforge.io/.

[13] app9 2018. Mantis Bug Tracker. https://github.com/mantisbt/mantisbt.

[14] Young-Min Baek and Doo-Hwan Bae. 2016. Automated Model-Based Android GUI Testing Using Multi-Level GUI Comparison Criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 238?249. https://doi.org/10.1145/2970276.2970313

[15] Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. 2019. Web Test Dependency Detection. In *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, 12 pages.

[16] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-based Web Test Generation. In *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, 12 pages.

[17] Lorenzo Blanco, Nilesh Dalvi, and Ashwin Machanavajjhala. 2011. Highly Efficient Algorithms for Structural Clustering of Large Websites. In *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*. ACM, 437–446.

[18] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. 1997. Syntactic Clustering of the Web. *Comput. Netw. ISDN Syst.* 29, 8-13 (Sept. 1997), 1157–1166.

[19] Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. 2004. Function Clone Detection in Web Applications: A Semiautomated Approach. *J. Web Eng.* 3, 1 (May 2004), 3–21.

[20] Moses S. Charikar. 2002. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing (STOC '02)*. ACM, New York, NY, USA, 380–388. https://doi.org/10.1145/509907.509965

[21] Teh-Chung Chen, Scott Dick, and James Miller. 2010. Detecting Visually Similar Web Pages: Application to Phishing Detection. *ACM Trans. Internet Technol.* 10, 2, Article 5 (June 2010), 38 pages. https://doi.org/10.1145/1754393.1754394

[22] Valter Crescenzi, Paolo Merialdo, and Paolo Missier. 2005. Clustering Web Pages Based on Their Structure. *Data Knowledge Engineering* 54, 3 (Sept. 2005), 279–299.

[23] Marc Najork Dennis Fetterly, Mark Manasse. 2004. On the Evolution of Clusters of Near-Duplicate Web Pages, In Journal of Web Engineering. *Journal of Web Engineering* 2, 228–246.

[24] Giuseppe A. Di Lucca, Massimiliano Di Penta, Anna Rita Fasolino, and Pasquale Granato. 2001. Clone Analysis in the Web Era: an Approach to Identify Cloned Web Pages. In *Proceedings of the International Workshop of Empirical Studies on Software Maintenance - November 2001 - Florence - Italy*. 107–113.

[25] Giuseppe Antonio Di Lucca, Massimiliano Di Penta, and Anna Rita Fasolino. 2002. An Approach to Identify Duplicated Web Pages. *2013 IEEE 37th Annual Computer Software and Applications Conference* 00, undefined (2002), 481.

[26] Cristian Duda, Gianni Frey, Donald Kossmann, Reto Matter, and Chong Zhou. 2009. AJAX Crawl: Making AJAX Applications Searchable. In *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE '09)*. IEEE, 78–89.

[27] E.D.C. Van Eyk and W. J. Van Leeuwen. 2014. *Performance of near-duplicate detection algorithms for Crawljax*. B.S. Thesis.

[28] Taher H. Haveliwala, Aristides Gionis, Dan Klein, and Piotr Indyk. 2002. Evaluating Strategies for Similarity Search on the Web. In *Proceedings of the 11th International Conference on World Wide Web (WWW '02)*. ACM, New York, NY, USA, 432–442. https://doi.org/10.1145/511446.511502

[29] Monika Henzinger. 2006. Finding Near-duplicate Web Pages: A Large-scale Evaluation of Algorithms. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '06)*. ACM, 284–291.

[30] VI Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (1966), 707.

[31] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. 2013. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology* 49, 4 (2013), 764 – 766. https://doi.org/10.1016/j.jesp.2013.03.013

[32] D. G. Lowe. 1999. Object recognition from local scale-invariant features. In *Proceedings of Seventh IEEE International Conference on Computer Vision*, Vol. 2. 1150–1157.

[33] Sonal Mahajan and William G.J. Halfond. 2014. Finding HTML Presentation Failures Using Image Comparison Techniques. In *Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, 91–96.

[34] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. Detecting Near-duplicates for Web Crawling. In *Proceedings of the 16th International Conference on World Wide Web (WWW '07)*. ACM, 141–150.

[35] Ali Mesbah. 2015. *Advances in Testing JavaScript-based Web Applications*. Advances in Computers, Vol. 97. Elsevier, Chapter 5, 201–235.

[36] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web* 6, 1 (2012), 3:1–3:30.

[37] Amin Milani Fard and Ali Mesbah. 2013. Feedback-directed Exploration of Web Applications to Derive Test Models. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 278–287.

[38] J. Oliver, C. Cheng, and Y. Chen. 2013. TLSH – A Locality Sensitive Hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. 7–13.

[39] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient Computation of the Tree Edit Distance. *ACM Trans. Database Syst.* 40, 1, Article 3 (March 2015), 40 pages.

[40] PHP AddressBook. 2015. Simple, web-based address & phone book. http://sourceforge.net/projects/php-addressbook. Accessed: 2018-10-01.

[41] Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglis. 2004. Automatic Detection of Fragments in Dynamically Generated Web Pages. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM, 443–454.

[42] Filippo Ricca and Paolo Tonella. 2001. Analysis and testing of Web applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*. IEEE, 25–34.

[43] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2013. X-PERT: Accurate Identification of Cross-browser Issues in Web Applications. In *Proc. of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, 702–711.

[44] Sreedevi Sampath. 2012. Advances in User-Session-Based Testing of Web Applications. *Advances in Computers* 86 (2012), 87–108.

[45] M. Schur, A. Roth, and A. Zeller. 2015. Mining Workflow Models from Web Applications. *IEEE Transactions on Software Engineering* 41, 12 (Dec 2015), 1184–1201. https://doi.org/10.1109/TSE.2015.2461542

[46] Selenium 2018. SeleniumHQ Web Browser Automation. http://www.seleniumhq.org/. Accessed: 2017-08-01.

[47] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2951–2959. http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf

[48] Marina Sokolova and Guy Lapalme. 2009. A systematic analysis of performance measures for classification tasks. *Information Processing Management* 45, 4 (2009), 427 – 437. https://doi.org/10.1016/j.ipm.2009.03.002

[49] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2016. Clustering-Aided Page Object Generation for Web Testing. In *Proceedings of 16th International Conference on Web Engineering (ICWE 2016)*. Springer, 132–151.

[50] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2017. APOGEN: Automatic Page Object Generator for Web Testing. *Software Quality Journal* 25, 3 (Sept. 2017), 1007–1039.

[51] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual Web Test Repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 503–514. https://doi.org/10.1145/3236024.3236063

[52] Michael J. Swain and Dana H. Ballard. 1992. Indexing via Color Histograms. In *Active Perception and Robot Vision*, Arun K. Sood and Harry Wechsler (Eds.). Springer Berlin Heidelberg, 261–273.

[53] Anastasios Tombros and Zeeshan Ali. 2005. Factors Affecting Web Page Similarity. In *Proceedings of the 27th European Conference on Advances in Information Retrieval Research (ECIR 2005)*. Springer-Verlag, 487–501.

[54] Paolo Tonella, Filippo Ricca, and Alessandro Marchetto. 2014. Recent Advances in Web Testing. *Advances in Computers* 93 (2014), 1–51.

[55] J. Upchurch and X. Zhou. 2016. Malware provenance: code reuse detection in malicious software at scale. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*. 1–9. https://doi.org/10.1109/MALWARE.2016.7888735

[56] Yitong Wang and Masaru Kitsuregawa. 2001. *Link Based Clustering of Web Search Results*. Springer Berlin Heidelberg, 225–236.

[57] B. Yang, F. Gu, and X. Niu. 2006. Block Mean Value Based Image Perceptual Hashing. In *2006 International Conference on Intelligent Information Hiding and Multimedia*. 167–172.

[58] Hector Yee, Sumanita Pattanaik, and Donald P. Greenberg. 2001. Spatiotemporal Sensitivity and Visual Attention for Efficient Rendering of Dynamic Environments. *ACM Trans. Graph.* 20, 1 (Jan. 2001), 39–65.

[59] Christoph Zauner. 2010. *Implementation and benchmarking of perceptual image hash functions*. Ph.D. Dissertation.