Neural Embeddings for Web Testing

Andrea Stocco, *Member, IEEE Computer Society*, Alexandra Willi, Luigi Libero Lucio Starace, Matteo Biagiola, and Paolo Tonella, *Member, IEEE Computer Society*

Abstract—Web test automation techniques employ web crawlers to automatically produce a web app model that is used for test generation. Existing crawlers rely on app-specific, threshold-based, algorithms to assess state equivalence. Such algorithms are hard to tune in the general case and cannot accurately identify and remove near-duplicate web pages from crawl models. Failing to retrieve an accurate web app model results in automated test generation solutions that produce redundant test cases and inadequate test suites that do not cover the web app functionalities adequately. In this paper, we propose WEBEMBED, a novel abstraction function based on neural network embeddings and threshold-free classifiers that can be used to produce accurate web app models during model-based test generation. Our evaluation on nine web apps shows that WEBEMBED outperforms state-of-the-art techniques by detecting near-duplicates more accurately, inferring better web app models that exhibit 22% more precision, and 24% more recall on average. Consequently, the test suites generated from these models achieve higher code coverage, with improvements ranging from 2% to 59% on an app-wise basis and averaging at 23%.

Index Terms—Web Testing, Neural Embeddings, GUI Testing, Doc2Vec.

1 INTRODUCTION

Test automation is used to enable end-to-end (E2E) functional testing of web apps. In this approach, testers exercise the application under test (AUT) with automated scripts [**?**], [**?**], [**?**] that imitate the end-user interactions with the web pages by simulating user-like events, such as clicks, scrolls, and form submissions. Therefore, the web app is tested for its ability to provide correct functionalities to the end user, through its graphical user interface (GUI).

The manual development of E2E test automation scripts is a costly endeavor in practice, and so is the maintenance of such scripts over time [**?**]. For this reason, researchers have proposed automated test generation solutions, most of which rely on a model of the web app [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. Model-based web testing techniques systematically build a web app model by exploring the functionalities of a given web app by means of a crawler [**?**], [**?**], [**?**]. The model is represented in terms of web app states, i.e., logical functional units, and transitions between states triggered by events (e.g., clicks). Ideally, a good web app model should contain all possible logical web pages—i.e., it should be *complete*—without representing the same logical page multiple times—i.e., it should be *concise* [**?**], [**?**].

To automatically determine the logical web app states, model-based techniques use a scoring function, called *state abstraction function* (SAF). When the SAF is ineffective, it causes clone or *near-duplicate* states that pollute the model. Near-duplicates are concrete instances of the same logical state, differing only by minor changes [**?**].

- *Andrea Stocco is with the Technical University of Munich, and fortiss GmbH, Germany. The work was carried out at the Università della Svizzera italiana, Switzerland.*
- *Luigi Libero Lucio Starace is with the Università degli Studi di Napoli Federico II, Italy.*
- *Alexandra Willi, Matteo Biagiola, and Paolo Tonella are with the Università della Svizzera italiana, Switzerland.*

The presence of near-duplicates makes a crawl model not concise (i.e., the same state appears multiple times) and incomplete because, in the presence of duplicated states, the crawler will waste part of its finite exploration budget re-exploring the same state many times, possibly missing other important, not yet discovered, states. A web app model containing near-duplicates undermines the effectiveness of the test suites generated from it in terms of completeness and adequacy [**?**], [**?**]. In fact, missing states will remain untested, potentially reducing the test suite adequacy (e.g., code coverage). Moreover, duplicated states might lead to the generation of redundant test cases that do not contribute to increasing the code coverage of the AUT [**?**].

A recent study [**?**] shows that current state-of-the-art SAF implementations, based on similarity algorithms, hashing algorithms, or visual resemblance of web app snapshots, are application dependent, as no algorithm is comparably effective across different web apps. Moreover, the study reports that, even for an effective SAF within the same web app, it is challenging for developers to find the optimal threshold that can detect near-duplicate states without collapsing logically distinct states into the same one [**?**].

This paper investigates the problem of building a robust SAF using *neural network embeddings* of web pages. An embedding is a mapping of an input belonging to a complex input space (e.g., natural language, images, or web pages) to a low-dimensional and continuous vector representation belonging to a *latent space* [**?**]. Embeddings are useful because of their capability to preserve the semantic similarity that holds in the original complex input space [**?**], which makes them suitable to address the web app similarity problem.

Our approach, implemented in a tool called WEBEMBED, consists of a novel SAF that turns multiple intermediate token-sequence representations of web pages into n-dimensional vector representations used by a classifier to estimate the similarity, or lack thereof, of web pages. Although there are many methods to produce vector em-

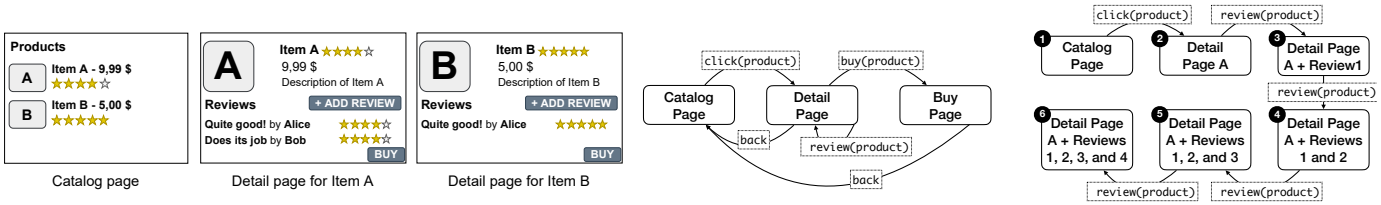

Fig. 1: Left: E-commerce web application. Center: Optimal web app model. Right: Incomplete crawl model w.r.t. the "buy" functionality due to redundant near-duplicate states for the same "Detail page". States ❸, ❹, ❺, and ❻ are functional near-duplicate states of state ❷.

beddings [?], [?], [?], [?], [?], this paper focuses on the vector representation produced by Doc2Vec [?], as web pages are a mixture of text and tags. More specifically, WEBEMBED is trained on three Doc2Vec models on a large corpus of web pages. The input to each of the three models is respectively the token-sequence representation of web pages built from their tags, from their textual content, or the union of the two. Once trained, WEBEMBED uses the three Doc2Vec models within a web crawler: upon exploration, for each web page, its three token-sequence representations are retrieved, the corresponding three neural embeddings are computed and compared with the states already present in the web app model, resulting in three similarity scores. The considered web page is retained only if it is not a near-duplicate state, according to a pre-trained classifier that maps the three similarity scores of two web pages into class 1 (near-duplicates) or 0 (distinct).

We have evaluated WEBEMBED empirically using benchmarks available from the literature containing a diverse set of web apps. We assessed three different tasks, namely near-duplicate detection, model coverage, and test generation, on three use cases with different requirements in terms of labeling cost for developers. In our experiments, accounting for more than 450 configurations, WEBEMBED achieved high accuracy scores on all use cases for all tasks, with a statistically significant margin over two existing state-of-the-art SAFs. Quite notably, our approach is threshold-free and can be applied even without defining any app-specific corpus of labeled data, still achieving satisfactory performance (75% accuracy on near-duplicate detection and 82% on model coverage). When a corpus of labeled data is available for a given web app or sufficient labeling cost is allowed, the accuracy of WEBEMBED is further improved (93% on near-duplicate detection—a 22% increment—and 92% concerning model coverage—10% increment). Lastly, by employing WEBEMBED, tests generated based on crawled models result in higher code coverage, with an average increase of 23% compared to current state-of-the-art techniques (ranging from 2% to 59% on an app-wise basis).

Our paper makes the following contributions:

Technique. A novel approach, implemented in the publicly available tool WEBEMBED [?], which uses neural embeddings and classifiers for web crawling and testing.

Evaluation. An empirical study showing that WEBEMBED is more effective than two state-of-the-art SAFs in the near-duplicate detection, model coverage, and test generation tasks under different configurations/use cases.

2 BACKGROUND

In this section, we describe the problem of automatically retrieving an accurate web app model for test generation and its challenges. Then, we introduce the concept of neural embeddings to overcome such challenges.

We use as a running example a simple e-commerce web app showing a product catalog. A user can view the details of a product, add a review and buy it (Figure 1 (left)).

2.1 Automated Web Model Inference

Automated web model inference techniques, such as crawling, operate through state exploration by triggering events (e.g., clicks) and by generating inputs that cause state transitions in the web app. Whenever significant changes in the current web page are detected, a new *state* is added to the model. A state can be viewed as an abstraction of all the dynamic, runtime versions of the same logical web page, often represented by their Document Object Models (DOMs). The final model is a set of states, i.e., the set of abstract web pages of the web app, and edges that represent transitions between states.

From a functional testing viewpoint, the optimal web app model, in terms of logical states and functionalities, is shown in Figure 1 (center). The model includes three states, namely *Catalog page*, *Detail page* and *Buy page*. From the Catalog page, it is possible to navigate to the Detail page by clicking on a product. From a product Detail page, it is possible to either write a review for the product, which leads back to the same page, or buy the product, which causes a transition to the Buy page. From both the Detail and Buy pages, users can navigate back to the initial Catalog page.

2.2 Near-duplicate States

Figure 1 (right) shows a crawl model produced by the state-of-the-art crawler Crawljax [?] with its default configuration, which consists of: (1) no state abstraction capability, i.e., all dynamic states are regarded as new states; (2) an ordered GUI events queuing strategy that considers HMTL elements from top to bottom and from left to right; (3) a depth-first exploration strategy.

In particular, once the web app is loaded, the crawler saves the initial home page (also called *index* page) as the first state of the crawl model (i.e., state ❶). In our running example, the index page is the Catalog page. Then, the crawler clicks on the first displayed product, i.e., Item A, which leads to the web page showing the item details. Such page is saved as a state into the crawl model (Detail page

A) and marked as *unvisited* (i.e., state ❷). Since a new state is added, the crawler returns to the index page (not shown in Figure 1) and crawls back to the newly added unvisited state. Next, the crawler clicks on the Add Review button, adds Review 1 to the page, either with random or manually provided input data, and stores another state to the crawl model (Detail page A + Review 1), marking it as unvisited since it is regarded as a different state. Similarly to the previously found state (i.e., state ❷), the crawler returns to the index page and crawls back to state ❸ by clicking on Item A. Since state ❷ is a new state and the "Add review" clickable is unfired, the crawler adds a new review creating a new state (i.e., state ❹); the process repeats until the crawler runs out of budget.

From a testing viewpoint, all web pages containing the details of the selected product and its reviews should be collapsed into the same logical page, which is not the case with our crawl model in which many similar replicas of the Detail page for Item A are present. In the literature, concrete instances of the same logical page, such as the detail pages of our example, are known as clones, or near-duplicate states [?]. The presence of near-duplicate states in web app models has a detrimental effect on the effectiveness of model-based test generation techniques, in terms of *conciseness* and *completeness*. Concerning the former, the presence of near-duplicates typically leads to test suites containing many redundant tests exercising the same functionality. In our running example, it would be sufficient to cover the Detail page only once with a test case, as covering all potential detail pages with many redundant test cases is unlikely to increase the code coverage achieved by the test suite [?]. As for completeness, when exploring large web apps, crawlers may waste a considerable part of their time budget visiting near-duplicate states, without exploring other relevant parts of the application. This harms the completeness of the inferred models, and thus the associated test suites. In the running example, the crawler failed to recognize that the 'new' updated Detail page for Item A, featuring the reviews, was a *near-duplicate* of the previously-visited Detail page. Therefore, it consumed the entire time budget failing to explore other significant parts of the application, such as the "buy" functionality, leading to an incomplete model.

The SAF used by the crawler is the main root cause for the lack of conciseness and completeness of automated crawl models [?]. Yandrapally et al. [?] showed that state-of-the-art structural and visual SAF implementations produce near-duplicate states. Moreover, the study highlighted the challenge to select optimal thresholds to distinguish near-duplicate from distinct web pages.

Motivated by these findings, in this paper we propose a novel SAF based on the usage of neural embeddings, paired with machine learning classifiers that require no thresholds.

2.3 Neural Embeddings and Doc2Vec

Neural embeddings have shown to be useful for many code analysis tasks such as code completion [?], log statement generation [?], code review [?] and other code-related tasks [?], [?]. In this work, we evaluate whether embedding models produced by Doc2Vec [?], a popular document embedding technique, can be useful to target the equivalence problem between web pages, possibly with some adaptation and fine-tuning to take into account the semi-structured nature of HTML documents. We focus on Doc2Vec because it has been applied to compute embeddings for large corpora of textual data [?], document classification [?], sentiment analysis [?], [?], and disease detection studies [?]. However, its application to web testing is still unexplored. We hypothesize that Doc2Vec can produce meaningful embeddings also for HTML pages since their textual representation contains both tags and text.

Doc2Vec aims to find an optimal embedding model such that similar text documents would produce embeddings that lie close in the vector space. Given a document, Doc2Vec creates and projects paragraph embeddings, as well as word embeddings, into the vector space and then uses a trained deep neural network model to predict words of paragraphs or documents in a corpus [?]. Instead of computing an embedding for each word like Word2Vec [?], Doc2Vec creates a different embedding for an entire paragraph or even a document. At inference time, the input paragraph id vector (a one-hot encoded vector) is unknown, hence it is first derived by gradient descent given the input and output words and it is concatenated with the one-hot encoded vectors of the paragraph words to predict the next word in the paragraph. The internal representation used to make such a prediction is averaged or concatenated across predictions to get the final document embedding [?].

Doc2Vec can be configured to use two different models: Paragraph Vector Distributed Memory (PV-DM) or Distributed Bag Of Words (DBOW). The former randomly picks a set of consecutive words in the paragraph and tries to predict the word in the middle, using the surrounding words (i.e., context words) and the paragraph id. The latter is similar to a `Skip-gram` model, in which, given a paragraph id, the model tries to predict the next word of a randomly picked sequence of words from the chosen paragraph [?].

3 Approach

The goal of our approach, which we call WebEmbed, is to automatically detect the occurrence of near-duplicate web pages during crawling, discard them from the web app crawl model and generate test suites. In a nutshell, during crawling, our approach uses a novel neural embedding model for web pages built on top of Doc2Vec [?]. Initially proposed for textual documents, we explore its applicability to HTML web pages containing both tags and text. The DOM tags and the text tokens of the retrieved web app states are represented as vectors in an n-dimensional embedding space and used by a novel SAF to assess the similarity between web pages.

Figure 2 illustrates our approach, which consists of four phases, namely (1) Training Doc2Vec, (2) Training the State Abstraction Function, (3) Crawling, and (4) Test Creation.

In the first phase, different Doc2Vec models are trained on a unlabeled corpus of web pages. From each web page, our approach extracts different token-sequence representations of the DOM, namely its textual *content*, its *tags* or a combined representation of *content+tags*. Then, for each representation, a Doc2Vec model is trained. In the second phase, we use a labeled corpus of web pages in which

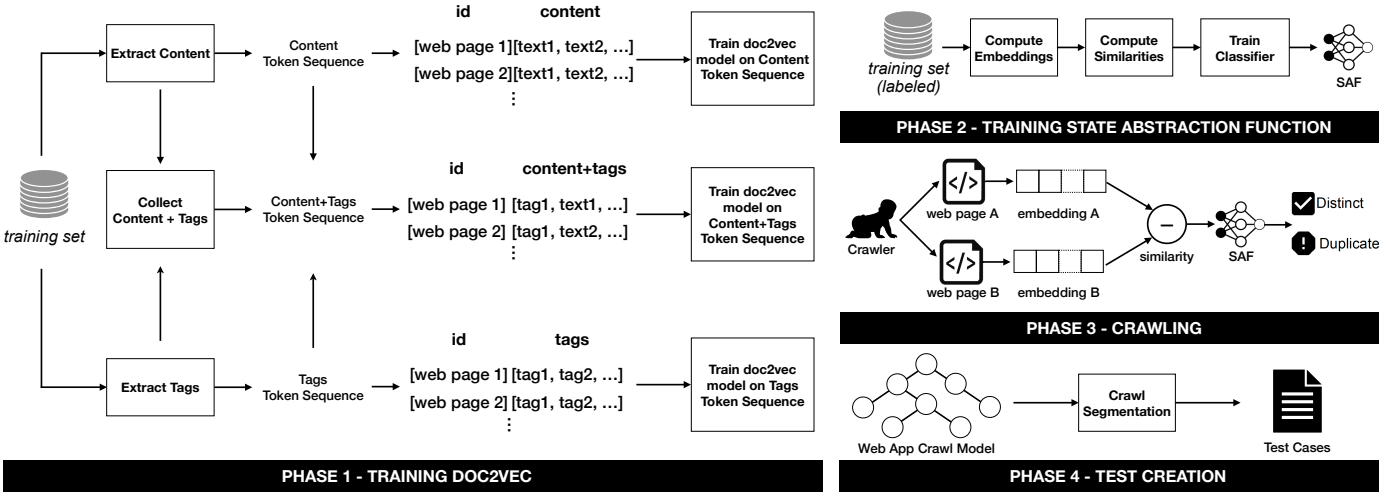

Fig. 2: Overview of the WEBEMBED approach

all pairs of web pages are annotated as being distinct or clone/near-duplicate. For each pair of web pages, we use the Doc2Vec models to compute their vector embeddings such that their similarity can be calculated. Once all similarities are computed, we train a classifier to distinguish clone/near-duplicate web pages based on such similarities. In the third phase, we use the Doc2Vec models at runtime during crawling as a SAF. When two web pages are available, our approach computes their embeddings based on a given representation (either content, tags, content+tags), computes their similarity, and uses the classifier to predict whether the two web pages are distinct or near-duplicate. In the fourth phase, each crawl path is turned into a web test case. We now detail each phase of our approach.

3.1 Training Doc2Vec

Our approach requires computing embeddings for HTML web pages. While originally conceived for general-purpose textual documents, in this work we extend Doc2Vec [**?**] to support web pages.

3.1.1 Token sequence extraction

The first phase requires an unlabeled corpus of HTML web pages, from which we extract a convenient representation for training a Doc2Vec model. Let us consider the HTML of the Detail page for Item A (Listing 1). We first retrieve a sequence of tokens of the DOM representing either its *content*, *tags* or *content+tags*. The procedure EXTRACTTOKENS, outlined in Algorithm 1 (lines 1–7), starts from the root node of the DOM and proceeds as follows: (1) the sequence of tokens (either tags, content, or content+tags) for the current node are extracted (line 3); (2) the extraction procedure is recursively called, in a depth-first fashion, on all children of the current node, from left to right. The result of these calls is then appended to the list of extracted tokens (lines 4–6); (3) the sequence of extracted tokens is returned (line 7).

Tags Token Sequence. The first extraction function considers only the tags of an HTML page while discarding comments, scripts, and CSS. The intuition is that tags indicate the general layout of an HTML document and may be effective for detecting structurally similar web pages [**?**].

```
1  <html lang="en">
2    <head>
3      <title>Item A detail page</title>
4      <link rel="stylesheet" href="styles.css">
5      <script type="text/javascript" src="utils.js"></script>
6    </head>
7    <body>
8      <img src="item_a.jpg" class="item_pic"/>
9      <h1>Item A</h1>
10     <img src="three-stars.png"/>
11     <p class="price">9.99 $</p>
12     <a href="/buy?i=item_a" class="btn">BUY</a>
13     <p class="descr">Detailed description for item A.</p>
14     <h2>Reviews</h2><!-- Reviews listed here -->
15     <a href="/addrev?i=item_a" class="btn">+ ADD REVIEW</a>
16     <table class="reviews">
17       <tr> <!-- First review -->
18         <td>Quite good by <a href="/u/alice">Alice</a></td>
19         <td><img src="four-stars.png"/></td>
20       </tr>
21       <tr> <!-- Second review -->
22         <td>Does its job by <a href="/u/bob">Bob</a></td>
23         <td><img src="four-stars.png"/></td>
24       </tr>
25     </table>
26   </body>
27   <script>add_review();</script>
28 </html>
```

Listing 1: HTML of Detail page for Item A. Tags are highlighted in boldface, content in blue.

Consequently, the tag token sequence of the web page in Listing 1 is: [`html`, `head`, `title`, `body`, `img`, `h1`, `img`, `p`, `a`, `p`, `h2`, `a`, `table`, `tr`, `td`, `a`, `td`, `img`, `tr`, `td`, `a`, `td`, `img`].

Content Token Sequence. The second extraction function only retrieves the textual content of a web page. Intuitively, two web pages sharing similar textual content have some degree of topical relatedness [**?**]. Consequently, the HTML in Listing 1 is converted to the following tokens of DOM content: [*item, a, detail, page, item, a, 9.99, $, buy, detailed, description, for, item, a, reviews, +, add, review, quite, good, by, alice, does, its, job, by, bob*].

Content+tags Token Sequence. The third extraction function considers both content and tags and combines the

Algorithm 1: Web App Crawling with WEBEMBED

```
1  Function EXTRACTTOKENS(n, et):        ▷ et: content, tags, or both
2      let tokens be an empty list;
3      tokens.append(getTokens(n, et));
4      foreach children node c of n, from left to right do
5          if c is not a script, style, or comment node then
6              tokens.append(EXTRACTTOKENS(c))
7          end
8      end
9      return tokens;
10 Function CRAWL(initial URL):
11     s1 ← getState(initial URL);
12     model ← initializeModel(s1);
13     while ¬ timeout do
14         next ← nextStateToExplore(model);
15         if next = nil then        ▷ app exhaustively explored
16             break;
17         end
18         s ← getToState(next);
19         for e ∈ getCandidateEvents(s) do
20             fireEvent(e);
21             sc ← current state after firing the event e;
22             if ¬ ISDUPLICATE(s, model) then
23                 add sc to model;
24             end
25         end
26     end
27     return model;
28 Function ISDUPLICATE(sc, model):
29     foreach state s′ in model do
30         if CLASSIFY(sc, s′) = 'clone' then
31             return True;          ▷ s is a duplicate of s′
32         end
33     end
34     return False;
35 Function CLASSIFY(p1, p2, ET):        ▷ ET: embedding types
36     let s be an empty list;
37     foreach embedding type et in ET do
38         r1 ← EXTRACTTOKENS(p1.getRootNode(), et);
39         r2 ← EXTRACTTOKENS(p2.getRootNode(), et);
40         doc2vec ← getDoc2VecModel(et);
41         e1 ← doc2vec.infer(r1) ;
42         e2 ← doc2vec.infer(r2);
43         s.append(cosineSimilarity(e1, e2));
44     end
45     return classifier.classify(s);
```

output of the two previous extraction functions. This can be effective in cases where using the tags or the content only is not enough to accurately classify two web pages. The HTML in Listing 1 is converted to the following content+tags token sequence: [`html`, `head`, `title`, *item*, *a*, *detail*, *page*, `link`, `body`, `img`, `h1`, *item*, *a*, `img`, `p`, *9.99*, *$*, `a`, *buy*, `p`, *detailed*, *description*, *for*, *item*, *a*, `h2`, *reviews*, `a`, *+*, *add*, *review*, `table`, `tr`, `td`, *quite*, *good*, *by*, `a`, *alice*, `td`, `img`, `tr`, `td`, *does*, *its*, *job*, *by*, `a`, *bob*, `td`, `img`].

In our empirical study, we evaluate the effectiveness of all three token sequences (tags, content, and content+tags) for near-duplicate web page detection.

3.1.2 Model Implementation and Training

Once the pre-processing for token sequence extraction is done, three different Doc2Vec models are trained, i.e., one model for each token-sequence type (using the DBOW model) [**?**]. Hence, we obtain three Doc2Vec models that allow us to compare pairs of web pages and compute their similarity based on one token-sequence representation of the pair at a time. For example, the following embeddings are produced for the HTML of Listing 1:

$$doc2vec(tags) = [-0.25, 0.48, ..., 0.03]$$
$$doc2vec(content) = [-0.55, 0.17, ..., 0.90]$$
$$doc2vec(content + tags) = [-0.40, 0.33, ..., 0.44]$$

3.2 Training State Abstraction Function

In the second phase, we train a SAF. This task requires a *labeled* corpus of web pages, in which each pair of web pages is manually labeled to indicate whether the web pages in the pair are clones/near-duplicates.

For each pair of web pages in such corpus, we use one of the different Doc2Vec models to compute their embeddings. Then, we compute the cosine similarity [**?**], a widely used metric to assess vector similarity. A combination of the three similarity scores, based on content, tags, or content+tags neural embeddings, is used to train a classifier to discriminate two web pages as being distinct or clones. In preliminary experiments, we also used the embeddings as input to the classifier without noticing any significant improvement. For performance reasons, we eventually used similarity scores, as a smaller input vector makes both the training and inference of the classifier faster. In particular, the inference time is critical as it directly impacts the time budget of the crawler.

3.3 Crawling

The third phase consists of using the trained SAF during crawling, to infer crawl models that can be used for automated test generation.

3.3.1 The Crawler

The crawler loads the web pages in a web browser and exercises client-side JavaScript code to simulate user-like interactions with the web app. This allows the crawler to support modern, client-side intensive, single-page web applications. The main conceptual steps performed when exploring a web application are outlined in the CRAWL function of Algorithm 1 (lines 8–20).

Crawling starts at an initial URL, the homepage is loaded into the browser and the initial DOM state, called index, is added to the model (line 10). Subsequently, the main loop (lines 11–19) is executed until the given time budget expires or there are no more states to visit (i.e., the web app has been exhaustively explored according to the crawler). In each iteration of the main loop, the first unvisited state in the model is selected (line 12), and the crawler puts in place adequate actions to reach said state. If the state cannot be reached directly, it retrieves the path from the index page and fires the events corresponding to each transition in the path. Upon reaching the unvisited state, the clickable web elements are collected (i.e., the web elements on which interaction is possible, line 16), and user events such as filling forms or clicking items are generated (line 17). After firing an event, the current DOM state s_c is captured (line 18). The ISDUPLICATE function supervises the construction of the model and checks whether s_c is a duplicate of an existing state (lines 22–26) by computing pairwise comparisons with all existing states in the model using the WEBEMBED SAF.

The state s_c is eventually added to the model if the SAF regards it as a distinct state, i.e., *a state that is not a duplicate of another existing state in the model* (lines 23–26). Otherwise, it is rejected and the crawler continues its exploration from the next available unvisited state until the timeout is reached.

3.3.2 Usage of the State Abstraction Function

The CLASSIFY procedure (lines 27–36) illustrates our neural-based SAF. Given two web pages p_1, p_2 and an embedding type ET, we first extract the token-sequence representations from each page based on the selected embedding types (ET can be any non empty subset of {*content*, *tags*, *content+tags*}), obtaining one list of tokens for each web page (lines 30–31). Each of the two token sequences r_1 and r_2 is then fed to the appropriate Doc2Vec model (line 32) to compute an embedding (lines 33–34). Then, the cosine similarity between the two resulting embeddings e_1 and e_2 is computed, obtaining a similarity score that is appended to the list s of similarities computed so far (line 35). Next, the classifier marks the two pages as either distinct or clones based on the list s of similarity scores and determines the SAF return value (line 36), which is '*clone*' in case of near-duplicate detection or '*distinct*' otherwise.

Example. Consider the following embeddings produced for our running example, for the embedding type '*tags*' (i.e., ET = ['*tags*']):

p_1 = Catalog Page $\quad e_1 = [-0.45, 0.56, ..., 0.30]$
p_2 = Detail Page A $\quad e_2 = [-0.55, 0.17, ..., 0.90]$
p_3 = Detail Page A + Review 1 $\quad e_3 = [-0.56, 0.19, ..., 0.95]$

During crawling, let us assume that a decision tree classifier flags a pair of pages as '*clone*' when the cosine similarity between their embeddings satisfies the root decision node condition ($s > 0.8$). If $sim(e_1, e_2) = 0.56$, p_2 is added to the model, as p_2 is not too similar to p_1. Then, when exploring p_3, we obtain $sim(e_3, e_1) = 0.58$ and $sim(e_2, e_3) = 0.95$. Hence, page p_3 is not added to the model as it is recognized as a near-duplicate ('*clone*') of p_2.

3.4 Test Creation

Our approach automatically generates a test suite during crawling through *segmentation* [?]. The crawl sequence of states is segmented into test cases when (1) the current DOM state no longer contains any candidate clickable elements to be fired and the crawler is reset to the index page; (2) no new states are present on the current path. In the case of Figure 1 (right), four (redundant) test cases are generated, one for each state representing the Detail page for item A. With WEBEMBED, the output model only has one state for the Detail page. Hence, only one test would be generated, reducing redundancy while keeping model and code coverage the same.

3.5 Implementation

We trained Doc2Vec models using the `gensim` [?] Python library and used the classifiers implementations available in the `scikit-learn` [?] Python library. We integrated WEBEMBED within Crawljax [?]. To automatically generate a test suite during crawling, we use the state-of-the-art DANTE web test generator [?]. DANTE generates fully compilable and functioning Selenium test cases [?] by segmenting a crawling session and by re-using the same inputs used during crawling.

TABLE 1: Web page characteristics across the datasets

		Web page metrics					
		DOM (# nodes)		Source (# chars)		Text content (# chars)	
Dataset	**# pages**	Mean	Std.	Mean	Std.	Mean	Std.
$\mathcal{DS}$	33,394	821	960	107,055	160,897	7,309	10,503
$\mathcal{RS}$	1,826	665	687	91,124	127,116	5,964	8,487
$\mathcal{CC}$	368,927	401	913	51,097	70,541	6,139	14,642
$\mathcal{SS}$	1,313	212	287	16,234	17,320	1,335	1,262

4 EMPIRICAL STUDY

4.1 Research Questions

To assess the practical benefits of neural embeddings for web testing, we consider the following research questions:

RQ$_1$ (near-duplicate detection). *How effective is* WEBEMBED *in distinguishing near-duplicate from distinct web app states?*
RQ$_2$ (model quality). *How do the web app models generated by* WEBEMBED *compare to a ground truth model?*
RQ$_3$ (code coverage). *What is the code coverage of the tests generated from* WEBEMBED *web app models?*

RQ$_1$ aims to assess what configuration of WEBEMBED, in terms of web embedding and classifier, is more effective at detecting near-duplicates through state-pair classification. RQ$_2$ focuses on the crawl model quality in terms of completeness and conciseness. RQ$_3$ evaluates WEBEMBED when used for web testing, specifically assessing the test suites generated by WEBEMBED crawl models in terms of code coverage of the web apps under test.

4.2 Datasets

We use three existing datasets available from the study by Yandrapally et al. [?], plus an additional dataset of web pages collected by the *Common Crawl* project [?]. Table 1 shows analytics information about the web pages of the considered datasets in terms of DOM size, length of the HTML source, and amount of text content.

The first dataset $\mathcal{DS}$ contains 493,088 state-pairs derived from automated crawls (using Crawljax [?]) of 1,031 randomly selected websites from the top one million provided by Alexa, a popular website that ranks sites based on their global popularity (dismissed as of May 1, 2022). For training Doc2Vec, we used an additional dataset (listed third in Table 1) of 368,927 web pages available from the *Common Crawl* project [?], also used in previous research [?]. We refer to this dataset as $\mathcal{CC}$. Similarly to $\mathcal{DS}$, the web pages in $\mathcal{CC}$ are also collected by crawling real-world websites.

The second dataset in Table 1, referred to as $\mathcal{RS}$, contains 1,000 state-pairs from $\mathcal{DS}$ that Yandrapally et al. [?] manually labeled as either clone, near-duplicate or distinct.

The fourth dataset $\mathcal{SS}$ contains 97,500 state-pairs of nine subject apps (Table 2), which were also manually labeled by Yandrapally et al. [?] as clone, near-duplicate or distinct.

TABLE 2: Subject Set with Manual Classification

	Logical States	Concrete States	Redundancy (%)	State-pairs	Distinct	Clones and Near-duplicates
App_1	25	131	524	8,515	6,142	2,373
App_2	36	189	525	17,766	14,988	2,778
App_3	23	99	430	4,851	432	531
App_4	14	151	1,079	11,325	7,254	4,071
App_5	53	151	285	11,325	10,206	119
App_6	21	153	729	11,628	10,683	945
App_7	20	140	700	9,730	5,782	3,948
App_8	10	150	1,500	11,175	6,569	4,606
App_9	14	149	1,064	11,175	9,411	1,615

These nine web apps (Table 2) have been used as subjects in previous research on web testing [?], [?], [?], [?]. Despite being developed with different frameworks, they all provide CRUD functionalities (e.g., login, or add user) which make them functionally similar. Five apps are open-source PHP-based applications, namely Addressbook (App_1, *v. 8.2.5*) [?], Claroline (App_2, *v. 1.11.10*) [?], PPMA (App_3, *v. 0.6.0*) [?], MRBS (App_4, *v. 1.4.9*) [?] and MantisBT (App_5, *v. 1.1.8*) [?]. Four are JavaScript single-page applications—Dimeshift (App_6, *commit 261166d*) [?], Pagekit (App_7, *v. 1.0.16*) [?], Phoenix (App_8, *v. 1.1.0*) [?] and PetClinic (App_9, *commit 6010d5*) [?]—developed using popular JavaScript frameworks such as *Backbone.js*, *Vue.js*, *Phoenix/React* and *AngularJS*.

4.3 Baselines

Based on the study by Yandrapally et al. [?], we selected two algorithms as baselines for WEBEMBED, one structural and one visual. The structural algorithm is RTED (Robust Tree Edit Distance) [?], a DOM tree edit distance algorithm. The visual algorithm is PDiff [?], which compares two web page screenshots based on a human-like concept of similarity that uses spatial, luminance, and color sensitivity. We chose them as baselines for WEBEMBED for the following reasons: (1) they were the best structural and visual algorithms for near-duplicate detection [?], (2) they were used as a SAF for web testing purposes within Crawljax.

4.4 Use Cases

To evaluate the effectiveness of WEBEMBED, we consider three use cases, summarized in Table 3. For all use cases, WEBEMBED relies on the embeddings computed by a common Doc2Vec model trained on the non-annotated pages of the considered datasets, namely $\mathcal{DS} \cup \mathcal{CC}$. The differences among the use cases are the datasets used to train the WEBEMBED classifiers and the associated labeling cost for developers. To avoid confounding factors, we used the tool `difflib` to assess the presence of cloned pages across datasets and results indicate that no such clones exist.

4.4.1 Beyond apps

This use case aims at investigating the feasibility of a general-purpose model trained on web pages that are different from the ones it is tested on. Therefore, we train the WEBEMBED classifiers on $\mathcal{RS}$ and test them on $\mathcal{SS}$. This use case requires *no labeling costs* to web developers, as the classifier we train on $\mathcal{RS}$ is supposed to be re-used as-is on any new web app.

TABLE 3: Use cases and variants of WEBEMBED

	WEBEMBED		
	Doc2Vec	Classifiers	
Use case	**Train Set**	**Train Set**	**Test Set**
Beyond apps	$\mathcal{DS} \cup \mathcal{CC}$	$\mathcal{RS}$	$\mathcal{SS}$
Across apps (for each App_i)	$\mathcal{DS} \cup \mathcal{CC}$	$\mathcal{SS} \setminus App_i$	App_i
Within apps (for each App_i)	$\mathcal{DS} \cup \mathcal{CC}$	80% App_i	20% App_i

4.4.2 Across apps

This use case investigates the generalizability of WEBEMBED when applied to web apps similar to the ones the classifier was trained on (similarity refers to having analogous CRUD functionalities, see Section 4.2). Indeed, we train a classifier for each of the nine web apps in $\mathcal{SS}$ in a leave-one-out fashion. The training set considers the annotated state-pairs of eight web apps, using the ninth web app as a test set. We iteratively vary the test web app until all nine subject apps are accounted for. In this use case, developers are supposed to find and manually label all pages of web apps similar to the ones under test. A company may develop a few web apps in a given domain, investing in manual labeling of the near-duplicates of such apps to save the near-duplicate detection effort later, when a new app will be developed in the same domain.

4.4.3 Within apps

We train an app-specific classifier for each of the nine web apps. For each app in $\mathcal{SS}$, we use 80% of the state pairs for training the classifier and the remaining 20% for testing. In this use case, developers are required to label a significant portion of the near-duplicate pages of the web app under test before a classifier can be trained and applied to the other pages of the same web app.

4.5 Procedure and Metrics

4.5.1 RQ_1 (near-duplicate detection)

For each use case of WEBEMBED (Section 4.4), we evaluate different WEBEMBED implementations, varying (1) the token sequence used to train Doc2Vec and (2) the classifier used to enable the SAF. Concerning the token sequences, we trained three different Doc2Vec models, one for each representation of the pages in the dataset $\mathcal{DS} \cup \mathcal{CC}$ (tags, content, content+tags). Concerning the training hyperparameters, we used the default parameters of the `gensim` [?] Python library and fitted the models for 100 epochs using a vector size of 100.

Concerning the classifiers, we evaluate a total of eight classifiers. We consider six machine learning classifiers from the `scikit-learn` [?] Python library, namely Decision Tree, Nearest Neighbour, SVM, Naïve Bayes, Random Forest, and Multi-layer Perceptron. We also consider their ensemble with majority voting and an additional threshold-based classifier.

The quality of near duplicate detection is measured using accuracy, precision, recall, and F_1, where the last three metrics are computed under the assumption that the positive class (output 1 of the classifier) is 'near-duplicate' ('distinct' being the negative class). Overall, we evaluate 456 WEBEMBED configurations (8 classifiers $\times$ 3 token sequences $\times$ 19 configurations, one for Beyond apps, nine for Across apps, and nine for Within apps).

4.5.2 RQ_2 (model quality)

The crawl models contain redundant concrete states that Yandrapally et al. [?] aggregated into the corresponding logical pages. Logical pages represent clusters of concrete pages that are semantically the same. To measure WEBEMBED's model quality w.r.t. the ground truth, we compute the precision, recall, and F_1 scores, considering the intra-pairs (IP) in common in the given model and the intra-pairs within each manually identified logical page (GT) [?]:

$$p = \frac{|IP_{GT} \cap IP_{\text{WEBEMBED}}|}{|IP_{\text{WEBEMBED}}|} \qquad r = \frac{|IP_{GT} \cap IP_{\text{WEBEMBED}}|}{|IP_{GT}|}$$

We also consider the F_1 score as the harmonic mean of (intra-pair) precision and recall. As an example, let us consider a set of 6 web pages $\{p_1, p_2, p_3, p_4, p_5, p_6\}$ with the following ground truth (GT) assignment: $\{p_1, p_2\}, \{p_3\}, \{p_4, p_5, p_6\}$. Suppose WEBEMBED produces the following assignment: $\{p_1, p_3\}, \{p_2\}, \{p_4, p_5\}, \{p_6\}$. The intra-pairs for GT are $\langle p_1, p_2\rangle$, $\langle p_4, p_5\rangle$, $\langle p_4, p_6\rangle$, $\langle p_5, p_6\rangle$, whereas the intra-pairs for WEBEMBED are $\langle p_1, p_3\rangle$, $\langle p_4, p_5\rangle$. Thus, $p = |\langle p4, p5\rangle|/|\langle p1, p3\rangle, \langle p4, p5\rangle| = 0.5$, $r = |\langle p4, p5\rangle|/|\langle p1, p2\rangle, \langle p4, p5\rangle, \langle p4, p6\rangle, \langle p5, p6\rangle| = 0.25$, and $F_1 = 2pr/(p + r) = 0.32$.

4.5.3 RQ_3 (code coverage)

To assess the effectiveness of WEBEMBED when used for web testing, we crawl each web application in $\mathcal{SS}$ multiple times, each time varying the SAF. For all tools and all use cases, we set the same crawling time of 30 minutes. We use DANTE to generate Selenium web test cases from the crawl sequences, execute the tests, and measure the web app code coverage. For JavaScript-based apps (Dimeshift, Pagekit, Phoenix, PetClinic), we measure *client-side* code coverage using `cdp4j` (v. 3.0.8) library, i.e., the Java implementation of Chrome DevTools. For PHP-based apps (Claroline, Addressbook, PPMA, MRBS, MantisBT), we measure the *server-side* code coverage using the `xdebug` (v. 2.2.4) PHP extension and the `php-code-coverage` (v. 2.2.3) library. We addressed randomness in our experiments by manually adding delays where appropriate in the web test suites, in order to mitigate flaky executions. Before measuring coverage, we executed each test suite three times to ensure comparable outcomes across executions of different test suites.

We assess the statistical significance of the differences between WEBEMBED and the baselines using the non-parametric Mann-Whitney U test [?] (with $\alpha = 0.05$) and the magnitude of the differences, if any, using Cohen's d effect size [?].

TABLE 4: RQ_1 Near-Duplicate Detection. The best average values of accuracy and F_1 are boldfaced.

	Beyond Apps				Across Apps				Within Apps			
Embed.	Acc.	Pr.	Rec.	F_1	Acc.	Pr.	Rec.	F_1	Acc.	Pr.	Rec.	F_1
content	0.73	0.97	0.67	0.79	0.82	0.89	0.88	0.87	0.91	0.92	0.95	0.93
tags	0.75	0.98	0.70	0.81	0.79	0.88	0.86	0.84	0.85	0.91	0.87	0.88
content + tags	**0.75**	0.97	0.70	0.81	**0.83**	0.89	0.90	**0.87**	**0.93**	0.94	0.97	**0.95**
RTED	0.75	0.86	0.81	**0.83**	0.77	0.85	0.86	0.80	0.84	0.92	0.85	0.86
PDiff	0.48	0.81	0.43	0.56	0.74	0.86	0.80	0.83	0.86	0.87	0.97	0.91

4.6 Results

4.6.1 RQ_1 (near-duplicate detection)

Table 4 shows the results for the tools being compared on the task of near-duplicate detection. Due to space reasons, we only present the scores of WEBEMBED when using the SVM classifier, which showed to be the best in our experiments across all use cases. For the Across apps and Within apps use cases, we present the scores averaged over all nine apps. All results are available in our replication package [?].

For each technique being compared, Table 4 shows average accuracy (Acc.), precision (Pr.), recall (Rec.), and F_1 scores, divided by use case. The scores for the baselines RTED and PDiff are also reported. In the Beyond apps use case, WEBEMBED and RTED have similar accuracy (resp. F_1) values, whereas WEBEMBED has a +56% (resp. +44%) increase w.r.t. PDiff. For the Across apps use case, WEBEMBED scores higher accuracy and F_1 w.r.t. the baselines (e.g., for accuracy, +8% increase w.r.t. RTED and +12% w.r.t. PDiff). In the Within apps use case, WEBEMBED scores higher accuracy and F_1 than the baseline approaches as well (e.g., for accuracy, +11% increase w.r.t. RTED and +8% increase w.r.t. PDiff).

Statistical tests confirmed that the differences in accuracy and F_1 between WEBEMBED and the best baseline (either RTED or PDiff, depending on the use case) are statistically significant (p-value < 0.05) with a *large* effect size in both the Across and Within apps use cases.

> **RQ_1**: WEBEMBED *achieves the highest accuracy scores (75–93%, on average) over all use cases when configured with content+tags embeddings and SVM classifier. The differences w.r.t. the baseline approaches are statistically significant in two out of three use cases.*

4.6.2 RQ_2 (model quality)

Table 5 shows, for each web app, intra-pairs precision (Pr.), intra-pairs recall (Rec.), and intra-pairs F_1 scores for all competing techniques, divided by use case. For WEBEMBED, we present the results for the best configuration resulting from RQ_1 (content+tags embeddings and SVM classifier).

Overall, WEBEMBED produces more accurate models (i.e., models more similar to the ground truth) than the competing techniques across all use cases, as summarized by the intra-pairs F_1 scores.

In the Beyond apps use case, WEBEMBED scores +18% and +37% average F_1 w.r.t. RTED and PDiff, respectively. In the Within apps use case, WEBEMBED scores +21% and

TABLE 5: RQ_2 Model Coverage. Best average F_1 scores are highlighted in bold.

	Beyond Apps									Across Apps									Within Apps								
	WEBEMBED			RTED			PDiff			WEBEMBED			RTED			PDiff			WEBEMBED			RTED			PDiff		
	Pr.	Rec.	F_1	Pr.	Rec.	F_1	Pr.	Rec.	F_1	Pr.	Rec.	F_1	Pr.	Rec.	F_1	Pr.	Rec.	F_1	Pr.	Rec.	F_1	Pr.	Rec.	F_1	Pr.	Rec.	F_1
App_1	0.89	0.95	0.92	0.68	0.72	0.70	0.55	0.58	0.57	0.89	0.95	0.92	0.69	0.73	0.70	0.86	0.91	0.88	0.84	0.89	0.86	0.15	0.16	0.15	0.27	0.28	0.28
App_2	0.93	1.00	0.97	0.93	1.00	0.97	0.93	1.00	0.96	0.93	1.00	0.97	0.93	1.00	0.97	0.93	1.00	0.96	0.93	1.00	0.97	0.93	1.00	0.97	0.93	1.00	0.97
App_3	0.82	1.00	0.90	0.82	1.00	0.90	0.82	1.00	0.90	0.79	0.96	0.87	0.82	1.00	0.90	0.82	1.00	0.90	0.82	1.00	0.90	0.82	1.00	0.90	0.82	1.00	0.90
App_4	0.94	0.98	0.96	0.92	0.96	0.94	0.56	0.58	0.57	0.92	0.96	0.94	0.92	0.96	0.94	0.50	0.52	0.51	0.96	1.00	0.98	0.92	0.96	0.94	0.71	0.73	0.72
App_5	0.84	0.97	0.90	0.58	0.67	0.62	0.75	0.86	0.80	0.81	0.93	0.87	0.58	0.67	0.62	0.75	0.86	0.80	0.87	1.00	0.93	0.87	1.00	0.93	0.87	1.00	0.93
App_6	0.84	1.00	0.91	0.81	0.96	0.88	0.56	0.67	0.61	0.76	0.90	0.82	0.81	0.96	0.88	0.56	0.67	0.61	0.84	1.00	0.91	0.83	0.99	0.90	0.62	0.73	0.67
App_7	0.56	0.58	0.57	0.21	0.21	0.21	0.21	0.21	0.21	0.82	0.84	0.83	0.21	0.21	0.21	0.21	0.21	0.21	0.95	0.98	0.96	0.30	0.31	0.30	0.36	0.38	0.37
App_8	0.71	0.73	0.72	0.45	0.47	0.46	0.33	0.34	0.34	0.71	0.73	0.72	0.45	0.47	0.46	0.33	0.34	0.34	0.80	0.83	0.81	0.15	0.16	0.15	0.47	0.48	0.48
App_9	0.72	0.79	0.75	0.70	0.77	0.73	0.53	0.58	0.55	0.77	0.85	0.81	0.70	0.77	0.73	0.53	0.58	0.55	0.91	1.00	0.95	0.82	0.90	0.86	0.91	1.00	0.95
Avg.	0.81	0.89	**0.84**	0.68	0.75	0.71	0.58	0.65	0.61	0.82	0.90	**0.86**	0.68	0.75	0.71	0.61	0.68	0.64	0.88	0.97	**0.92**	0.64	0.72	0.68	0.66	0.73	0.70

+34% average F_1 w.r.t. RTED and PDiff, respectively. In the Across apps use case, WEBEMBED scores an average F_1 of 92%, a +35%, and +31% increase w.r.t. RTED and PDiff, respectively. Statistical tests confirmed that the differences in accuracy are statistically significant (p-value < 0.05) with a *large* effect size in all use cases, except Across apps, in which the differences between WEBEMBED and RTED are statistically significant with a *medium* effect size.

> **RQ_2**: WEBEMBED *achieves the highest F_1 scores (84–92%, on average) over all use cases: neural embeddings are able to approximate the ground truth model better than structural and visual techniques. The differences with the baseline approaches are statistically significant in all use cases, with a medium to large effect size.*

4.6.3 RQ_3 (code coverage)

Table 6 shows the code coverage results for each tool, grouped by use case. Considering the average scores over all nine apps, the scores for WEBEMBED (WE) are consistently the best across all use cases.

For the Beyond Apps use case, WEBEMBED achieves +6-14% code coverage w.r.t. RTED and PDiff. Concerning the Across Apps use case, WEBEMBED achieves +12–13% code coverage w.r.t. RTED and PDiff. About the Within Apps use case, WEBEMBED achieves +20–36% code coverage w.r.t. RTED and PDiff. The differences in code coverage between WEBEMBED and PDiff are statistically significant for all use cases (i.e., p-value < 0.05, with *small/negligible/medium* effect sizes). The differences in code coverage between WEBEMBED and RTED are significant only for the Within App use case, with a *small* effect size.

> **RQ_3**: *The tests generated from* WEBEMBED *crawl models achieve the highest code coverage scores over all use cases (up to +36% improvement) thanks to the more accurate and complete web app models generated using neural embeddings.*

TABLE 6: RQ_3 Code Coverage. The best average scores are boldfaced.

	Beyond Apps			Across Apps			Within Apps		
	WE	RTED	PDiff	WE	RTED	PDiff	WE	RTED	PDiff
App_1	14.54	13.76	10.23	14.68	14.78	13.94	15.19	14.06	14.06
App_2	12.89	12.49	3.50	13.27	7.90	4.71	28.81	23.87	5.45
App_3	19.04	15.66	18.39	19.18	15.75	17.44	34.97	22.27	23.08
App_4	16.19	10.50	8.75	17.55	10.50	8.75	18.94	17.49	9.17
App_5	18.13	13.52	15.80	22.65	15.79	18.36	26.60	16.76	18.35
App_6	15.27	15.27	13.78	13.46	13.78	13.78	13.46	13.78	13.78
App_7	56.31	56.11	54.13	56.51	56.11	55.57	58.11	56.98	55.67
App_8	28.78	28.78	28.78	29.06	28.78	28.78	45.51	30.74	30.74
App_9	31.63	32.28	31.94	31.94	32.11	32.11	33.44	32.42	32.11
Avg.	**23.64**	22.04	20.59	**24.26**	21.72	21.49	**30.56**	25.38	22.49

4.7 Final Remarks

Overall, WEBEMBED was more effective than the considered baseline approaches across all use cases. From a practical point of view, looking at the accuracy scores in conjunction with code coverage, we suggest: (1) using WEBEMBED (Beyond apps) if no labeling budget is allowed for developers. Indeed, the effectiveness of this configuration is close to WEBEMBED (Across apps), which instead requires a non-negligible labeling cost. (2) using WEBEMBED (Within apps) in all other cases, especially if the labeling cost is affordable. Indeed, the gain in code coverage is significant (+29–26% w.r.t. the Beyond and Across apps use cases).

4.8 Threats to Validity

4.8.1 Internal validity

We compared all variants of WEBEMBED and baselines under identical experimental settings and on the same evaluation set (Section 4.2). In our experiments, a crawling time of 30 mins allowed all crawls to explore all logical pages of the AUTs within the timeout. Setting a shorter crawling time (<30mins) would favor the techniques that make better use of a limited crawling budget (i.e., WebEmbed and RTED). The test generation budget refers to the crawling time allowed for model inference, as the tests are extracted directly from the crawl sequences. The main threat to internal validity concerns our implementation of

the testing scripts to evaluate the results, which we tested thoroughly.

4.8.2 External validity

The limited number of subjects in our evaluation poses a threat in terms of the generalizability of our results to other web apps. Moreover, we considered only the embeddings produced by Doc2Vec [**?**], and WEBEMBED's effectiveness may change when considering other algorithms.

4.8.3 Reproducibility

All our results, the source code of WEBEMBED, and all subjects are available [**?**].

5 DISCUSSION

6 RELATED WORK

6.1 End-to-End Web Test Automation

Andrews et al. [**?**] propose a test generation approach based on a hierarchical finite state machine model to achieve full transition coverage. Biagiola et al. [**?**], [**?**] use Page Objects to guide the generation of tests. Marchetto et al. [**?**] propose a combination of static and dynamic analysis to model the AUT into a finite state machine and generate tests based on multiple coverage criteria.

Mesbah et al. [**?**] propose ATUSA, a tool that leverages the model of the AUT produced by Crawljax to automatically generate test cases to cover all the transitions of the model. Biagiola et al. [**?**] propose DANTE, an automated approach to test generation, aimed at producing minimal test suites from web app crawlings. DANTE turns the raw output of a crawler into executable test cases by reusing the same inputs used upon crawling, resolving dependencies among tests and eliminating redundant tests. Sunman et al. [**?**] propose AWET, an approach that leverages existing test cases, possibly obtained via capture-and-replay in an exploratory testing fashion, to guide crawling.

These works do not address the redundancy in the web app model during crawling due to an ineffective SAF. These test generators can be used in conjunction with WEBEMBED, to increase the accuracy of the inferred web app models.

6.2 Empirical Studies on Near-Duplicates

Fetterly et al. [**?**] study the nature of near-duplicates during software evolution, reporting their low variability over time. Yandrapally et al. [**?**] compares different near-duplication detection algorithms as SAFs in a web crawler. The paper reports on the impossibility of finding an optimal threshold that can accurately detect functional near-duplicates across web apps. Motivated by these findings, in our paper, we use ML classifiers instead of threshold-based classifiers. Moreover, we adopt neural embeddings applied to web pages and use the best detection algorithms from the study by Yandrapally et al. [**?**].

6.3 Automated Near-Duplicate Detection

Regarding detection of near-duplicates *within* the same AUT, Crescenzi et al. [**?**] propose a structural abstraction for web pages and a clustering algorithm based on such abstraction. Di Lucca et al. [**?**], [**?**] evaluate the Levenshtein distance and the tag frequency for detecting near-duplicate web pages. Stocco et al. [**?**] use clustering on structural features as a post-processing technique to discard near-duplicates in crawl models. Corazza et al. [**?**] propose the usage of tree kernels, functions that compute the similarity between tree-structured objects, to detect near-duplicates.

Concerning detection of near-duplicates *across* AUTs, researchers mainly considered clustering techniques on raw structural features [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. Other works, such as the one by Henzinger [**?**], use shingles, i.e., n-grams composed of contiguous subsequences of tokens, to ascertain the similarity between web pages. Manku et al. [**?**] use `simhash` to detect near-duplicates in the context of information retrieval, plagiarism, and spam detection. Yandrapally and Mesbah [**?**] use web page fragments in which they combine both visual and structural features to detect near-duplicates.

In this paper, we consider HTML neural embeddings to train an ML classifier for near-duplicate detection and we illustrate that its usage for functional testing of web apps outperforms state-of-the-art techniques [**?**].

6.4 Embeddings in Software Engineering

Alon et al. [**?**] present code2vec, a neural model for learning embeddings for source code, based on its representation as a set of paths in the abstract syntax tree. Hoang et al. [**?**] propose CC2Vec, a neural network model that learns distributed representations of code changes. The model is applied for log message generation, bug fixing patch identification, and just-in-time defect prediction.

Feng et al. [**?**] use representation learning applied across web apps for phishing detection. Similarly, we use embeddings produced by Doc2Vec on HTML features to learn a neural representation of the web pages, both beyond, across, and within web apps. Lugeon et al. [**?**] propose Homepage2Vec, an embedding method for website classification. Namavar et al. [**?**] performed a large-scale experiment comparing different code representations to aid bug repair tasks. In this work, we propose an embedding method that works at a finer granularity level and that can integrate both structural (HTML tags) and textual (content) information. We study this embedding in the context of automated crawling and web testing.

Among the grey literature, Ma et al. [**?**] propose GraphCode2Vec, a technique that joins code analysis and graph neural networks to learn lexically and program dependent features to support method name prediction. Dakhel et al. [**?**] propose dev2vec, an approach to embed developers' domain expertise within vectors for the automated assessment of developers' specialization. Jabbar et al. [**?**] propose to encode the test execution traces for test prioritization.

Differently, we use the embeddings of Doc2Vec to train an ML classifier that is used as SAF within a crawl-based test generator for functional testing.

7 Conclusions and Future Work

In this paper, we aim to improve the crawlability of modern web applications by designing and evaluating WebEmbed, a novel state abstraction function for web testing based on neural embeddings of web pages. Neural embeddings are used to train machine learning classifiers for near-duplicate detection. We demonstrate their effectiveness in inferring accurate models for functional testing of web apps, while also discussing their cost for developers in three settings, namely beyond, across and within web apps. Our results show that crawl models produced with WebEmbed have higher precision and recall than the ones produced with existing approaches. Moreover, these models allow test suites generated from them to achieve higher code coverage.

Future work includes exploring other forms of embeddings to further improve the accuracy of WebEmbed. For example, usage of visual embeddings on the web screenshots, e.g., with autoencoders, will be explored, as well as hybrid solutions.

Acknowledgments

This work was partially supported by the H2020 project PRECRIME, funded under the ERC Advanced Grant 2017 Program (ERC Grant Agreement n. 787703).