Neural Embeddings for Web Testing

Kasun Kanaththage[1], Luigi Libero Lucio Starace[2], Matteo Biagiola[3,4], Paolo Tonella[4], Andrea Stocco[1]
[1]*Technical University of Munich*, Munich, Germany
{kasun.kanaththage, andrea.stocco}@tum.de
[2]*Università degli Studi di Napoli Federico II*, Napoli, Italy
luigiliberolucio.starace@unina.it
[3]*University of St. Gallen*, St. Gallen, Switzerland
matteo.biagiola@usi.ch
[4]*Università della Svizzera italiana*, Lugano, Switzerland
{matteo.biagiola, paolo.tonella}@usi.ch

***Abstract*—Web test automation techniques often rely on crawlers to infer models of web applications for automated test generation. However, current crawlers rely on state equivalence algorithms that struggle to distinguish near-duplicate pages, often leading to redundant test cases and incomplete coverage of application functionality. In this paper, we present a model-based test generation approach that employs transformer-based Siamese neural networks (SNNs) to infer web application models more accurately. By learning similarity-based representations, SNNs capture structural and textual relationships among web pages, improving near-duplicate detection during crawling and enhancing the quality of inferred models, and thus, the effectiveness of generated test suites. Our evaluation across nine web apps shows that SNNs outperform state-of-the-art techniques in near-duplicate detection, resulting in superior web app models with an average F_1 score improvement of 56%. These enhanced models enable the generation of more effective test suites that achieve higher code coverage, with improvements ranging from 6% to 21% and averaging at 12%.**

I. Introduction

Test automation enables end-to-end (E2E) functional testing of web applications by simulating user interactions through automated scripts [1], [2], [3]. The manual development of E2E test automation scripts is notoriously costly and time-consuming in practice [4]. To address this, researchers have proposed automated test generation techniques, many of which rely on constructing state-based web application models [5], [6], [7], [8], [9], [10]. These model-based web testing approaches systematically explore a web application using a crawler to infer a model that captures its functional behavior in terms of states (logical web pages, representing different functionalities) and transitions between them, triggered by user interactions such as clicks [11], [10].

Effective models should be both *complete* (i.e., capture all distinct logical pages, adequately covering web app functionality) and *concise* (i.e., avoid redundant representations of the same states) [12], [13]. In practice, however, models obtained via crawling are often affected by *near-duplicate* states, i.e., replicas of the same logical page differing only by minor changes [12].

Near-duplicates negatively affect the model's conciseness, because the same logical state is represented multiple times, and completeness, because the crawler, operating under a finite exploration budget, spends resources revisiting redundant states instead of discovering new ones. This has negative effects on test generation: test suites derived from such models tend to be less effective [12], [13]. Undiscovered states remain untested, reducing adequacy measures such as code coverage, while redundant test cases targeting duplicate states offer little to no contribution in expanding behavioral coverage [7].

To reduce near-duplicate states, model-based testing relies on a state abstraction function (SAF) to identify and discard redundant states during crawling. Existing SAFs use Document Object Model (DOM), visual, or hybrid similarity metrics, but studies show their effectiveness is highly application-dependent, making it challenging to design a universally reliable SAF [12].

This paper introduces WebEmbed, a novel SAF that combines neural embeddings with transformer-based Siamese neural networks (SNNs). Our approach encodes each web page into an n-dimensional embedding derived from token sequences of its markup and textual content. These embeddings are used to train SNN classifiers with distance-based losses (e.g., contrastive or triplet loss), enabling the model to learn semantic similarity within a shared embedding space. Unlike traditional classifiers, SNNs learn a general similarity function rather than relying on fixed class boundaries, allowing them to generalize to unseen states, an important property for web model inference, where labeled data and balanced class distributions are often unavailable. During crawling, each newly encountered page is embedded and compared to existing states; the classifier then maps its similarity score to near-duplicate or distinct, ensuring that only new states are retained in the inferred model.

We empirically evaluate WebEmbed using established web application benchmarks covering a diverse set of domains. Three tasks are considered: near-duplicate detection, model coverage, and test generation. Specifically, we assess four embedding models—Doc2Vec [14], BERT [15], ModernBERT [16], and MarkupLM [17]—and train two SNN variants per model using Binary Cross-Entropy (BCE) [18] and Triplet Loss [19], [20]. Across all tasks, SNNs trained with Doc2Vec embeddings and BCE loss consistently achieve the best performance, with statistically significant improvements

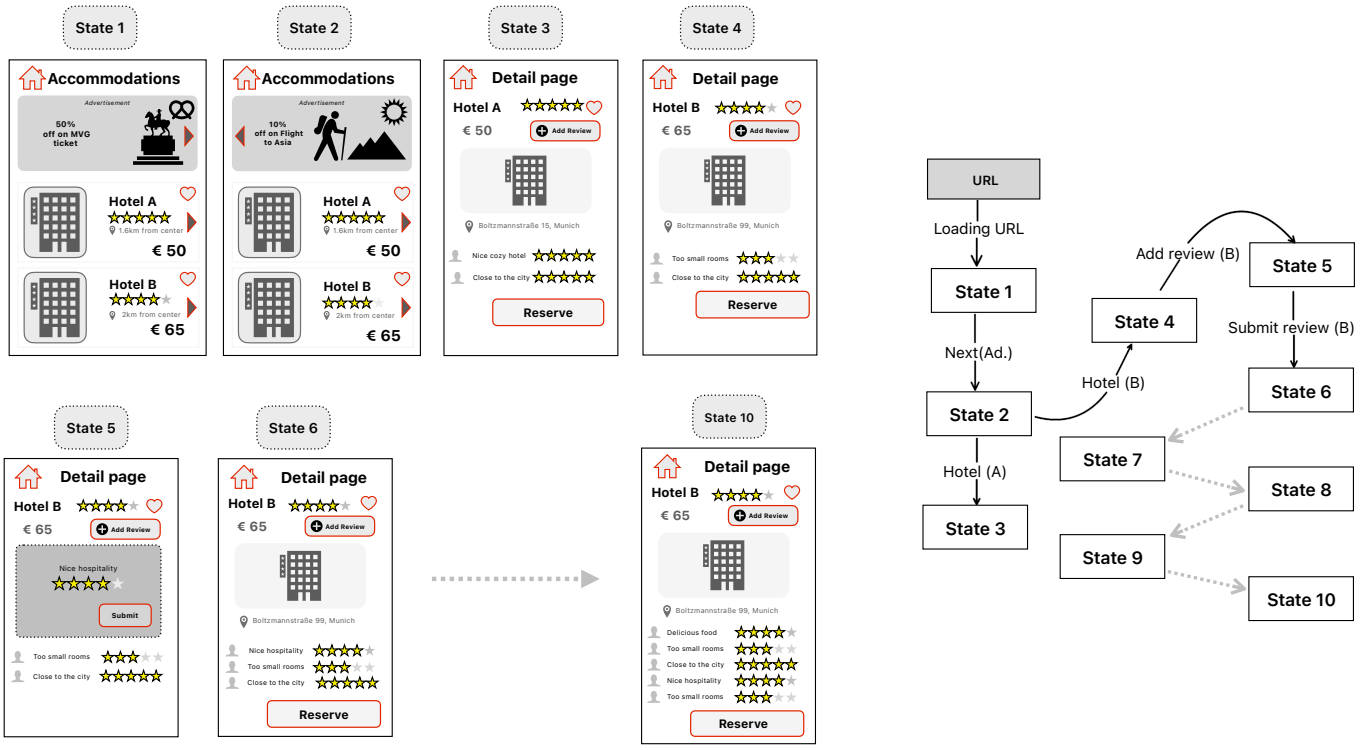

Fig. 1: Left: Hotel reservation web app: app states with actionable outlined. Right: An inferred model with near-duplicates.

over four state-of-the-art SAFs. Notably, this configuration attains an F_1 score of 96% in near-duplicate detection (a 45% improvement) and 81% in model coverage (a 57% improvement). Furthermore, tests generated from SNN-based SAF models yield up to 21% higher code coverage across applications, with an average increase of 12% compared to current leading techniques.

Our paper makes the following contributions:

Technique. A novel approach called WEBEMBED that uses neural embeddings and Siamese neural networks for web model inference and testing, which is available [21].

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

II. BACKGROUND

A. Automated Web Model Inference

Automated web model inference techniques, such as crawling, construct structural models of web applications by systematically exploring their GUI states. This process involves triggering user-driven events (e.g., clicks) and generating input values that cause the application to transition between different pages or views. When a significant change in the rendered content is observed, a new abstract state is added to the inferred model. Each state serves as a generalization of all dynamic runtime instances of a logical page, typically characterized by their underlying DOMs. The resulting model comprises a set of abstract states representing logical web pages and a set of transitions (edges) that capture possible navigations or interactions connecting them.

Consider a simple hotel reservation web application that displays available accommodations for the reservation. Users can view hotel details, add reviews, and make reservations (Figure 1, left). They can navigate from the Listings page to the Details page by clicking on a hotel item. On the Details page, users have the option to write a review or proceed with making a reservation. Additionally, users can return to the Details page from any other page by clicking the Home icon. After exploring these states, the model is generated by constructing a graph that records the reached states as nodes and the transitions as the edges of the graph (Figure 1, right).

B. Near-Duplicate States

During exploration, newly encountered states fall into three categories. Distinct states exhibit at least one functional or semantic difference from previously explored states (e.g., State 2 vs. State 3) and must be added to the model. Clone states are exact replicas with no functional or semantic differences and are therefore discarded. The most challenging category, Near-duplicate states, are functionally similar pages whose minor variations do not affect overall behavior [13]. Identifying such states is essential, as they add limited testing value. Indeed, web crawlers can optimize their exploration process by skipping near-duplicate states and prioritizing diverse states, leading to inferred models that are more representative and less redundant.

Existing work on near-duplicate detection [12] distinguishes three classes: *Cosmetic* (ND_1), involving purely visual changes that do not affect functionality; *Dynamic Data* (ND_2), where pages share the same template but differ in dynamically populated content; and *Structural* (ND_3), where one page contains additional elements whose functionality and semantics are fully represented in the other.

Prior approaches for near-duplicate detection rely on structural, visual, or hybrid heuristics that require handcrafted rules or similarity threshold tuning. In this paper, we present a new approach that learns semantic similarity through neural embeddings and SNNs, avoiding the need for manual calibration and thereby enabling more scalable integration in the automatic web test generation process.

III. Embedding Models

In this work, we evaluate several embedding methods, ranging from static representations (Doc2Vec) to transformer-based, context-aware models (BERT, ModernBERT, and MarkupLM), as they are suited to capture both semantic and structural dependencies within web pages, key aspects to distinguishing distinct, near-duplicate, and clone states. The main characteristics of each method are summarized below.

A. Doc2Vec

Doc2Vec [14] is a static embedding technique that generates a single dense vector for an entire document without explicit token-level context. Unlike BERT-based models, Doc2Vec does not require subword or byte-pair tokenization. Instead, it processes the entire document as a sequence of words and learns a fixed-length representation for each document. This approach is typically more efficient for large datasets, but the downside is that it may lack the nuanced context-dependent representation that BERT-like models provide.

B. BERT

BERT [15] relies on the WordPiece [22] subword tokenization algorithm to split input text into smaller units, limiting each sequence to a maximum length of 512 tokens. Any token exceeding this limit is truncated by the chunking process detailed in Section V-A2. After tokenization, the tokens are passed to the BERT model to generate embeddings, and this process is repeated for all chunks. The final input embedding is constructed by averaging the embeddings of all chunks into a single vector. We use the averaging technique for BERT embeddings because the dimensionality increases as the input size grows, which would cause our model to become larger.

C. ModernBERT

ModernBERT [16] also relies on the WordPiece [22] subword tokenization algorithm to split input text into smaller units, but with a larger context, as it can process sequences of up to 8,192 tokens. This extended context window is more than sufficient for the data considered in this work, and no truncation is therefore needed.

D. MarkupLM

MarkupLM [17] is a transformer model tailored for HTML documents. Building upon BERT, it encodes both textual and structural information by incorporating markup-aware features such as tag hierarchies, DOM positions, and XPath representations. Unlike standard BERT tokenizers that treat tags as plain text, MarkupLM preserves HTML structure during tokenization, allowing it to capture relationships between nodes and their content. Similar to BERT, inputs are limited to 512 tokens, but MarkupLM's HTML-aware positional encoding enables the model to represent hierarchical layouts that standard transformers can not [23]. In our setup, each node's text and XPath are jointly embedded to form structure-aware vector representations. These embeddings, concatenated across document chunks, yield richer, context-sensitive encodings of web pages for downstream similarity and modeling tasks.

IV. Siamese Neural Networks

Siamese Neural Networks (SNNs) are neural architectures designed to learn similarity relationships between pairs of inputs by projecting them into a shared embedding space. Originally developed for tasks such as signature and face verification, SNNs have shown strong performance in domains requiring precise discrimination between similar but non-identical instances. Their strength lies in learning a generalized similarity function rather than fixed class boundaries, making them particularly suitable for web application state comparison, where new, previously unseen states frequently emerge, and labeling all possible variations is infeasible.

We train two variants of SNN, each adopting a different loss function: one using Binary Cross-Entropy (BCE) [18] and the other using Triplet Loss [19], [20]. In the following, we detail these two training strategies and provide the mathematical formulations for each loss function.

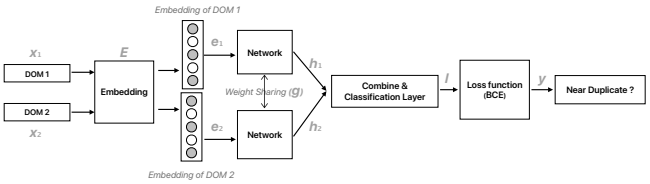

Fig. 2: BCE-based SNN Architecture.

1) BCE-Based SNN: The BCE-based SNN architecture is displayed in Figure 2. Initially, a pair of input samples, (x_1, x_2), is transformed into initial embeddings using a selected embedding technique E. These embeddings, namely $\mathbf{e}_1 = E(x_1)$ and $\mathbf{e}_2 = E(x_2)$, are then passed through a *shared* feature extraction network g, yielding two lower-dimensional representations, $\mathbf{h}_1 = g(\mathbf{e}_1)$ and $\mathbf{h}_2 = g(\mathbf{e}_2)$, where g denotes the learned task-specific embedding function, typically implemented as a feed-forward neural network. To explicitly model the relationship between the embeddings, these hidden representations are combined into a single representation $\mathbf{h}_{\text{combined}}$, as follows:

$$\mathbf{h}_{\text{combined}} = [\,\mathbf{h}_1; \mathbf{h}_2; |\mathbf{h}_1 - \mathbf{h}_2|; (\mathbf{h}_1 \odot \mathbf{h}_2); \text{sim}(\mathbf{h}_1, \mathbf{h}_2)\,],$$

where $[\cdot;\cdot]$ denotes concatenation, $\odot$ is the element-wise product, and $\text{sim}(\mathbf{h}_1, \mathbf{h}_2)$ is the cosine similarity metric. The combined representation $\mathbf{h}_{\text{combined}}$ is then fed into a small classification layer c, which computes a logit $\ell = c(\mathbf{h}_{\text{combined}}) = W\mathbf{h}_{\text{combined}} + b$, where W and b represent the learned weights and biases in the classification layer, respectively. Finally, the logit ℓ is passed through a sigmoid activation function σ, producing the estimated probability $\hat{y} = \sigma(\ell)$ that the input pair represents a near-duplicate pair. Let $y \in \{0, 1\}$ be the ground-truth label, where $y = 1$ indicates a near-duplicate pair, and $y = 0$ indicates a distinct pair. We optimize the network parameters by minimizing the BCE loss:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}\big[\, y_i \,\log\big(\sigma(\ell_i)\big) \;+\; (1 - y_i)\,\log\big(1 - \sigma(\ell_i)\big)\big],$$

where N is the total number of training pairs, ℓ_i is the predicted logit for the i-th pair, and $\sigma(\ell_i)$ is the output probability after applying the sigmoid function. By minimizing this loss, the network learns to assign higher probabilities to near-duplicate pairs and lower probabilities to distinct pairs.

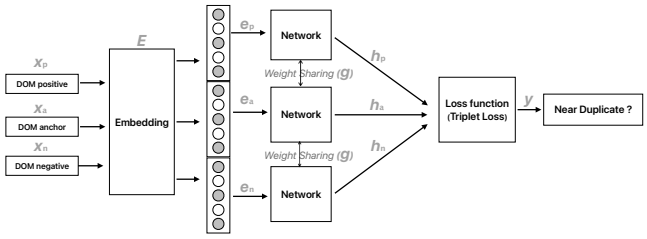

Fig. 3: Triplet-Based SNN Architecture

2) Triplet-Loss-Based SNN: In the Triplet-Based SNN approach as visible in Figure 3, we enforce *relative distance* constraints rather than directly predicting a binary label. Each training sample is organized into a triplet (x_a, x_p, x_n), where x_a is an *anchorDOM*, x_p is a *positiveDOM*, and x_n is a *negativeDOM*. A shared embedding function g projects these inputs into a latent space, i.e., $\mathbf{h}_a = g(E(x_a))$, $\mathbf{h}_p = g(E(x_p))$, and $\mathbf{h}_n = g(E(x_n))$. We then compute the squared Euclidean distances $D_{ap} = \|\mathbf{h_a} - \mathbf{h_p}\|_2^2$, $D_{an} = \|\mathbf{h_a} - \mathbf{h_n}\|_2^2$. The triplet loss encourages $\mathbf{a}$ to be closer to $\mathbf{p}$ than to $\mathbf{n}$ by at least a margin α. Formally, the triplet loss is given by:

$$\mathcal{L}_{\text{triplet}} = \frac{1}{M}\sum_{i=1}^{M}\max\big(0,\, \alpha + D_{ap}^{i} - D_{an}^{i}\big),$$

where M is the number of triplets and α is a hyperparameter that controls how strongly the positive distance must be smaller than the negative distance. This objective naturally clusters near-duplicate samples in the embedding space and pushes distinct samples farther apart, without requiring an explicit probability output. We set α to 1.0, the default parameter provided in the original paper.

During inference, we use the shared weights learned during training to compute embeddings directly for any two given samples (without requiring a triplet). We then measure their distance in the embedding space and compare this distance to a standard threshold 0.5, selected through pilot experiments. If the distance is below this threshold, the pair is classified as a near-duplicate; otherwise, it is considered distinct.

V. Approach

Figure 4 illustrates our approach, which consists of four phases, namely (1) Embedding Pipeline, (2) Dataset Creation and Training Pipeline, (3) Crawling, and (4) Test Creation.

In the Embedding Pipeline phase, we start from a labeled corpus of web pages where pairs are annotated as either distinct or clone/near-duplicate. For each distinct web page (GUI state), we extract a token-sequence representation of the DOM – a combined representation of *content+tags* – and then compute embeddings using a pre-trained model. After computing the embedding representations of the distinct web pages, we use them in the subsequent Dataset Creation and Training Pipeline phase to train SNN classifiers to distinguish distinct and clone/near-duplicate web pages. In the third phase, Crawling, the trained SNN-based classifiers are deployed as runtime state abstraction functions (SAFs). For each newly encountered web page, our approach computes its embedding, compares it against embeddings of existing states in the model, and uses the classifier to predict whether the page represents a distinct state or a near-duplicate of an existing one. In the fourth phase, Test Generation, each crawl path in the model inferred in the previous phase is turned into a web test case. We now detail each phase of our approach.

A. Embedding Pipeline

1) Data Preprocessing: Our approach requires computing embeddings for web pages. We adopt a combined representation based on *content+tags*, as it retains both textual and structural information for web page understanding. We apply an initial set of preprocessing steps to all embedding models considered in this work, namely: Doc2Vec [14], BERT [15], ModernBERT [16] and MarkupLM [17]. First, we remove any `<script>` and `<style>` tags, HTML comments, and HTML attributes that do not contribute to a page's primary content or structure.

Listing 1: HTML of the My Reservation page (State 7)

```
<html lang="en">
<head>
  <title>Reservations</title>
  <link rel="stylesheet" href="styles.css"></link>
  <script src="utils.js"></script>
</head>
<body>
  <div class="reservations-container">
    <h2 class="header">
      <i class="home-icon">&#8962;</i>
      My Reservations
    </h2>
    <div class="reservation-card">  <!-- Reservation 1 -->
      <img src="hotel-icon.png" alt="Hotel" class="hotel-icon">
      <h3 class="hotel-name">Hotel B</h3>
      <p class="stay-details">25 Apr - 30 Apr, Munich</p>
      <span class="status ongoing">On going</span>
      <i class="icon heart-icon">&#9825;</i>
      <i class="icon arrow-icon">&#9654;</i>
    </div>
    <div class="reservation-card">  <!-- Reservation 2 -->
      <img src="hotel-icon.png" alt="Hotel" class="hotel-icon">
```

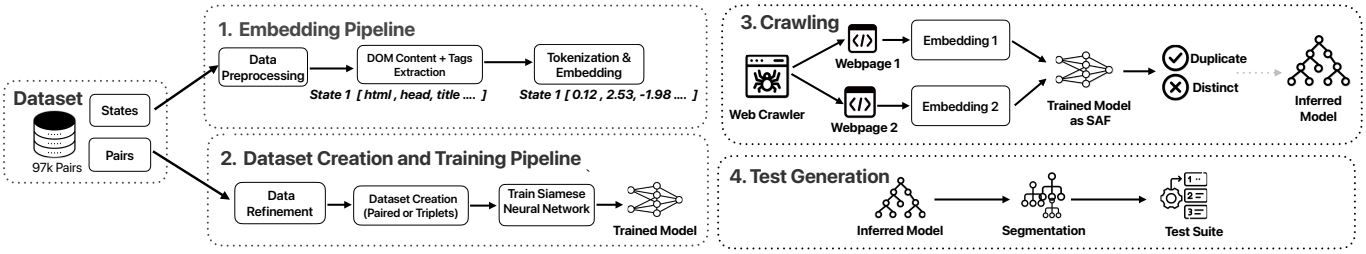

Fig. 4: Overview of our approach.

```
        <h3 class="hotel-name">Hotel C</h3>
        <p class="stay-details">1 Jan - 5 Jan, Munich</p>
        <span class="status completed">Completed</span>
        <i class="icon heart-icon">&#9825;</i>
        <i class="icon arrow-icon">&#9654;</i>
      </div>
    </div>
  </body>
  </html>
```

As shown in Listing 1, the HTML snippet contains additional *tags* (e.g., `<script>` and `<link>`), *comments*, and *attributes* that we remove in the first preprocessing step. We then retain only the *tags* and *text content* and remove special characters such as `<`, `!`, `/`, and `>`. This final token sequence preserves the essential structural and textual elements needed for our downstream embedding models while excluding *tags*, *attributes*, and content deemed non-essential. For Doc2Vec, BERT and ModernBERT, the input token representation is:

```
[html, head, title, Reservations, title, head,
body, div, h2, i, &#8962;, i, My, Reservations,
h2, div, img, h3, Hotel, B, h3, p, 25, Apr, -,
30, Apr, Munich, p, span, On, Going, i, &#9825;,
i, i, &#9654;, i, div, img, h3, Hotel, C, h3, p,
1, Jan, -, 5, Jan, Munich, p, span, Completed,
span, i, &#9825;, i, i, &#9654;, i, div, div,
body, html]
```

Since MarkupLM [17] is trained directly on HTML pages rather than plain text, we provide it with each web page's node content as a list alongside their corresponding XPath to obtain embeddings. For the example in Listing 1, the MarkupLM input token representation is shown below.

Node Sequence:

```
[Reservations, &#8962;, My, Reservations, Hotel,
B, 25, Apr, -, 30, Apr, Munich, On, going,
&#9825;, &#9654;, Hotel, C, 1, Jan, -, 5, Jan,
Munich, Completed, &#9825;, &#9654;]
```

XPath Sequence:

```
[/html/head/title, /html/body/div/h2/i, /html/body/div/h2
, /html/body/div/h2, /html/body/div/div[1]/h3, /html/body
/div/div[1]/h3, /html/body/div/div[1]/p, /html/body/div/
div[1]/p, /html/body/div/div[1]/p, /html/body/div/div[1]/
p, /html/body/div/div[1]/p, /html/body/div/div[1]/p, /
html/body/div/div[1]/span, /html/body/div/div[1]/span, /
html/body/div/div[1]/i, /html/body/div/div[1]/i, /html/
body/div/div[2]/h3, /html/body/div/div[2]/h3, /html/body/
div/div[2]/p, /html/body/div/div[2]/p, /html/body/div/div
[2]/p, /html/body/div/div[2]/p, /html/body/div/div[2]/p,
/html/body/div/div[2]/p, /html/body/div/div[2]/span, /
html/body/div/div[2]/i, /html/body/div/div[2]/i]
```

2) Chunking: Doc2Vec embeddings are static and can handle inputs of any length. ModernBERT can also handle 8,192 sequence lengths, which provides a sufficient context for the data we consider in this work. However, BERT and MarkupLM embeddings are constrained by a maximum input size of 512 tokens. For large documents, such as lengthy HTML files, any token beyond this limit is ignored during embedding generation, leading to potential loss of relevant information. Even after our preprocessing steps, we observe that approximately 34% of tokens for BERT are trimmed on average, whereas MarkupLM trims about 6% (Table I).

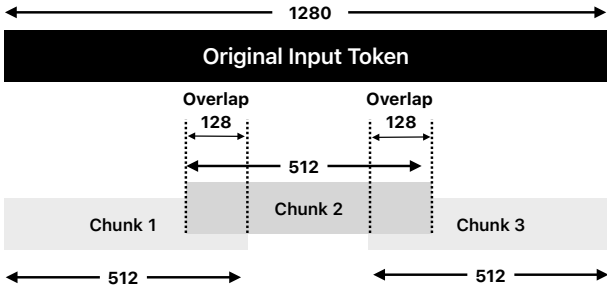

Fig. 5: Chunking with a 128-token overlap.

To mitigate this limitation, we introduce chunking. First, we split each document into chunks of up to 512 tokens, with a defined overlap to maintain contextual coherence across chunk boundaries. This overlap helps ensure that important information spanning the boundary between two chunks is not lost. If a document does not reach the maximum number of chunks, we pad the remaining chunks with zeros so that all inputs maintain a consistent dimensionality during training.

Too many chunks can lead to excessive zero padding, which wastes computational resources and may negatively impact model training. To address this, we introduce a maximum number of chunks, i.e., the *chunk threshold*, which is a hyperparameter that we tune empirically. For instance, a sequence of 1280 tokens can be split into three chunks, each with a 128-token overlap, as illustrated in Figure 5. After applying chunking, the average truncation for BERT is reduced from 34% to 18% with 2 chunks, and with four chunks, it can be lowered to just 3%. For MarkupLM, using two chunks decreases its truncation rate from 6% to a mere 1%, as indicated in Table I.

TABLE I: Average Truncated Percentage.

Embedding	No Chunking	2 Chunks	3 Chunks	4 Chunks
Bert Base	34%	18%	15%	3%
MarkupLM	6%	1%	0%	0%

Algorithm 1: Web App Crawling with SNNs

```
1  EXTRACTTOKENS(n) let tokens be an empty list;
2  tokens.append(getTokens(n));
3  foreach children node c of n, from left to right do
4      if c is not a script, style, or comment node then
5          tokens.append(EXTRACTTOKENS(c))
6  return tokens;
7  Function CRAWL(initial URL):
8      s_1 ← getState(initial URL);
9      model ← initializeModel(s_1);
10     while ¬ timeout do
11         next ← nextStateToExplore(model);
12         if next = nil then                    ▷ app exhaustively explored
13             break;
14         s ← getToState(next);
15         for e ∈ getCandidateEvents(s) do
16             fireEvent(e);
17             s_c ← current state after firing the event e;
18             if ¬ ISDUPLICATE(s, model) then
19                 add s_c to model;
20     return model;
21 Function ISDUPLICATE(s_c, model):
22     foreach state s′ in model do
23         if CLASSIFY(s_c, s′) = 'clone' then
24             return True;                      ▷ s is a duplicate of s′
25     return False;
26 Function CLASSIFY(p_1, p_2, EM):              ▷ EM: embedding model
27     r_1 ← EXTRACTTOKENS(p_1.getRootNode());
28     r_2 ← EXTRACTTOKENS(p_2.getRootNode());
29     e_1 ← EM.computeEmbedding(r_1) ;
30     e_2 ← EM.computeEmbedding(r_2);
31     s ← cosineSimilarity(e_1, e_2);
32     return classifier.classify(s);
```

3) Tokenization and Embedding: In this work, we perform classification tasks without fine-tuning the transformer-based models to assess whether these rich embeddings are sufficient for our downstream task of web page understanding.

For Doc2Vec, we used a pre-trained embedding model to generate embeddings [24], as this model is already fine-tuned for generating embeddings for HTML inputs.

For BERT, ModernBERT, and MarkupLM, we used the given tokenizer (available on `HuggingFace`) to convert the input into a sequence of tokens.

BERT and MarkupLM limit each sequence to a maximum length of 512 tokens. We addressed longer sequences using chunking (Section V-A2). After tokenization, the tokens are passed to the corresponding models to generate embeddings, and this process is repeated for all chunks. The final input embedding is constructed by averaging the embeddings of all chunks into a single vector.

On the other hand, ModernBERT can process sequences of up to 8,192 tokens. This extended context window is more than sufficient for the data considered in this work; hence, we do not employ chunking.

B. Training State Abstraction Function

In the second phase, we train the SAF using a labeled corpus of web page pairs, where each pair is annotated to indicate whether the pages are clones or near-duplicates. For each pair, we generate embeddings using one of the selected embedding models and compute their cosine similarity [25], a standard metric for measuring vector similarity. These embeddings and similarity scores are then used to train an SNN-based classifier to distinguish between near-duplicate and distinct web pages. For simplicity, clones are grouped under the near-duplicate category, as their detection does not require additional sophistication beyond our similarity analysis.

C. Crawling and Model Inference

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

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—21).

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 9). Subsequently, the main loop (lines 11—20) 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 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.

2) Usage of the State Abstraction Function: The CLASSIFY procedure (lines 27—33) illustrates the SAF. Given two web pages p_1, p_2, we first extract the token-sequence representation from each page based on the selected embedding model EM, obtaining the 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 embedding 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 33), which is '*clone*' in case of near-duplicate detection or '*distinct*' otherwise.

Example. Consider the following embeddings produced for our running example, i.e., embedding on content+tags and EM = ['*BERT*']:

p_1 = Accomodations Page $e_1 = [-0.45, 0.56, ..., 0.30]$
p_2 = Detail Page Hotel A $e_2 = [-0.55, 0.17, ..., 0.90]$
p_3 = Detail Page Hotel B $e_3 = [-0.56, 0.19, ..., 0.95]$

During crawling, let us assume that a decision tree classifier flags a pair of pages as '*near-duplicate*' 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 of p_2.

D. Test Creation

Our approach automatically generates a test suite during crawling through *segmentation* [5]. 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 our SAF, 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.

VI. Empirical Study

A. Research Questions

To assess the practical benefits of WebEmbed 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*'s web app models?*

RQ_4 (time efficiency). *How efficient is* WebEmbed *in terms of training and inference time, and its applicability in real-world web crawling?*

RQ_1 aims to assess what configuration (Embedding model and network variant) of our approach, in terms of 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 our approach when used for web testing, specifically assessing the test suites generated from the crawl models in terms of code coverage of the web apps under test. RQ_4 addresses the computational efficiency of our approach. Two sub-questions related to time efficiency are evaluated: the model training duration ($RQ_{4.1}$) and inference time ($RQ_{4.2}$) from 1,000 randomly selected state sample pairs.

TABLE II: 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

B. Datasets

We used an existing dataset ($\mathcal{SS}$) available from the study by Yandrapally et al. [12]. It contains 97,500 state-pairs of nine subject apps (Table II), which were also manually labeled by the authors of the study as clone, near-duplicate or distinct. These nine web apps (Table II) have been used as subjects in previous research on web testing [26], [27], [8], [7]. 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*) [28], Claroline (App_2, *v. 1.11.10*) [29], PPMA (App_3, *v. 0.6.0*) [30], MRBS (App_4, *v. 1.4.9*) [31] and MantisBT (App_5, *v. 1.1.8*) [32]. Four are JavaScript single-page applications–Dimeshift (App_6, *commit 261166d*) [33], Pagekit (App_7, *v. 1.0.16*) [34], Phoenix (App_8, *v. 1.1.0*) [35] and PetClinic (App_9, *commit 6010d5*) [36]–developed using popular JavaScript frameworks such as *Backbone.js*, *Vue.js*, *Phoenix/React* and *AngularJS*.

C. Baselines

Based on the study by Yandrapally et al. [12], [37], we selected four algorithms as baselines for our approach, one structural, one visual, and one hybrid method. The structural algorithm is RTED (Robust Tree Edit Distance) [38], a DOM tree edit distance algorithm. The visual algorithm is PDiff [39], which compares two web page screenshots based on a human-like concept of similarity that uses spatial, luminance, and color sensitivity. The hybrid method is FragGen [37], a fragment-based approach for near-duplicate detection. FragGen uses a more granular representation of a web page by incorporating both structural and visual aspects. Specifically, it employs a deterministic recursive method built upon the RTED algorithm for structural comparison, and the Visual Histogram algorithm [40] for visual comparison.

We chose them as baselines for our approach because they are the best structural, visual, and hybrid algorithms for near-duplicate detection [12], [37].

TABLE III: WEBEMBED's configurations.

Name	Embedding Model	SNN Variant/Loss
SNN (T_1)	Doc2Vec	BCE
SNN (T_2)	Doc2Vec	Triplet
SNN (T_3)	BERT	BCE
SNN (T_4)	BERT	Triplet
SNN (T_5)	ModernBERT	BCE
SNN (T_6)	ModernBERT	Triplet
SNN (T_7)	MarkupLM	BCE
SNN (T_8)	MarkupLM	Triplet

TABLE IV: Near-Duplicate Detection (RQ_1). Bold=best F_1.

Technique	Precision	Recall	F_1
SNN (T_1)	**0.95**	**0.98**	**0.96**
SNN (T_2)	0.68	0.97	0.77
SNN (T_3)	0.80	0.98	0.85
SNN (T_4)	0.58	0.93	0.70
SNN (T_5)	0.80	0.96	0.86
SNN (T_6)	0.60	0.90	0.70
SNN (T_7)	0.80	0.90	0.79
SNN (T_8)	0.58	0.93	0.69
FragGen	0.83	0.70	0.76
RTED	0.65	0.74	0.67
PDiff	0.72	0.62	0.65

D. Configurations

We evaluated eight configurations of WEBEMBED, combining four initial embedding models (Doc2Vec, BERT, ModernBERT, and MarkupLM) with two SNN network variants. All the configurations are listed in Table III and are named $SNN_{1\text{-}8}$. We trained an app-specific classifier for each of the nine web apps. In total, we trained and fine-tuned 72 SNN models for all nine applications. For each app, we used 80% of the state pairs for training the classifier, 10% for validation, and the remaining 10% for testing.

E. Procedure and Metrics

1) RQ_1 (near-duplicate detection): We evaluated WEBEMBED's configurations towards maximizing near-duplicate detection by assessing them on the labeled $\mathcal{SS}$ dataset, and comparing them against baseline methods. We used Precision, Recall, and the F_1-score, standard metrics for binary classification, where the positive class corresponds to *near-duplicate* states. Precision measures the proportion of correctly identified near-duplicate pairs among all predicted near-duplicates, while Recall measures the proportion of actual near-duplicate pairs correctly identified. The F_1-score is their harmonic mean, providing a balanced view of both precision and recall.

2) RQ_2 (model quality): The crawl models contain redundant concrete states that Yandrapally et al. [12] 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 computed the precision, recall, and F_1 scores, by simulating the crawling process using manual ground-truth classifications. In this context, Precision describes the ratio of unique states to the total number of states in the model, while Recall represents the ratio of covered functional bins to the total number of functional bins.

3) RQ_3 (code coverage): To assess the effectiveness of WEBEMBED when used for web testing, we crawled each web application in $\mathcal{SS}$ multiple times, each time varying the SAF. For all tools and apps, we set the same crawling time of 30 minutes. We used DANTE [5] to generate Selenium web test cases from the crawl sequences, execute the tests, and measured the web app code coverage. For JavaScript-based apps (Dimeshift, Pagekit, Phoenix, PetClinic), we measured *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 measured the *server-side* code coverage using the `xdebug` (v. 2.2.4) PHP extension and `php-code-coverage` (v. 2.2.3). During our evaluation, we first determined the maximum achievable code coverage through manual exploration.

4) RQ_4 (time efficiency): Time efficiency is critical for any SAF integrated within a real-time web-crawling pipeline. Indeed, during crawling, thousands of state comparisons are performed, and even small delays can accumulate and significantly slow down exploration. To quantify this aspect, we measured the inference time of the investigated techniques, which directly impacts the efficiency and effectiveness of model inference and test generation.

We introduced two metrics related to time efficiency: model training time ($RQ_{4.1}$) and inference time ($RQ_{4.2}$). Specifically, model training time refers to the duration taken to train the SNN model. The inference time efficiency is evaluated using a subset of 1,000 randomly selected state pairs. Since FragGen, RTED, and PDiff methods are closely integrated within Crawljax, we extracted the distance calculation components from these methods and measured the execution time needed to generate the results.

F. Results

1) RQ_1 (near-duplicate detection): Table IV presents the results for near-duplicate detection comparing the proposed technique against baseline methods. The configuration SNN (T_1) achieves the best results, with the highest average F1-score of 0.96 for near-duplicate detection. When compared to the baselines, it provides a considerable improvement in near-duplicate detection, yielding approximately 23%, 43%, and 47% increases in F_1-score compared to FragGen (0.76), RTED (0.67), and PDiff (0.65), respectively.

We further analyzed the number of correctly classified near-duplicate state pairs. To ensure a fair comparison with the baselines, we excluded App_4, for which FragGen's results were not available. Looking at Figure 6, SNN (T_1) outperforms the baselines by a large margin on all categories: Clones, ND_2 near-duplicates, and ND_3 near-duplicates. Overall, it correctly classifies 10,594 pairs, outperforming FragGen (7,538), PDiff (7,451), and RTED (7,882), representing improvements of 41%, 42%, and 34%, respectively.

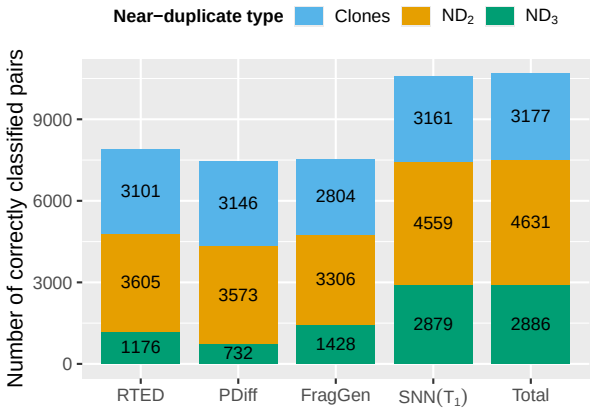

Fig. 6: Correctly classified near-duplicate state-pairs.

TABLE V: Model Quality (RQ_2). Bold=best F_1.

	SNN (T_1)			FragGen			RTED			PDiff		
App	**Pr**	**Re**	**F_1**	**Pr**	**Re**	**F_1**	**Pr**	**Re**	**F_1**	**Pr**	**Re**	**F_1**
App$_1$	1.00	0.50	0.67	0.38	1.00	0.55	0.70	0.58	0.64	0.18	1.00	0.31
App$_2$	0.62	1.00	0.77	0.92	0.92	0.92	1.00	0.83	0.91	0.94	0.81	0.86
App$_3$	0.96	1.00	0.98	0.45	0.43	0.44	0.23	1.00	0.38	1.00	0.87	0.93
App$_4$	0.65	0.93	0.76	N/A	N/A	N/A	0.53	0.64	0.58	0.20	.64	0.31
App$_5$	0.71	1.00	0.83	0.44	0.74	0.55	0.59	0.72	0.65	0.81	0.57	0.67
App$_6$	1.00	0.67	0.80	0.10	0.67	0.18	0.80	0.76	0.78	0.35	0.81	0.49
App$_7$	0.59	1.00	0.74	0.38	0.80	0.52	0.47	0.75	0.58	0.35	0.81	0.41
App$_8$	1.00	0.70	0.82	0.30	0.80	0.43	0.55	0.60	0.57	0.09	0.90	0.17
App$_9$	1.00	0.79	0.88	0.29	0.86	0.44	0.57	0.73	0.60	0.18	0.79	0.29
Avg.	**0.84**	**0.84**	**0.81**	0.40	0.78	0.50	0.57	0.73	0.60	0.47	0.79	0.51

RQ_1: WEBEMBED *achieves an overall improvement of 23%—68% of F_1 scores compared to all baseline methods. These enhancements correspond to an increased number of accurately classified near-duplicate state pairs.*

2) RQ_2 (model quality): Table V shows the Precision (Pr), Recall (Re), and F_1 scores (F_1) of WEBEMBED and the baselines for all apps, along with the average per-app results. Although all WEBEMBED's configurations were evaluated, only the top-performing technique from RQ_1 (i.e., SNN(T_1)) alongside the baseline is reported due to space limitations.

Overall, WEBEMBED produces more accurate models (i.e., models more similar to the ground truth) than the competing techniques across all use cases. Specifically, the SNN(T_1) configuration of WEBEMBED achieved the highest F_1-score of 0.81, a substantial improvement of 62% compared to FragGen, 35% compared to RTED, and 59% compared to PDiff.

RQ_2: WEBEMBED *shows an overall improvement of 35% to 72% over baseline methods and is better at approximating the ground-truth model than all existing techniques.*

3) RQ_3 (code coverage): Table VI presents the code coverage results for WEBEMBED as well as for the baseline methods. Taking into account the average scores for all nine applications, WEBEMBED achieves the highest score compared to all baselines. Recall that the coverage results reported in Table VI are normalized relative to the maximum achievable through manual exploration, which averaged at 38.4% across all applications. WEBEMBED (SNN (T_1)) demonstrates the best code coverage of 71.99% on average. This result indicates an improvement of approximately 6%, 11%, and 35% over FragGen, RTED, and PDiff, respectively. A Wilcoxon signed-rank test and Vargha–Delaney effect size analysis were conducted over the nine applications. SNN (T_1) achieved the highest average coverage (71.99%), outperforming FragGen (67.96%), RTED (64.74%), and PDiff (53.25%). The improvements over RTED ($p = 0.019, A_{12} = 0.83$) and PDiff ($p = 0.019, A_{12} = 0.89$) are statistically significant with large effect sizes, while the difference over FragGen ($p = 0.078, A_{12} = 0.72$) shows a medium-to-large practical effect, though not significant at the 0.05 level.

TABLE VI: Code Coverage (RQ_3). Bold=best average score.

App	SNN (T_1)	FragGen	RTED	PDiff
App$_1$	87.89	91.48	90.11	44.98
App$_2$	29.73	41.34	41.58	22.50
App$_3$	85.10	67.17	47.87	48.75
App$_4$	75.25	60.69	66.49	35.06
App$_5$	35.76	38.18	32.06	23.94
App$_6$	52.50	32.65	28.62	28.62
App$_7$	95.18	96.46	92.36	92.31
App$_8$	94.67	95.10	95.26	95.26
App$_9$	87.81	88.60	88.28	87.81
Avg.	**71.99**	67.96	64.74	53.25

TABLE VII: Time Efficiency (RQ_4) Inference time (Seconds). The best inference time is boldfaced.

Technique	Avg. Inference Time	Δ vs. WEBEMBED
WEBEMBED	**0.09**	–
FragGen	1.44	×16
RTED	0.68	×7.6
PDiff	2.03	×22.6

RQ_3: *The tests generated from* WEBEMBED *crawl models achieve the highest code coverage scores, showcasing 6%—21% improvement thanks to the more accurate and complete web app models.*

4) RQ_4 (time efficiency): Table VII shows the average inference times for all techniques. WEBEMBED achieves the fastest inference time, which is an order of magnitude quicker than FragGen and PDiff, and significantly faster than RTED. This supports the applicability of WEBEMBED in real-world crawling scenarios, where thousands of state comparisons must be performed efficiently to keep exploration and test generation within practical time budgets.

RQ$_4$: WEBEMBED *achieves the fastest inference time at* 0.09 *seconds, delivering remarkable speedups of* 16× *over FragGen,* 7.6× *over RTED, and* 22.6× *over PDiff.*

G. Threats to Validity

1) Internal validity: We compared all variants of WEBEMBED and baselines under identical experimental settings and on the same evaluation set (Section VI-B). In our experiments, a crawling time of 30 minutes allowed all crawls to explore all logical pages of the AUTs within the timeout. 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. The inference times reported in Table VII compare the runtime of Java implementations of the baseline methods (FragGen, RTED, and PDiff within Crawljax) with Python implementations (WEBEMBED). Although all experiments were conducted on identical hardware, the startup overhead of the Java Virtual Machine and Python's Global Interpreter Lock may introduce biases in the absolute timings.

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. However, our evaluation considered nine representative web applications, that cover a range of architectures (single- and multi-page), GUI complexity, and dynamic behavior typical of modern SPAs. Results may however not transfer to applications whose technologies, languages, or interaction patterns were absent from our corpus.

VII. RELATED WORK

A. End-to-End Web Test Automation

Most E2E testing techniques rely on a web app model of the application under test, either manually generated by developers or automatically inferred, e.g., through crawling [9], [41], [42], [43], [27], [26], [44]. Biagiola et al. [6], [8] use Page Objects to guide the generation of tests. Marchetto et al. [42] 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. [9] 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.

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.

B. Empirical Studies on Near-Duplicates

Fetterly et al. [45] study the nature of near-duplicates during software evolution, reporting their low variability over time. Yandrapally et al. [12] compare different near-duplication detection algorithms as SAFs from the fields of computer vision and information retrieval. The paper reports on the challenge of finding an accurate SAF using DOM-based or visual techniques, which motivates the work in this paper to use advanced neural embeddings and classifiers from the deep learning domain.

C. Automated Near-Duplicate Detection

Regarding detection of near-duplicates *within* the same app, Crescenzi et al. [46] propose a structural abstraction for web pages and a clustering algorithm based on such abstraction. Di Lucca et al. [47], [48] evaluate the Levenshtein distance and the tag frequency for detecting near-duplicate web pages. Stocco et al. [26] use clustering on structural features as a post-processing technique to discard near-duplicates in crawl models. Corazza et al. [49] propose the usage of tree kernels.

Concerning detection of near-duplicates *across* apps, researchers considered clustering techniques on raw structural features [50], [51], [45], [52], [47], [48], [46], [26]. Other works, such as the one by Henzinger [50], use shingles, i.e., n-grams composed of contiguous subsequences of tokens, to ascertain the similarity between web pages. Manku et al. [51] use `simhash` to detect near-duplicates in the context of information retrieval, plagiarism, and spam detection. In this paper, we consider neural embeddings to train an SNN-based classifier for near-duplicate detection, and we illustrate that its usage for functional testing of web apps outperforms state-of-the-art techniques.

D. Embeddings in Software Engineering

Alon et al. [53] 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. [54] 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. [24] use representation learning applied across web apps for phishing detection. Lugeon et al. [55] propose Homepage2Vec, an embedding method for website classification. Namavar et al. [56] performed a large-scale experiment comparing different code representations to aid bug repair tasks. In this work, we use neural embeddings from the deep learning domain, a novel contribution in the context of automated crawling and web testing.

Ma et al. [57] 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. [58] propose dev2vec, an approach to embed developers' domain expertise within vectors for the automated assessment of developers' specialization. Jabbar et al. [59] encode the test execution traces for test prioritization.

Differently, we use several neural embeddings to train an SNN classifier that is used as SAF within a crawl-based test generator for functional testing.

VIII. 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 SNN-based classifiers for near-duplicate detection. We demonstrate their effectiveness in inferring accurate models for functional testing of 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 will explore richer embeddings to further enhance WEBEMBED's accuracy, including visual embeddings from web screenshots (e.g., via autoencoders [60], [61]), hybrid representations, and fine-tuned transformer models. In this study, we deliberately avoided fine-tuning to assess the intrinsic quality of web-aware embeddings and ensure fair, reproducible comparisons. We also plan to evaluate WEBEMBED's bug-finding capabilities, for instance, through mutation testing.

REFERENCES

[1] F. Ricca, M. Leotta, and A. Stocco, "Three open problems in the context of e2e web testing and a vision: Neonate," in *Advances in Computers*, vol. 113, pp. 89–133, Elsevier, 01 2019.

[2] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Approaches and tools for automated end-to-end web testing," *Advances in Computers*, vol. 101, pp. 193–237, 01 2016.

[3] P. Tonella, F. Ricca, and A. Marchetto, "Recent advances in web testing," *Advances in Computers*, vol. 93, pp. 1–51, 2014.

[4] L. Christophe, R. Stevens, C. D. Roover, and W. D. Meuter, "Prevalence and maintenance of automated functional tests for web applications," in *Proceedings of 30th International Conference on Software Maintenance and Evolution*, ICSME 2014, IEEE, 2014.

[5] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Dependency-aware web test generation," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, ICST '20, pp. 175–185, IEEE, IEEE, 2020.

[6] M. Biagiola, F. Ricca, and P. Tonella, "Search based path and input data generation for web application testing," in *International Symposium on Search Based Software Engineering*, pp. 18–32, Springer, 2017.

[7] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella, "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, p. 12 pages, ACM, 2019.

[8] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "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, 07 2019.

[9] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 35–53, 2012.

[10] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web*, vol. 6, no. 1, pp. 3:1–3:30, 2012.

[11] M. Leithner and D. E. Simos, "Xiev: dynamic analysis for crawling and modeling of web applications," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pp. 2201–2210, 2020.

[12] R. Yandrapally, A. Stocco, and A. Mesbah, "Near-duplicate detection in web app model inference," in *Proceedings of 42nd International Conference on Software Engineering*, ICSE '20, p. 12 pages, ACM, 2020.

[13] R. K. Yandrapally and A. Mesbah, "Fragment-based test generation for web apps," *IEEE Transactions on Software Engineering*, vol. 49, pp. 1–1, Mar. 2022.

[14] J. H. Lau and T. Baldwin, "An empirical evaluation of doc2vec with practical insights into document embedding generation," *CoRR*, vol. abs/1607.05368, 2016.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[16] B. Warner, A. Chaffin, B. Clavié, O. Weller, O. Hallström, S. Taghadouini, A. Gallagher, R. Biswas, F. Ladhak, T. Aarsen, N. Cooper, G. Adams, J. Howard, and I. Poli, "Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference," 2024.

[17] J. Li, Y. Xu, L. Cui, and F. Wei, "Markuplm: Pre-training of text and markup language for visually-rich document understanding," 2022.

[18] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd ed., 2025. Online manuscript released January 12, 2025.

[19] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, p. 815–823, IEEE, June 2015.

[20] E. Hoffer and N. Ailon, "Deep metric learning using triplet network," 2018.

[21] "Replication package." https://github.com/ast-fortiss-tum/near-duplicate-detection-siamese-networks, 2026.

[22] X. Song, A. Salcianu, Y. Song, D. Dopson, and D. Zhou, "Fast wordpiece tokenization," 2021.

[23] V. Shiv and C. Quirk, "Novel positional encodings to enable tree-based transformers," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[24] J. Feng, L. Zou, O. Ye, and J. Han, "Web2vec: Phishing webpage detection method based on multidimensional features driven by deep learning," *IEEE Access*, vol. 8, pp. 221214–221224, 2020.

[25] A. Singhal, "Modern information retrieval: A brief overview.," *IEEE Data Eng. Bull.*, vol. 24, no. 4, pp. 35–43, 2001.

[26] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, "Clustering-aided page object generation for web testing," in *Proceedings of 16th International Conference on Web Engineering*, ICWE 2016, pp. 132–151, Springer, 2016.

[27] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, "APOGEN: Automatic Page Object Generator for Web Testing," *Software Quality Journal*, vol. 25, pp. 1007–1039, Sept. 2017.

[28] w.-b. a. . p. b. Simple, "Simple, web-based address & phone book." http://sourceforge.net/projects/php-addressbook, 2015. Accessed: 2018-10-01.

[29] Claroline, "Claroline. Open Source Learning Management System.." https://sourceforge.net/projects/claroline/, 2015.

[30] P. P. Manager, "PHP Password Manager." https://github.com/pklink/ppma, 2018.

[31] M. R. B. System, "Meeting Room Booking System." https://mrbs.sourceforge.io/, 2018.

[32] M. B. Tracker, "Mantis Bug Tracker." https://github.com/mantisbt/mantisbt, 2018.

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

[34] Pagekit, "Pagekit: modular and lightweight CMS. ." https://github.com/pagekit/pagekit, 2018.

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

[36] S. PetClinic, "Angular version of the Spring PetClinic web application." https://github.com/spring-petclinic/spring-petclinic-angular, 2018.

[37] Anonymous, "Fraggen replication package (2.0)." https://doi.org/10.5281/zenodo.5981993, 2022. Data set.

[38] M. Pawlik and N. Augsten, "Efficient computation of the tree edit distance," *ACM Trans. Database Syst.*, vol. 40, pp. 3:1–3:40, Mar. 2015.

[39] H. Yee, S. Pattanaik, and D. P. Greenberg, "Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments," *ACM Trans. Graph.*, vol. 20, pp. 39–65, Jan. 2001.

[40] M. J. Swain and D. H. Ballard, "Indexing via color histograms," in *Active Perception and Robot Vision* (A. K. Sood and H. Wechsler, eds.), (Berlin, Heidelberg), pp. 261–273, Springer Berlin Heidelberg, 1992.

[41] A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing web applications by modeling with fsms," *Software & Systems Modeling*, vol. 4, pp. 326–345, July 2005.

[42] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 121–130, IEEE, 2008.

[43] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, "Why creating web page objects manually if it can be done automatically?," in *Proceedings of 10th IEEE/ACM International Workshop on Automation of Software Test*, AST 2015, pp. 70–74, IEEE/ACM, 2015.

[44] B. Karagöz, F. Ricca, M. Biagiola, and A. Stocco, "Towards automated page object generation for web testing using large language models," in *Proceedings of the 19th IEEE International Conference on Software Testing, Verification and Validation*, ICST '26, p. 12 pages, IEEE, 2026.

[45] M. N. Dennis Fetterly, Mark Manasse, "On the evolution of clusters of near-duplicate web pages," in *Journal of Web Engineering*, vol. 2, pp. 228–246, Institute of Electrical and Electronics Engineers, Inc., October 2004.

[46] V. Crescenzi, P. Merialdo, and P. Missier, "Clustering web pages based on their structure," *Data Knowledge Engineering*, vol. 54, pp. 279–299, Sept. 2005.

[47] G. A. Di Lucca, M. Di Penta, A. R. Fasolino, and P. Granato, "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*, pp. 107–113, 2001.

[48] G. A. Di Lucca, M. D. Penta, and A. R. Fasolino, "An approach to identify duplicated web pages," *2013 IEEE 37th Annual Computer Software and Applications Conference*, vol. 00, no. undefined, p. 481, 2002.

[49] A. Corazza, S. Di Martino, A. Peron, and L. L. L. Starace, "Web application testing: Using tree kernels to detect near-duplicate states in automated model inference," in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, vol. abs/2108.13322 of *ESEM '21*, (USA), p. 1–6, ACM, Oct. 2021.

[50] M. Henzinger, "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, (New York, NY, USA), pp. 284–291, ACM, 2006.

[51] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," in *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pp. 141–150, ACM, 2007.

[52] L. Ramaswamy, A. Iyengar, L. Liu, and F. Douglis, "Automatic detection of fragments in dynamically generated web pages," in *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pp. 443–454, ACM, 2004.

[53] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[54] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, (New York, NY, USA), p. 518–529, ACM, 2020.

[55] S. Lugeon, T. Piccardi, and R. West, "Homepage2vec: Language-agnostic website embedding and classification," in *Proceedings of the International AAAI Conference on Web and Social Media*, vol. 16, pp. 1285–1291, 2022.

[56] M. Namavar, N. Nashid, and A. Mesbah, "A controlled experiment of different code representations for learning-based bug repair," *Empirical Software Engineering*, 2022.

[57] W. Ma, M. Zhao, E. Soremekun, Q. Hu, J. M. Zhang, M. Papadakis, M. Cordy, X. Xie, and Y. L. Traon, "Graphcode2vec: generic code embedding via lexical and program dependence analyses," in *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, (New York, NY, USA), p. 524–536, Association for Computing Machinery, 2022.

[58] A. M. Dakhel, M. C. Desmarais, and F. Khomh, "Dev2vec: Representing domain expertise of developers in an embedding space," *Information and Software Technology*, vol. 159, p. 107218, 2023.

[59] E. Jabbar, H. Hemmati, and R. Feldt, "Investigating execution trace embedding for test case prioritization," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pp. 279–290, 2023.

[60] A. Stocco, M. Weiss, M. Calzana, and P. Tonella, "Misbehaviour prediction for autonomous driving systems," in *Proceedings of 42nd International Conference on Software Engineering*, ICSE '20, p. 12 pages, ACM, 2020.

[61] A. Stocco and P. Tonella, "Towards anomaly detectors that learn continuously," in *Proceedings of 31st International Symposium on Software Reliability Engineering Workshops*, ISSREW 2020, IEEE, 2020.