Learning-based Widget Matching for Migrating GUI Test Cases

Yakun Zhang
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
zhangyakun@stu.pku.edu.cn

Wenjie Zhang
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
zhang_wen_jie@pku.edu.cn

Dezhi Ran
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
dezhiran@pku.edu.cn

Qihao Zhu
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
zhuqh@pku.edu.cn

Chengfeng Dou
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
2101111463@stu.pku.edu.cn

Dan Hao
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
haodan@pku.edu.cn

Tao Xie
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
taoxie@pku.edu.cn

Lu Zhang*
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
zhanglucs@pku.edu.cn

ABSTRACT

GUI test case migration is to migrate GUI test cases from a source app to a target app. The key of test case migration is widget matching. Recently, researchers have proposed various approaches by formulating widget matching as a matching task. However, since these matching approaches depend on static word embeddings without using contextual information to represent widgets and manually formulated matching functions, there are main limitations of these matching approaches when handling complex matching relations in apps. To address the limitations, we propose the first learning-based widget matching approach named *TEMdroid* (*TE*st *M*igration) for test case migration. Unlike the existing approaches, TEMdroid uses BERT to capture contextual information and learns a matching model to match widgets. Additionally, to balance the significant imbalance between positive and negative samples in apps, we design a two-stage training strategy where we first train a hard-negative sample miner to mine hard-negative samples, and further train a matching model using positive samples and mined hard-negative samples. Our evaluation on 34 apps shows that TEMdroid is effective in event matching (i.e., widget matching and target event synthesis) and test case migration. For event matching, TEMdroid's Top1 accuracy is 76%, improving over 17% compared to baselines. For test case migration, TEMdroid's F1 score is 89%, also 7% improvement compared to the baseline approach.

*Corresponding author

ICSE 2024, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623322

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Test migration, GUI testing, Deep learning

ACM Reference Format:
Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. 2024. Learning-based Widget Matching for Migrating GUI Test Cases. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623322

1 INTRODUCTION

GUI testing is common for testing the functionality of mobile apps [17, 51, 78]. A GUI test case is composed of some ordered test events and assertions [19, 46]. The events aim to explore the functionalities of GUI widgets. The assertions aim to check whether the outcomes of the events satisfy the expectations of developers. Manually constructing GUI test cases is tedious and time-consuming [18, 27, 49, 65]. To reduce the cost of manually writing GUI test cases, various approaches [19, 35, 46, 50, 58, 59] have been proposed to migrate GUI test cases that share similar functionalities from a source app to a target app. We refer to the widgets in the source app as the source widgets and the widgets in the target app as the target widgets. The key to fulfill the migration task is widget matching [43, 58] (i.e., mapping source widgets to their corresponding target widgets). If an incorrect target widget is mapped for a source widget, the migrated test case may fail to execute or can explore only partial functionality. Thus, the results of widget matching directly affect the migrated test cases.

There are two main categories, namely classification approaches and matching approaches, of existing approaches for widget matching. **Classification approaches**. Initially, researchers [35] formulate widget matching as a classification task, i.e., identifying widget labels and mapping source widgets to target widgets that

have the same widget labels. The main limitation of this category is that lower classification accuracy of widget labels can cause lower matching accuracy of widgets. **Matching approaches**. To address the limitation of the classification-based widget matching, subsequent research [19, 46, 50, 59] formulates widget matching as a matching task. A typical matching approach [46] uses a manually defined matching function to calculate the matching relation between a source widget and a target widget, where the representations of widgets use word embeddings (e.g., those from word2vec [62]) in natural language processing.

However, the existing matching approaches for widget matching still suffer from two main limitations. First, these approaches rely on static word embeddings (i.e., each word has one unique representation). However, in test case migration, widget semantics is typically represented by word sequences, and each word within a sequence can be contextualized by the other words. Hence, the existing matching approaches may fail to precisely capture the semantics of individual words, as they cannot effectively leverage contextual information (i.e., the representation of each word being contextualized by its surrounding words [13]). Second, the existing matching approaches require manually defined matching functions, which cannot handle complex matching relations. In a manually defined matching function, the knowledge for building the function comes from human analysis, insight, and experience. Consequently, matching functions developed in this way can hardly adapt to diverse matching relations in real-world apps.

To address the limitations of the existing matching approaches, we propose *TEMdroid* (*TE*st *M*igration), a novel learning-based widget matching approach for test case migration. TEMdroid is trained with migration data for GUI test cases and uses BERT [20] to incorporate contextual information instead of static word embeddings. By learning a matching model rather than relying on manually defined matching functions, TEMdroid can handle complex matching relations and adapt to diverse scenarios in real-world apps.

Specifically, we use migration data for GUI test cases to train the matching model, and design three modules of this model: the context-extraction, semantic-alignment, and widget-assessment modules. First, in the *context-extraction module*, TEMdroid uses BERT (a popular pre-trained language model with the self-attention mechanism [79]) to incorporate contextual information. Furthermore, to better understand widget semantics, TEMdroid customizes sentence embeddings, rather than using word embeddings [19, 46, 50, 58, 59]. The term "sentence embedding" represents semantics of word sequences. When TEMdroid is trained, a word sequence includes the textual information (e.g., the text on a widget) of a widget, and the sentence embedding for the word sequence represents not only the semantics of each word but also the contribution of each word to the word sequence. Second, the *semantic-alignment module* employs a Siamese network [25] to align the representations of two widgets into the same semantic space for fair comparison [28, 33, 54]. Third, the *widget-assessment module* measures the similarity between two widgets using the cosine distance.

To address a challenge (i.e., *imbalance between a few positive samples and many negative samples*[1] *for widget matching*) faced when training the matching model, we design a two-stage training strategy that incorporates the idea of mining hard samples (i.e., samples that are easily misjudged)[2]. **Stage 1: training and applying a hard-negative sample miner**. We first train a hard-negative sample miner by using all the positive samples from the migration data and an equal number of randomly selected negative samples from the migration data. Then we apply this miner to mine hard-negative samples (i.e., negative samples that are easily misjudged as positive samples) in the migration data. **Stage 2: training a matching model**. We then train a matching model for identifying the matching relations of the widgets using all the positive samples in the migration data and all the mined hard-negative samples produced in the first stage (the ratio of the used positive and negative samples is about 1:4 as illustrated in Section 3.4).

We evaluate TEMdroid on 34 real-world apps in eight categories from the SemFinder dataset[3] [9] and the Craftdroid dataset [4]. The experimental results demonstrate that TEMdroid is effective in event matching[4] (i.e., widget matching and target event synthesis) and test case migration. In terms of event matching, TEMdroid achieves a Top1 accuracy of 76%, improving more than 17% over the baseline approaches [19, 35, 46, 58]. In terms of test case migration, the F1-score for TEMdroid is 89%, outperforming the baseline approach by 7%. Moreover, we evaluate the usefulness of TEMdroid on 5 highly popular industrial apps (beyond the preceding 34 apps from the existing datasets). TEMdroid achieves an F1-score of 87%, demonstrating its usefulness in real-world scenarios.

This paper makes the following main contributions:

- The first learning-based widget matching approach named TEMdroid[5] for migrating test cases across apps.
- Empirical evaluations on real-world apps demonstrating the effectiveness and usefulness of TEMdroid.

2 RUNNING EXAMPLE

Figure 1 shows the migration of a test case for registration from a source app to a target app. This example is adapted from test cases for shopping apps in the Craftdroid dataset [4]. We tailor off some widgets for ease of presentation.

Specifically, associated with two GUI screens (S1 and S2), the source test case includes six events and one assertion. A user inputs different information to the five widgets (S1-1 to S1-5) and generates five events. Then the user clicks the "Create Account" button (S1-6) to generate the sixth event. The assertion checks for successful completion of the registration process, represented by a transition to a new screen (S2) with the username (S2-1). Associated with three GUI screens (T1 to T3), the migrated target test case, which implements similar functionalities, includes five events corresponding to five widgets (T2-1 to T2-5), and one assertion associated with a widget (T3-1).

[1] Positive samples are should-be-matched widget pairs and negative samples are should-not-be-matched widget pairs.

[2] The idea of mining hard samples is borrowed from computer vision [71, 76, 80].

[3] We refer to the dataset provided by Marini et al. [58] as "the SemFinder dataset", and the empirical study provided by them as "the SemFinder study". We use "SemFinder" to denote the matching approach provided by them.

[4] We evaluate TEMdroid and the baselines in event matching rather than in widget matching, as event matching is the common evaluation task used in related work [58]. Section 4 discusses the differences between event matching and test case migration.

[5] TEMdroid's artifacts are available in our repository [10].

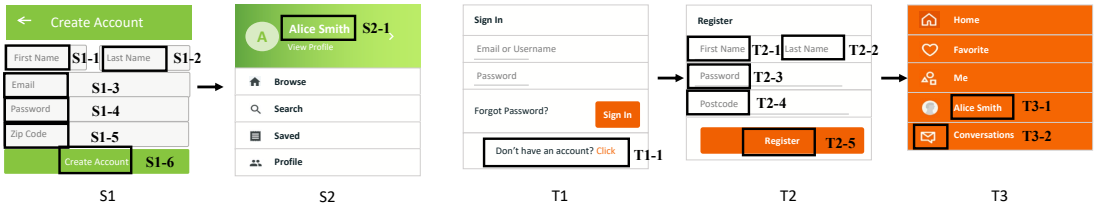
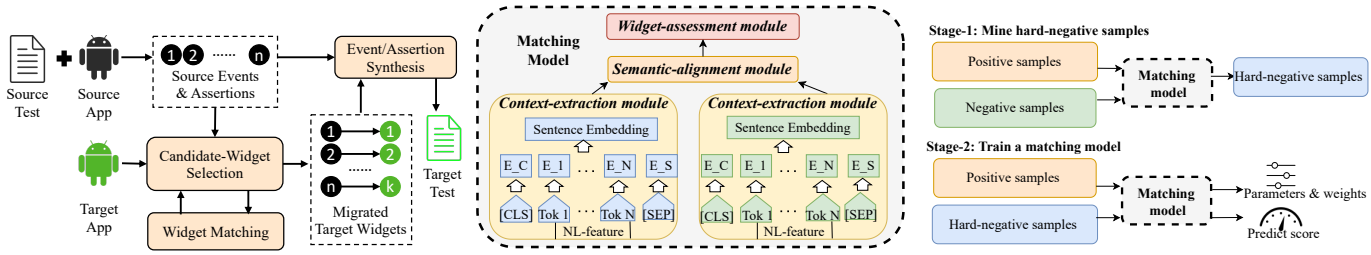


Figure 1: Excerpted registration processes on two shopping apps.



(a) Workflow of TEMdroid **(b) Main architecture of TEMdroid's matching model** **(c) Training strategy of TEMdroid's matching model**

Figure 2: TEMdroid.

During test case migration, widget semantics is typically captured as sequences of words; however, using contextual information is important here. In particular, the same word may contain different semantics within different widgets due to distinct contexts. For example, the semantics of "Create" in the common context is to build something, being different from the semantics of "Register". However, within shopping apps, the combined use of "Create Account" (S1-6) shares similar semantics as "Register" (T2-5).

3 TEMDROID

Given a source app, a source test case, and a target app as input, TEMdroid (whose workflow is shown in Figure 2a) outputs a migrated test case for the target app using three components. First, for each source event or each source assertion in the source test case, the component of candidate-widget selection (Section 3.1) identifies a sequence of candidate widgets from the target app. Second, the component of widget matching (Section 3.2) identifies matched target widgets from the candidate widgets by a learning-based matching model. Third, the component of event/assertion synthesis (Section 3.3) synthesizes target events or target assertions based on the matched target widgets and corresponding source events or source assertions. The synthesized target events and the target assertions are used to form the target test case.

3.1 Candidate-Widget Selection

Aiming to identify candidate widgets in the target app for further widget matching, TEMdroid's component of candidate-widget selection is based on ATM [19], but it explores the target app dynamically rather than requiring prior static analysis of the target app. This strategy makes TEMdroid more practical for real-world applications where access to the source code may not be feasible.

Given a source test case, TEMdroid extracts a sequence of source events and source assertions and launches the target app. The current GUI screen is the landing screen. For each source event or source assertion, TEMdroid tries to find a matched target widget in the current GUI screen of the target app, according to TEMdroid's matching model. Details of the matching model are shown in Section 3.2. After finding the matched target widget, TEMdroid updates the current GUI screen.

If TEMdroid does not find any matched target widget in the current GUI screen, it then uses three steps based on two strategies to try to find two candidate widgets and select a matched widget. First, TEMdroid tries to find a target widget in the current GUI screen that matches the subsequent k (a predefined parameter) source events or source assertions. Specifically, among the subsequent k source events or source assertions, TEMdroid sequentially tries to find the first target widget that matches the source event or source assertion and saves the similarity score between the target widget and the source event or source assertion obtained from the matching model. The target widget is *the selected widget by the first strategy*. Second, in the directly or indirectly reached screens of the current GUI screen, TEMdroid tries to find a target widget that matches the current source event or source assertion. Specifically, TEMdroid leverages the technique of breadth-first search to explore directly or indirectly reached screens of the current GUI screen within a time limit and tries to find the first target widget that matches the source event among these screens. TEMdroid also saves the similarity score of the target widget. The target widget is *the selected widget by the second strategy*. Third, after trying both strategies, if TEMdroid finds two target widgets, TEMdroid selects the target widget with a higher similarity score as the matched target widget. If TEMdroid only finds one target widget, the target

widget is deemed the matched target widget. Otherwise, TEMdroid does not find any matched target widget for this source event or source assertion.

3.2 Widget Matching

This section presents the process of training a matching model, which comprises four main parts, for widget matching. First, Section 3.2.1 introduces the inputs used in the matching model. Second, Section 3.2.2 explains the neural architecture of the matching model. Third, in Section 3.2.3, we describe the methodology used to train the matching model. Fourth, Section 3.2.4 illustrates the training process for the matching model.

3.2.1 Feature extraction. TEMdroid extracts text features to compare two widgets with two steps. First, TEMdroid extracts the *widget text*, the *widget explanation*, and the *widget identity* from each widget. For each GUI widget w, TEMdroid obtains the preceding three features of w according to `text`, `content-desc`, and `resource-id` in the GUI layout file of the GUI screen containing w. Second, the textual features are concatenated and fed into TEMdroid's matching model. Note that, some image widgets have textual information that cannot be extracted from GUI layout files. In this way, we use Tesseract OCR [11] to extract the textual information of image widgets from the screenshots of the GUI screens.

If all the three features (i.e., the widget text, widget explanation, and widget identity) are empty for a widget w, TEMdroid uses the neighbor text to represent this widget with two steps. First, TEMdroid extracts a sequence of candidate widgets that belong to the father and the children widgets of w. These widgets can be identified in the GUI layout file. The candidate widgets are required to comprise widget texts. Second, from the candidate widgets, TEMdroid selects a widget that is closest to w. The widget text of this widget is used as the neighbor text of w. For example, the neighbor text of the conversation icon (T3-2) is `Conversations`.

3.2.2 Model architecture. As illustrated in Figure 2b, the main architecture of TEMdroid comprises three distinct but interconnected modules. Initiated with a pair of text features representing two widgets, TEMdroid first processes these inputs into high-dimensional vectors through the *context-extraction module*. Second, these vectors are aligned in the same semantic space via the *semantic-alignment module*. Third, the *widget-assessment module* calculates a similarity score for the two widgets.

Context-extraction module. This module aims to generate context-aware embeddings for the text features of each widget, thus using its contextual information in widget matching. Specifically, this module employs two key mechanisms: the *encoding mechanism* and the *pooling mechanism*.

Encoding mechanism. To integrate contextual information, TEMdroid requires the use of a context-comprehending pre-trained model. For this purpose, we use BERT as the encoding mechanism due to its widespread popularity as a pre-training model for context understanding in natural language processing [52, 70] and software engineering [22, 45, 81].

BERT [26] uses a self-attention [79] mechanism to encode contextual information. The self-attention mechanism is computed as Eqn. (1). Here, Q, K, and V refer to the query, the key, and the value vector, respectively. Matrix A represents the attention score.

$$A = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d}}\right) \qquad Y = AV \tag{1}$$

In TEMdroid, we obtain the textual information for each widget from the feature extraction step (see Section 3.2.1) as a word sequence, and send the word sequence into BERT. The embedding of each word in a sequence is integrated with the remaining words' embeddings, enabling the representation of each word to contain contextual information.

Pooling mechanism. To leverage the embeddings generated from the encoding mechanism for the matching task, a pooling mechanism is needed to combine these embeddings into a high-dimensional vector representing the widget. Instead of employing max or mean pooling mechanism, TEMdroid employs sentence embedding as the pooling mechanism to better capture the widget semantics.

Sentence embedding [60, 73] (originated from BERT [20]) represents a word sequence as a fixed-length vector. The key insight is that the semantics of a sentence is constructed from the semantics of individual words and their relations with each other. For this purpose, the authors of BERT design a unique token (i.e., "[CLS]") to symbolize the word sequence during the pre-training phase. Motivated by the sentence embedding, TEMdroid adopts the output embedding of the "[CLS]" token as each widget's embedding. Various applications [16, 36] with BERT also employ sentence embeddings to capture the semantics of word sequences and achieve promising effectiveness in downstream tasks.

Semantic-alignment module. For a fair comparison between two widgets in the same semantic space, we design a *semantic-alignment module* that aligns the semantics of the two widgets via weight sharing and non-linear transformation.

Siamese network. TEMdroid uses a Siamese network to facilitate weight sharing between the context-extraction modules of source widgets and target widgets. In TEMdroid, the Siamese network, a prevalent network architecture for matching tasks [25, 61, 64], uses two identical sub-networks [25] to process the textual features of both the source and the target widgets. The Siamese network shares parameters between the two sub-networks, thereby transforming the representations of both widgets into a common semantic space. This mechanism ensures that both widgets are compared under equivalent conditions.

Transformation layer. To learn non-linear relations, a Multilayer Perceptron (MLP) is typically integrated prior to the widget-assessment module [34, 37]. The MLP in TEMdroid has 1 hidden layer with *hidden_kernel*, using ReLU [63] as the activation.

Widget-assessment module. This module is to calculate a similarity score based on the final representations of two widgets. Following previous matching models [31, 44, 48], TEMdroid uses the cosine distance between the generated vectors from MLPs as the similarity assessment metric. The cosine similarity score is 1 for two identical vectors and 0 for two totally different vectors.

In general, given the two widgets with textual information t_1, t_2, Eqn. (2) obtains the semantic representation of each widget through the context-extraction module and the semantic-alignment module. The similarity of the two widgets is then calculated in the widget-assessment module using Eqn. (3), where θ is a collection of all shared parameters including BERT parameters θ_{BERT}, MLP weights

W_1, W_2, b_1, and b_2.

$$E(t, \theta) = \text{ReLU}\left(\text{BERT}\left(t, \theta_{\text{BERT}}\right) W_1 + b_1\right) W_2 + b_2 \quad (2)$$

$$s_i = \cos\left(E\left(t_{i,1}, \theta\right), E\left(t_{i,2}, \theta\right)\right) \quad (3)$$

3.2.3 Training and Inference. This section illustrates training and inference in TEMdroid.

Training. The training process of TEMdroid's matching model involves a fine-tuning process of BERT. Here, we describe the fine-tuning process for a pre-trained model. Notably, a pre-trained model has been trained on a large corpus incorporating various self-supervised tasks. Fine-tuning is to adjust the pre-trained model's parameters using a smaller learning rate and a domain-specific corpus. This approach requires fewer domain-specific data and reduces training time compared to learning from scratch [26].

To enhance BERT's comprehension of the semantics of widgets in widget matching, we fine-tune a pre-trained BERT using migration data for GUI test cases during the training process with two steps. First, the matching model loads the pre-trained parameters of BERT and adjusts these parameters via backpropagation with migration data. This way helps BERT exploit both the knowledge of the pre-trained language model and that specific to test case migration within a short time and with fewer data. Second, TEMdroid uses the mean squared error (MSE) loss to calculate the distance between the predicted results s_i and the ground-truth label l_i. This way helps optimize the matching model.

$$\text{minimize} \sum_i |s_i - l_i|^2 \quad (4)$$

Inference. For each pair of two widgets, TEMdroid inputs two sequences of textual information to the matching model. If the similarity score is above a threshold, TEMdroid then predicts that the widget pair is a matched widget pair. Otherwise, TEMdroid deems the widget pair not matched.

3.2.4 Two-stage training strategy. In widget matching, positive samples are much fewer than negative samples. For example, in Figure 1, the should-be-matched widget of "Create Account" (S1-6) is "Register" (T2-5). In this case, there is only one positive sample, i.e., "Create Account" and "Register". However, "Create Account" and all other widgets in the target app form negative samples. It is not practical to train a model with all positive and negative samples. To train a matching model, we need to balance the number of positive samples and negative samples.

Inspired by the idea of mining hard samples (i.e., the samples that are easily misjudged) explored in computer vision [71, 76, 80], we use a *two-stage training strategy* to mine hard-negative samples and train TEMdroid's matching model using mined hard-negative samples and all the positive samples in the migration data. The overview of our two-stage training strategy is shown in Figure 2c.

Stage 1: training and applying a hard-negative sample miner. We train a model as a hard-negative sample miner using all the positive samples and an equal number of randomly selected negative samples in the migration data. This model architecture is the same as the architecture in Section 3.2.2.

We further construct a candidate negative set for mining the hard-negative samples. For each positive sample $\langle w_s, w_t \rangle$, we generate a list of negative samples $\langle w_s, w_t' \rangle$, where w_t and w_t' are in the same GUI screen. We use the trained hard-negative sample miner to predict the matching relations for the negative samples in the candidate negative set. The *hard-negative samples* are those misjudged by our hard-negative sample miner.

Stage 2: training a matching model. We train a model as TEMdroid's matching model using all the positive samples in the migration data and all the hard-negative samples mined in the first stage. The training process uses stochastic gradient descent with the Adam optimizer.

3.3 Event/Assertion Synthesis

TEMdroid leverages a sequence of matched widget pairs along with their corresponding source events (comprising widgets, actions, and optional input values) or source assertions (comprising widgets, conditions, and optional text properties) to synthesize target events and target assertions that can be used to form the target test cases. **Event synthesis.** For each matched widget pair and its corresponding source event, TEMdroid uses the source action, source input value, and target widget to synthesize the target event. If the target event cannot be executed in the target app, TEMdroid selects other common actions of the widget type that the target widget belongs to. For example, in a shopping cart of a shopping app, removing a product widget with a `TextView` widget type may be performed by a swipe, while the same task in another shopping app might be performed by a long click. To overcome this challenge, TEMdroid considers common actions of the widget type and finds the one that properly works on the widget in the target app as the action of the target event. **Assertion synthesis.** For each matched widget pair and its corresponding source assertion, TEMdroid employs the source condition, source property, and target widget to synthesize the target assertion.

3.4 Implementation

We implement TEMdroid in Python. For the component of candidate-widget selection, TEMdroid sets the forward-search steps k to be 2 and uses a Pixel 3 Emulator running Android 6.0 to explore apps. The matching model of TEMdroid is implemented based on PyTorch [8] and the pre-trained BERT is from Huggingface [3]. We set the *hidden_kernel* to be 700. To train a hard-negative sample miner, we use all the positive samples and an equal number of negative samples in the migration data. The number of hard-negative samples mined by the hard-negative sample miner is four times the number of positive samples. The training process of TEMdroid's matching model uses one NVIDIA TITAN RTX GPU.

4 EVALUATION

Event matching, which involves widget matching and target event synthesis, is the central step of test case migration [58]. Thus, we use event matching to evaluate the effectiveness of TEMdroid's matching model. Moreover, we assess the impact of each technique of TEMdroid's matching model in event matching. Since the goal of TEMdroid is to migrate test cases, we also evaluate the effectiveness of TEMdroid in test case migration.

Based on the preceding analysis, we design the following research questions to evaluate the effectiveness of TEMdroid in event matching and test case migration.

Table 1: Statistics of benchmark apps.

Dataset	Category	App	Test	Event	Assertion	Ave_size
SemFinder	Expense	4	4	27	-	3.6M
	Note	3	3	12	-	2.1M
	Shopping-list	4	4	21	-	2.0M
	Browser	4	4	15	-	4.8M
	To-do	5	5	23	-	1.6M
	Mail	2	3	12	-	15.9M
	Calculator	5	5	18	-	1.7M
	Total	**27**	**28**	**128**	-	-
Craftdroid	Browser	5	10	22	15	4.3M
	To-do	5	10	24	10	1.6M
	Shopping	4	8	32	8	25.3M
	Mail	4	8	20	12	11.0M
	Calculator	5	10	19	5	1.7M
	Total	**23**	**46**	**117**	**50**	-

RQ1: *How does TEMdroid perform in event matching?*
RQ2: *How do TEMdroid's techniques affect event matching?*
RQ3: *How effective is TEMdroid in test case migration?*

4.1 Experimental Setup

Experimental subjects. There are some datasets [2, 4, 5, 9] available for evaluating event matching and test case migration. In terms of event matching, we use the SemFinder dataset [9], as it is specifically designed for this purpose. The SemFinder dataset includes popular apps with more than 1000 downloads in Google Play [6]. In terms of test case migration, we use the Craftdroid dataset [4], a popular dataset for test case migration that has been used in previous studies [47, 50, 58]. According to Craftdroid [46], the Craftdroid dataset [4] consists of 25 apps from five typical categories used in mobile testing [55, 56, 69].

For the selection of apps, we consider all the installable apps provided by the SemFinder dataset and the Craftdroid dataset as our experimental subjects[6]. Therefore, we exclude three outdated apps from our experimental subjects. The apps in our experimental subjects cover eight categories, representing the broadest range of app categories compared with the experimental subjects used in related studies [19, 35, 46]. The basic statistics of our experimental subjects, including the average app size for each category, are displayed in Table 1. Detailed app information can be found in our repository [10] due to space limit.

Ground-truth. The Craftdroid dataset provides the test case migration relations from source apps to target apps with 674 should-be-matched event pairs and 291 should-be-matched assertion pairs. The SemFinder dataset provides event matching relations with 295 should-be-matched and 4,649 should-not-be-matched event pairs.

Dataset division. For event matching, we use a 5-fold cross-validation strategy [21] to train TEMdroid's matching model. We also ensure that the target app never appears in the training set during testing. Specifically, we randomly divide the apps into five folds, with each fold containing 20% of the apps. During each round of cross-validation, we use the apps in one fold as the test apps and the apps in the other four folds as the training apps. When training TEMdroid in each round, we only keep the widget pairs where both the source widgets and the target widgets are in the training apps. On the other hand, when evaluating TEMdroid, we use the widget pairs where the target widgets are in the test apps. We also choose 20% of the training set as the validation set to select the hyper-parameters in Section 3.4. The division of apps into five folds is at [10].

For test case migration, we also randomly divide the apps into five folds. Each target app is assigned to a unique test set, and the matching model trained for this test set is used for test case migration of the target app.

Baselines. For event matching, SemFinder [58] is the state-of-the-art approach. Additionally, the matching results of ATM [19] and Craftdroid [46] on the SemFinder dataset are publicly available. Therefore, we compare TEMdroid with SemFinder [58], ATM [19], and Craftdroid [46] on the SemFinder dataset for event matching. For test case migration, Craftdroid [46] is the state-of-the-art approach. Since there are no other approaches with published results on the Craftdroid dataset, we compare TEMdroid solely with Craftdroid [46] for test case migration.

Evaluation metrics. In event matching, we assess TEMdroid and compare it with baseline approaches, employing two widely recognized statistical metrics for matching tasks [24, 58], namely MRR and Top1, which have also been adopted in previous studies [58]. MRR (i.e., mean reciprocal rank) is the average of the reciprocal ranks of total queries Q. On the other hand, Top1 accuracy is the ratio of queries where the ground-truth has the highest similarity score in the returned list of event pairs.

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{5}$$

$$Top1 = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\begin{cases}1, & \text{if } rank_i = 1\\ 0, & \text{otherwise}\end{cases} \tag{6}$$

In test case migration, given a source event from a source test case and a target app, existing migration approaches try to find a matched target widget and synthesize a target event. A synthesized target event can report three kinds of results. (1) A synthesized target event is an event in the ground-truth (*TP*). (2) A synthesized target event cannot be found in the ground-truth (*FP*). (3) A target event in the ground-truth is not found by existing migration approaches (*FN*). The synthesized assertion can also report the preceding results. To study the effectiveness of TEMdroid and compare it with the baseline approach, we use the same three metrics (i.e., Precision, Recall, and F1-score). These metrics have been used in previous studies [19, 46, 85] to measure the effectiveness of test case migration.

$$Precision = TP\ /\ (TP + FP) \tag{7}$$

$$Recall = TP\ /\ (TP + FN) \tag{8}$$

$$F1 = 2 * Precision * Recall\ /\ (Precision + Recall) \tag{9}$$

[6]The SemFinder dataset and the Craftdroid dataset share 16 common apps, leading to 34 distinct popular industrial applications in our experimental subjects.

Table 2: Evaluation of TEMdroid in event matching.

Category	TEMdroid_E		TEMdroid_G	
	Top1	MRR	Top1	MRR
Expense	71%	81%	45%	63%
Note	85%	93%	55%	69%
Shopping-list	55%	73%	50%	63%
Browser	94%	94%	62%	76%
To-do	69%	81%	59%	74%
Mail	100%	100%	90%	95%
Calculator	83%	92%	56%	66%
Total	**76%**	**85%**	**57%**	**71%**

Table 3: Matching effectiveness of TEMdroid and baselines.

Approach	TEMdroid_E	TEMdroid_G	SemFinder	ATM	Craftdroid
Top1	**76%**	57%	65%	45%	45%
MRR	**85%**	71%	79%	67%	65%

4.2 RQ1: Event Matching

To answer this question, we evaluate the effectiveness of TEMdroid within the same categories (Section 4.2.1) and compare it with baselines based on the SemFinder dataset. Furthermore, we evaluate the generalizability of TEMdroid across app categories (Section 4.2.2).

4.2.1 Effectiveness of TEMdroid. This section presents the results of TEMdroid in event matching with a case study and a false analysis.

Matching results. TEMdroid's results in event matching for the SemFinder dataset are shown in the column "TEMdroid_E" (i.e., the effectiveness of TEMdroid) of Table 2. Baseline results (i.e., SemFinder [58], ATM [19], and Craftdroid [46]) are shown in Table 3. TEMdroid achieves a Top1 accuracy of 76% and an MRR of 85%, surpassing the baselines by more than 17% and 8%, respectively.

These results indicate that TEMdroid is effective in event matching and outperforms the baseline approaches. Following the finding of the previous study [58], Top1 accuracy is more important to evaluate event matching approaches than MRR, as event matching is a core component of test case migration, and migration approaches often choose the Top1 events as the matched events. TEMdroid substantially improves Top1 accuracy compared to the baselines, indicating that TEMdroid's matching model is more suitable for test case migration.

We observe that the effectiveness of event matching for TEMdroid varies across different categories, with Top1 accuracy ranging from 100% for Mail apps to 55% for Shopping-list apps, which are similar to baselines [19, 46, 58][7]. We notice that Mail apps typically have a simple main screen with few actionable widgets to write and check emails. This simplicity in app design makes event matching easier. On the other hand, Shopping-list apps include multiple screens and widgets that implement different but relevant functionalities. This complexity makes it more challenging to find correct matches.

Case study. To better understand why TEMdroid outperforms the baselines, we delve into specific cases where TEMdroid achieves accurate matches whereas baselines do not. Table 4 provides some instances, showing the textual information extracted from the widgets, the similarity scores computed by TEMdroid and SemFinder (the state-of-the-art approach), and the corresponding ground-truth matching results (i.e., "Yes" for should-be-matched pairs and "No" for should-not-be-matched pairs).

[7]The detailed results of baseline approaches can be found in our repository [10].

Table 4: Examples of the event matching results.

Source	Target	TEMdroid	SemFinder	Match
note body edit	note content	**0.987**	0.289	Yes
guest number edit text	bill content main	**0.002**	0.204	No

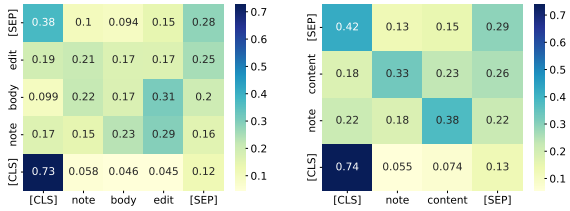


Figure 3: Attention weight visualization for TEMdroid.

The widgets of the first example are derived from Note apps. Despite "body" or "edit" lacking semantic similarity to "content" in isolation, the combined use of "body edit" often indicates writing texts in a Note app. This context is similar to "content" (i.e., the texts of a note). TEMdroid can match the two widgets correctly by using contextual information.

To substantiate this hypothesis, we visualize the attention weights of the first hidden layer in TEMdroid's matching model for these widgets, as depicted in Figure 3. Notably, the left part of Figure 3 reveals "note" directing attention to both "body" and "edit", whereas in the right part, "note" pays attention to "content". In these contexts, "body edit" and "content" are likely to have similar semantics. Thus, these visualizations support our hypothesis.

To further validate that TEMdroid's matching model effectively captures contextual information rather than solely relying on specific word correspondence, we also analyze widget pairs that share common words but should have opposite matching relations. The widget pairs in the first and second rows of Table 4 demonstrate such instances. Despite having the common words "edit" and "content", these pairs have contrasting matching relations. TEMdroid accurately predicts the correct results for these pairs, whereas SemFinder fails. Additionally, we also encounter widget pairs with the same word "note" that should not match. In these situations, TEMdroid also correctly predicts the matching results. These examples illustrate that TEMdroid primarily determines widget matching based on contextual information.

False analysis. To understand the weakness of TEMdroid, we manually analyze 25% of the samples that the Top1 event pairs are incorrect in the SemFinder dataset. The main reasons are as follows.

First, our matching model misunderstands some related but different widgets (e.g., the "confirm email" widget and the "email" widget). Wrongly understanding textual information makes TEMdroid fail. For instance, some apps include registration screens.

Table 5: Contributions of each technique in TEMdroid.

Technique	Top1	MRR
TEMdroid	**76%**	**85%**
w/o Siamese network	49%	67%
w/o Fine-tuning BERT	37%	54%
w/o Contextual information	58%	73%
w/o Hard negative samples	37%	52%

The registration screens often include an "email" widget for users to input email addresses, and a "confirm email" widget to verify these inputs. These two widgets, although related, are of distinct semantics and functionalities and should not be matched to each other. Relying on the keyword "email" with insufficient consideration of the semantics of the term "confirm" may lead the matching model to match them incorrectly. Second, the textual information extracted from some widgets cannot represent the semantics of the widgets, leading to our matching model not being able to match the should-be-matched widgets.

4.2.2 Generalizability of TEMdroid. To evaluate the generalizability of TEMdroid across app categories, we divide the app categories of the SemFinder dataset (outlined in Table 1) into two parts: the first part consists of the first three categories, while the second part includes the remaining four categories. We train TEMdroid on one part and test it on the other part to ensure that the app categories of the test data have not appeared in training.

Matching results. The results of this experiment, as displayed in the column "TEMdroid_G" (representing the generalizability of TEMdroid) of Table 2, show that TEMdroid achieves a Top1 score of 57% and an MRR score of 71% when *the categories of the test apps have not appeared in training*. Furthermore, we compare these results with the baselines [19, 46, 58] for event matching (see Table 3) within the same categories (i.e., *the categories of the test apps have appeared in training*). Even in this unfair setting, TEMdroid outperforms both ATM and Craftdroid, and is comparable to SemFinder. These results demonstrate that TEMdroid performs well to adapt across different app categories.

Based on the preceding analysis, we draw the following conclusion to RQ1: *TEMdroid is effective and substantially outperforms the baselines in event matching. In addition, TEMdroid has good generalizability across different app categories.*

4.3 RQ2: Ablation Study of TEMdroid

We evaluate the effectiveness of the main techniques used in TEMdroid on the SemFinder dataset. These techniques include the use of a Siamese network, fine-tuning BERT, using contextual information, and mining hard negative samples.

Experimental setting. To assess the effectiveness of the main techniques used in TEMdroid, we conduct various experiments on the SemFinder dataset. First, we train a non-Siamese network, of which the two BERTs and the two MLPs do not share weights, to evaluate the effectiveness of selecting the Siamese network in TEMdroid. Second, we evaluate the effectiveness of the fine-tuning process of BERT by training a Siamese network with fixed parameters of the pre-trained BERT. Third, to evaluate the effectiveness of using contextual information, we train a matching model, where we remove BERT's attention mechanism and directly output the pre-trained BERT's word embeddings as the final representations. Based on this modification, the embedding of the first token "[CLS]" becomes a constant value, so we have to adjust the representation approach by using the average embedding of all words to represent the semantics of the whole sentence. Fourth, we evaluate the effectiveness of the mined hard-negative samples by training a matching model with an equal number of randomly selected negative samples, using the same model architecture as TEMdroid.

Results for ablation study. The results of the ablation study are presented in Table 5. The first row displays the overall effectiveness (76% Top1) of TEMdroid in event matching, being discussed in Section 4.2. The subsequent rows show the effectiveness of TEMdroid's ablation models. The results indicate that the mined hard-negative samples (37% Top1) and fine-tuning process (37% Top1) have the most substantial impact on improving event matching. Additionally, contextual information (58% Top1) and the Siamese network (49% Top1) are crucial for enhancing effectiveness.

Based on the preceding analysis, we draw the conclusion to RQ2: *TEMdroid's techniques substantially contribute to event matching.*

Table 6: Migration effectiveness of TEMdroid and Craftdroid.

Approach	Category	Type	Precision	Recall	F1
TEMdroid	Browser	Event	100%	100%	100%
		Assertion	100%	100%	100%
		All	100%	100%	100%
	To-do	Event	76%	90%	82%
		Assertion	70%	100%	82%
		All	74%	93%	82%
	Shopping	Event	75%	82%	78%
		Assertion	83%	71%	76%
		All	76%	80%	78%
	Mail	Event	88%	100%	94%
		Assertion	88%	100%	94%
		All	88%	100%	94%
	Calculator	Event	87%	90%	88%
		Assertion	100%	100%	100%
		All	90%	92%	91%
	Total	Event	85%	92%	88%
		Assertion	89%	96%	92%
		All	**86%**	**93%**	**89%**
Craftdroid	Total	Event	71%	93%	81%
		Assertion	90%	89%	90%
		All	**77%**	**91%**	**83%**

4.4 RQ3: Test Case Migration

We evaluate the effectiveness of TEMdroid in test case migration and compare it with a baseline approach on the Craftdroid dataset.

Migration results. Table 6 presents the effectiveness of TEMdroid and Craftdroid in test case migration. TEMdroid achieves a

precision of 86%, a recall of 93%, and an F1-score of 89%, which are 12%, 2%, and 7% higher than Craftdroid, respectively.

In test case migration, precision is a crucial metric because it measures the accuracy of the generated test cases. A high precision score indicates that the generated test cases contain fewer error events and assertions, and are more likely to be executed without human intervention. In contrast, a low precision score implies that the generated test cases are less accurate and require more human effort for revision. Therefore, improving precision is important to reduce human effort and ensure the quality of the generated test cases. Notably, TEMdroid's precision score (i.e., 86%) is 12% higher than Craftdroid, indicating that it is more likely to generate fully correct test cases without the need for human intervention. This characteristic makes TEMdroid a valuable tool for developers looking to reduce their workload. Regarding recall improvement, the baseline approach achieves a high recall of 91%, making it more challenging to improve upon. However, TEMdroid still manages to improve recall by 2%, demonstrating its effectiveness in test case migration. Overall, these results highlight the potential superiority of TEMdroid for test case migration.

Although TEMdroid's improvement over Craftdroid is not as substantial as in event matching, it still shows a promising improvement in precision, recall, and F1-score. The reason for this difference in improvement is that TEMdroid mainly focuses on the improvement of the component of widget matching, which is responsible for matching source widgets to target widgets, whereas the component of candidate-widget selection has fewer rules than Craftdroid. This limitation makes TEMdroid sometimes unable to reach the correct GUI screens. Adding more exploration rules could further enhance TEMdroid's effectiveness.

Efficiency Study. We compare the runtime information between TEMdroid and Craftdroid on the Craftdroid dataset. The training time of TEMdroid on the Craftdroid dataset is approximately 34 minutes, which is a one-time labor-intensive task. Once trained, TEMdroid is able to migrate a test case with an average of 9 minutes. In contrast, the baseline approach Craftdroid, as reported in [46], has an average runtime of approximately 89 minutes for each test case migration. Therefore, TEMdroid offers a substantial speed improvement compared to Craftdroid.

4.5 Threats to Validity

A possible threat to external validity is the generalization to other mobile apps and test cases. We mitigate this threat by using the most app categories compared with the experimental subjects of existing studies [19, 35, 46]. A possible threat to the internal validity is the possible mistakes involved in our implementation and experiments. We mitigate this threat by manually inspecting our results and analyzing the incorrect matching results. A possible threat to construct validity is evaluation metrics. We mitigate this threat by using the same evaluation metrics as existing research [19, 46, 58, 85].

Based on the preceding analysis, we draw a conclusion to RQ3: *TEMdroid is effective and outperforms baseline in test case migration.*

5 USEFULNESS STUDY

TEMdroid has achieved high effectiveness in test case migration using the Craftdroid dataset. However, there may be a difference between the evaluations on this dataset and in real-world scenarios. The evaluation on the Craftdroid dataset involves comparing the migrated test cases with the ground-truth test cases that the dataset provides. Conversely, in real-world scenarios, the ground-truth test cases are not available. Users evaluate the effectiveness of migration approaches according to the modifications of the migrated test cases. To further evaluate the effectiveness of TEMdroid, we conduct a study to assess the usefulness of TEMdroid in test case migration with a real-world scenario. In this study, we migrate test cases from the Craftdroid dataset to popular industrial apps in Google Play [6].

Experimental setup. To migrate test cases of apps from the Craftdroid dataset to apps in Google Play [6], we identify a set of target apps in Google Play whose functionalities are similar to those in the Craftdroid dataset. Specifically, we first maintain an app collection for each category of the Craftdroid dataset [4]. For each app in one category, we search for the top ten similar apps available in Google Play [6] to the app collection. Second, to determine the target app for the category, we select the app with the highest number of appearances in the app collection. In case multiple apps have the same maximum appearances, we consider the app with the highest number of downloads as the final choice. Third, the five selected target apps with app sizes for the five categories are shown in Table 7. In this usefulness study, we evaluate the effectiveness of TEMdroid by migrating test cases from the Craftdroid dataset to these target apps.

To fairly compare the migration results on the Craftdroid dataset (in Section 4.4) with those in this study, we select the model with the lowest accuracy among the five models used in Section 4.4. Instead of retraining a new model, we select one of the five models to ensure the same training sizes of the matching models for both experiments. Furthermore, by selecting the lowest accuracy model, we aim to avoid bias that could potentially favor TEMdroid.

For each source test case, TEMdroid automatically migrates it into a test case for a corresponding target app. However, as some migrated test cases may not work in the target apps, users typically need to modify the migrated test cases to align them with the desired functionalities while minimizing the number of modifications. Moreover, the number of modifications measures the effectiveness of TEMdroid. To simulate this manual process, one author manually modifies the migrated test cases if necessary. To assure the correctness of modifications, two other authors evaluate the modification process individually. In case of disagreement, the three authors discuss the disagreement to reach a consensus.

Evaluation metrics. We evaluate the usefulness of TEMdroid according to the differences between the modified test cases and those directly migrated by TEMdroid, using Precision, Recall, and F1-score. Following the FrUITeR study [85], we also calculate a *reduction score* to assess the manual effort saved through TEMdroid compared with writing desired test cases from scratch. Based on the Levenshtein distance [41], we first compute the number of steps ($Step_m$) required to modify a migrated test case to reach the desired test case. In this context, a step in the Levenshtein distance refers to an insertion, a modification, and a deletion of an event or an assertion. Second, we compute the number of steps ($Step_d$) required to write the desired test case from scratch. Third, we assess the ratio of steps saved by employing the migrated test cases compared with writing the desired test cases from scratch.

Table 7: Results of the usefulness study in TEMdroid.

App	Size	Type	Precision	Recall	F1	Reduction
Web Browser	5M	Event	92%	96%	94%	93%
		Oracle	100%	95%	97%	97%
		All	96%	96%	96%	95%
Done	1.6M	Event	76%	96%	85%	75%
		Oracle	67%	100%	80%	75%
		All	73%	97%	84%	75%
Fivemiles	25.4M	Event	62%	97%	75%	63%
		Oracle	60%	92%	73%	60%
		All	61%	95%	75%	62%
Pro Mail	98.2M	Event	85%	96%	90%	89%
		Oracle	100%	96%	98%	96%
		All	92%	96%	94%	93%
Tip Calculator	4.3M	Event	100%	92%	96%	90%
		Oracle	89%	89%	89%	80%
		All	91%	91%	91%	85%
Total	134.5M	Event	79%	95%	86%	83%
		Oracle	84%	95%	89%	82%
		All	**80%**	**95%**	**87%**	**82%**

$$Reduction = (Step_d - Step_m) \,/\, Step_d \tag{10}$$

Migration results. In this study, TEMdroid achieves an F1-score of 87% (see Table 7), which closely aligns with its effectiveness (an F1-score of 89%) on the Craftdroid dataset (see Table 6). Furthermore, TEMdroid saves 82% of manual effort compared with writing the desired test cases from scratch. Our thorough analysis, which involved manually examining all inaccurately predicted samples, has revealed two primary reasons. Predominantly, the inaccuracies are related to incorrect widget matching. Additionally, there are also a smaller number of errors stemming from the incorrect selection of candidate widgets. By recognizing these primary sources of inaccuracies, we can pinpoint areas for potential refinement and improvement in TEMdroid. Overall, this study demonstrates the strong usefulness of TEMdroid in real-world scenarios.

While this experiment demonstrates the usefulness of TEMdroid, there is room for further validation. One potential direction for validating the effectiveness of TEMdroid is to conduct a controlled experiment involving two groups of developers. In this controlled experiment, one group of developers generates target test cases based on migrated test cases, while the other group starts from scratch. By comparing the cost and time-saving between developers in two groups, we could provide more concrete evidence regarding the practical advantages of TEMdroid. Furthermore, expanding the scope of the validation process by incorporating a wider range of apps could help validate the usefulness of TEMdroid across various application domains and scenarios, providing a more comprehensive understanding of its practical benefits.

6 RELATED WORK

Test case migration across applications. Based on how to achieve widget matching, existing migration approaches can be divided into two categories: classification approaches [35], and matching approaches [19, 46, 50, 59].

There is only one *classification* approach (i.e., AppFlow [35]) for GUI test case migration. AppFlow uses a trained multi-classifier to identify widget labels (e.g., menu widget) of source widgets and target widgets. A source widget and a target widget sharing the same label are considered matched widgets.

The key difference between AppFlow and TEMdroid is the matching process. AppFlow's matching accuracy relies on classification accuracy. Lower classification accuracy of widget labels can cause lower matching accuracy of widgets. On the contrary, TEMdroid trains a matching model to directly identify each should-be-matched widget pair, thus reducing error accumulation.

Aware of the weakness of the classification task, some migration approaches [19, 46, 50, 59] formulate widget matching as a *matching* task. For widget matching, ATM [19] uses word embeddings from word2vec [62] fine-tuned with app manuals to represent each word of widgets. Furthermore, the authors of ATM manually define a matching function to identify matched widgets. That is, the final similarity of the two widgets is the average of the similarities of all word pairs. Compared with ATM, Craftdroid [46] and TRASM (a closed-source approach) [50] use word embeddings from a standard word2vec [62]. Adaptdroid [59] uses word embeddings from a standard word move distance [39] (another word embedding model) to calculate the similarity of two widgets.

There are two key differences between TEMdroid and the existing matching approaches. First, the embeddings from word2vec [62] and word move distance [39], which the existing approaches used, can only represent static word embeddings without contextual information. However, TEMdroid uses BERT to integrate contextual information and uses sentence embeddings to represent widget semantics, thus making widget matching more accurate. Second, these matching approaches rely on manually defined matching functions, limiting their abilities to match complex relations. On the contrary, TEMdroid trains a matching model to identify matching relations, enabling it to handle complex scenarios.

There are also two *empirical studies* [58, 85] related to test case migrations. The FrUITeR study [85] builds a dataset containing 20 real-world apps, and evaluates the effectiveness of ATM [19], Craftdroid [46] and AppFlow [35] in test case migration. The SemFinder study [58] evaluates the effectiveness of ATM [19] and Craftdroid [46] in event matching. There are also three *related approaches*. Rida [43] targets cross-app record and replay. Mao et al., [57] target user-behavior mining and reuse across apps. SemFinder [58] is an event matching approach but not a migration approach like TEMdroid.

Test case migration across platforms. Some approaches migrate test cases across platforms for the same apps. Testmig [66] leverages static analysis to guide event exploration and maps similar events from iOS to Android. MAPIT [77] performs bi-directional test migration between Android and iOS using dynamic analysis. LIRAT [83] leverages computer vision techniques to map similar events across platforms (including Android and iOS). TRANSDROID [47] is capable to migrate test cases from Web to Android. TEMdroid's matching model may also enhance cross-platform test migration by improving event matching accuracy.

Automated testing. Automated testing approaches based on exploration strategies can be classified into four categories: random testing approaches [12, 53, 67], model-based approaches [14, 17, 30, 40, 74, 82, 84, 86], systematic testing approaches [15, 29, 56], and

learning-based approaches [23, 38, 42, 68, 72, 87]. TEMdroid can also be viewed as an automated testing approach that leverages test cases from a source app to test a target app. However, there are two key differences between TEMdroid and existing automated testing approaches. First, the automated testing approaches can generate only events, but struggle to generate oracles. However, TEMdroid can automatically generate both events and oracles if the source test cases contain them. Second, due to the lack of oracle information, the automated testing approaches primarily reveal crash bugs, but rarely reveal functional bugs. Conversely, the functional test cases generated by TEMdroid can act as a supplement to these approaches in revealing functional bugs.

7 DISCUSSION

Pre-trained language models. TEMdroid uses BERT to represent widget semantics due to its ability to perceive contextual information. In natural language processing, there are other BERT-like large language models (LLM), such as RoBERTa [52], DistilBERT [70], and MobileBERT [75]. RoBERTa has the same architecture as BERT with a larger training set. DistilBERT and MobileBERT have similar architectures to BERT with fewer layers. To investigate whether the replacement of LLMs can improve the effectiveness of TEMdroid, we replace BERT with RoBERTa and DistilBERT. We then assess the effectiveness of the revised versions of TEMdroid with their event matching results. The Top1 scores are 75% for RoBERTa and 71% for DistilBERT, which are similar to the effectiveness achieved using BERT (i.e., 76%). The detailed results can be found in our repository [10]. This finding demonstrates that merely replacing the LLMs in TEMdroid does not substantially improve its effectiveness. We plan to explore the design of task-specific pre-training tasks to enhance the effectiveness of TEMdroid.

ChatGPT [1] and GPT-4 [7], two powerful language models, demonstrate remarkable proficiency in comprehending natural languages [32, 88]. However, ChatGPT and GPT-4 cannot be directly replaced or fine-tuned in a way similar to open-source models like BERT. We intend to integrate ChatGPT/GPT-4 into TEMdroid and investigate the potential enhancements to TEMdroid.

Impact of text features on event matching. We have investigated the impact of text features on event matching, specifically focusing on *the lengths of text features* and *the neighbor texts*.

In our analysis of the SemFinder dataset, we examine the effectiveness of TEMdroid based on the lengths of the text features from the source widgets. Our finding shows a fluctuating upward trend between the lengths of the text features and the matching results. Due to space limit, we provide detailed results in our repository [10]. This finding suggests that longer features may provide more information, leading to improved matching accuracy. This interesting phenomenon reinforces us to further explore the way to improve event matching based on enhancing the short text features.

In cases where the widget resources, the widget contents, and the widget texts of widgets are empty, TEMdroid uses neighbor texts as text features. We observe that in the SemFinder dataset, there are only two source widgets for that TEMdroid relies on neighbor texts as the text features. For both source widgets, TEMdroid predicts the correct target widgets. Although the sample size of the widgets using neighbor texts is limited, this finding suggests that the neighbor text can be a valuable feature in cases where the primary text features are not available.

Impact of widget types on event matching. We investigate the effectiveness of TEMdroid's matching model on the widgets whose widget types have not appeared in the training data. Since TEMdroid does not use widget types as a feature, we hypothesize that the effectiveness of TEMdroid's event matching should not be substantially impacted when confronted with widgets whose types are not present in the training data.

To validate this hypothesis, we train TEMdroid on the apps from the "Mail" category in the SemFinder dataset (see Table 1) and test it on the apps from the "Calculator" category. The "Calculator" category includes `Button` widgets, which are not present in the "Mail" category. The results counted according to the source widget types indicate that the `Button` type, despite not being seen during training, achieves a higher Top1 score than the `EditText` type (67% vs. 25%), which has appeared in the training data. This experiment reinforces that TEMdroid's matching effectiveness remains robust even when dealing with widgets of types that are not in training.

Component of candidate-widget selection. Apart from the component of widget matching, existing migration approaches [19, 46, 50, 59] have also devised various strategies for the component of candidate-widget selection. Typically, the authors of these approaches employ one of two main strategies to design this component. ATM [19], TRASM [50], and Craftdroid [46] employ a combination of static analysis and dynamic exploration to select widgets. This combined approach uses the static analysis results to guide dynamic exploration. However, it requires access to the source code of apps. On the other hand, to enhance approach applicability, Adaptdroid [59] and TEMdroid employ purely dynamic exploration strategies, thereby removing the need for source code. Future research on more effective dynamic exploration strategies could help improve the effectiveness of migration approaches.

8 CONCLUSION

In this paper, we have proposed a new approach called TEMdroid for test case migration. TEMdroid is the first widget matching approach that learns a matching model trained on migration data for GUI test cases. We have evaluated TEMdroid on 34 real-world apps from eight categories. Our experimental results demonstrate that using a learning-based matching model and incorporating contextual information are effective in event matching and test case migration. TEMdroid also outperforms baseline approaches in both two tasks.

ACKNOWLEDGEMENTS

We thank the anonymous ICSE reviewers for their valuable feedback and the insightful comments provided by Zhengwei Tao, Chen Liu, and Zhiyong Zhou on the case study. Lu Zhang was partially supported by National Natural Science Foundation of China under Grant No.62232003. Dan Hao was partially supported by National Natural Science Foundation of China under Grant No.62372005. Tao Xie was partially supported by National Natural Science Foundation of China under Grant No.62161146003, and the Tencent Foundation/XPLORER PRIZE.

REFERENCES

[1] 2022. *ChatGPT*. https://chat.openai.com/
[2] 2023. *ATM dataset*. https://sites.google.com/view/apptestmigrator
[3] 2023. *BERT base uncased*. https://huggingface.co/bert-base-uncased
[4] 2023. *Craftdroid dataset*. https://github.com/seal-hub/CraftDroid
[5] 2023. *FrUITeR dataset*. https://felicitia.github.io/FrUITeR
[6] 2023. *Google Play store*. https://play.google.com/store/games
[7] 2023. *GPT-4, a large multimodal model*. https://openai.com/research/gpt-4
[8] 2023. *PyTorch: from research to production*. https://pytorch.org/
[9] 2023. *SemFinder dataset*. https://doi.org/10.5281/zenodo.4725222
[10] 2023. *Source code and extra materials for TEMDroid*. https://github.com/YakZhang/TEMdroid
[11] 2023. *Tesseract OCR: an optical character recognition engine*. https://github.com/tesseract-ocr
[12] 2023. *UI/application exerciser Monkey*. https://developer.Android.com/studio/test/monkey
[13] Alan Akbik, Duncan Blythe, and Roland Vollgraf. 2018. Contextual string embeddings for sequence labeling. In *ACL*. 1638–1649.
[14] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2014. MobiGUITAR: automated model-based testing of mobile apps. *IEEE Software* 32, 5 (2014), 53–59.
[15] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *FSE*. 1–11.
[16] Issa Annamoradnejad and Gohar Zoghi. 2020. ColBERT: using BERT sentence embedding for humor detection. *arXiv preprint arXiv:2004.12765* (2020).
[17] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *ASE*. 238–249.
[18] Yude Bai, Sen Chen, Zhenchang Xing, and Xiaohong Li. 2023. ArgusDroid: detecting Android malware variants by mining permission-API knowledge graph. *SCIS* 66, 9 (2023), 1–19.
[19] Farnaz Behrang and Alessandro Orso. 2019. Test migration between mobile apps with similar functionality. In *ASE*. 54–65.
[20] Luca Bertinetto, Jack Valmadre, Joao F Henriques, Andrea Vedaldi, and Philip HS Torr. 2016. Fully-convolutional Siamese networks for object tracking. In *ECCV*. 850–865.
[21] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Vol. 4. Springer.
[22] Eeshita Biswas, Mehmet Efruz Karabulut, Lori Pollock, and K Vijay-Shanker. 2020. Achieving reliable sentiment analysis in the software engineering domain using BERT. In *ICSME*. 162–173.
[23] Nataniel P Borges Jr, Maria Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *MOBILESoft*. 133–143.
[24] Asli Celikyilmaz, Marcus Thint, and Zhiheng Huang. 2009. A graph-based semi-supervised learning for question-answering. In *ACL*. 719–727.
[25] Sumit Chopra, Raia Hadsell, and Yann LeCun. 2005. Learning a similarity metric discriminatively, with application to face verification. In *CVPR*. 539–546.
[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional Transformers for language understanding. In *NAACL*. 4171–4186.
[27] Felix Dobslaw, Robert Feldt, David Michaëlsson, Patrik Haar, Francisco Gomes de Oliveira Neto, and Richard Torkar. 2019. Estimating return on investment for GUI test automation frameworks. In *ISSRE*. 271–282.
[28] Xingping Dong and Jianbing Shen. 2018. Triplet loss in Siamese network for object tracking. In *ECCV*. 459–474.
[29] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *ASE*. 419–429.
[30] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lü. 2017. AimDroid: activity-insulated multi-level automated testing for Android applications. In *ICSME*. 103–114.
[31] Jiafeng Guo, Yixing Fan, Qingyao Ai, and W Bruce Croft. 2016. A deep relevance matching model for ad-hoc retrieval. In *CIKM*. 55–64.
[32] Walid Hariri. 2023. Unlocking the potential of ChatGPT: a comprehensive exploration of its applications, advantages, limitations, and future directions in Natural Language Processing. *arXiv preprint arXiv:2304.02017* (2023).
[33] Anfeng He, Chong Luo, Xinmei Tian, and Wenjun Zeng. 2018. A twofold Siamese network for real-time object tracking. In *CVPR*. 4834–4843.
[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR*. 770–778.
[35] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *ESEC/FSE*. 269–282.
[36] Haruna Isotani, Hironori Washizaki, Yoshiaki Fukazawa, Tsutomu Nomoto, Saori Ouji, and Shinobu Saito. 2021. Duplicate bug report detection by using sentence embedding and fine-tuning. In *ICSME*. 535–544.
[37] Bekir Karlik and A Vehbi Olgac. 2011. Performance analysis of various activation functions in generalized MLP architectures of neural networks. *IJAE* 1, 4 (2011), 111–122.
[38] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of Android applications. In *ICST*. 105–115.
[39] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. 2015. From word embeddings to document distances. In *ICML*. 957–966.
[40] Duling Lai and Julia Rubin. 2019. Goal-driven exploration for Android applications. In *ASE*. 115–127.
[41] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. doklady* 10, 8 (1966), 707–710.
[42] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: a deep learning-based approach to automated black-box Android app testing. In *ASE*. 1070–1073.
[43] Jiayuan Liang, Sinan Wang, Xiangbo Deng, and Yepang Liu. 2023. RIDA: cross-app record and replay for Android. In *ICST*. 246–257.
[44] Jing Liao, Yikun Huang, Haolin Wang, and Mengting Li. 2021. Matching ontologies with Word2Vec model based on cosine similarity. In *AICV*. 367–374.
[45] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. 2021. Traceability transformed: generating more accurate links with pre-trained BERT models. In *ICSE*. 324–335.
[46] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test transfer across mobile apps through semantic mapping. In *ASE*. 42–53.
[47] Jun-Wei Lin and Sam Malek. 2022. GUI test transfer from Web to Android. In *ICST*. 1–11.
[48] Kuo-Sui Lin and Chih-Chung Chiu. 2015. A fuzzy similarity matching model for interior design drawing recommendation. In *ASE BD&SI*. 1–6.
[49] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test Android applications?. In *ICSME*. 613–622.
[50] Shuqi Liu, Yu Zhou, Tingting Han, and Taolue Chen. 2023. Test reuse based on adaptive semantic matching across Android mobile applications. *arXiv preprint arXiv:2301.00530* (2023).
[51] Yi Liu, Yun Ma, Xusheng Xiao, Tao Xie, and Xuanzhe Liu. 2023. LegoDroid: flexible Android app decomposition and instant installation. *SCIS* 66, 4 (2023), 142103.
[52] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: a robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).
[53] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *ESEC/FSE*. 224–234.
[54] Saket Maheshwary and Hemant Misra. 2018. Matching resumes to jobs via deep Siamese network. In *WWW*. 87–88.
[55] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *FSE*. 599–609.
[56] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *ISSTA*. 94–105.
[57] Qun Mao, Weiwei Wang, Feng You, Ruilian Zhao, and Zheng Li. 2022. User behavior pattern mining and reuse across similar Android apps. *JSS* 183 (2022), 111085.
[58] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic matching of GUI events for test reuse: are we there yet?. In *ISSTA*. 177–190.
[59] Leonardo Mariani, Mauro Pezzè, Valerio Terragni, and Daniele Zuddas. 2023. An evolutionary approach to adapt tests across mobile apps. In *AST*. 70–79.
[60] Chandler May, Alex Wang, Shikha Bordia, Samuel R. Bowman, and Rachel Rudinger. 2019. On measuring social biases in sentence encoders. In *NAACL*. 622–628.
[61] Iaroslav Melekhov, Juho Kannala, and Esa Rahtu. 2016. Siamese network features for image matching. In *ICPR*. 378–383.
[62] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NIPS*. 3111–3119.
[63] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted Boltzmann machines. In *ICML*. 807–814.
[64] Ekaterina Nepovinnykh, Tuomas Eerola, and Heikki Kalviainen. 2020. Siamese network based pelage pattern matching for ringed seal re-identification. In *WACVW*. 25–34.
[65] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *ISSTA*. 153–164.
[66] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. TestMig: migrating GUI test cases from iOS to Android. In *ISSTA*. 284–295.
[67] Dezhi Ran, Zongyang Li, Chenxu Liu, Wenyu Wang, Weizhi Meng, Xionglin Wu, Hui Jin, Jing Cui, Xing Tang, and Tao Xie. 2022. Automated visual testing for mobile apps in an industrial setting. In *ICSE-SEIP*. 55–64.
[68] Dezhi Ran, Hao Wang, Wenyu Wang, and Tao Xie. 2023. Badge: prioritizing UI events with hierarchical multi-armed bandits for automated UI testing. In *ICSE*. 894–905.
[69] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. 2018. Transferring tests across Web applications. In *ICWE*. 50–64.

[70] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).
[71] Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. 2016. Training region-based object detectors with online hard example mining. In *CVPR*. 761–769.
[72] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *ISSTA*. 12–22.
[73] Jianlin Su, Jiarun Cao, Weijie Liu, and Yangyiwen Ou. 2021. Whitening sentence representations for better semantics and faster retrieval. *arXiv preprint arXiv:2103.15316* (2021).
[74] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *ESEC/FSE*. 245–256.
[75] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. MobileBERT: a compact task-agnostic BERT for resource-limited devices. *arXiv preprint arXiv:2004.02984* (2020).
[76] K-K Sung and Tomaso Poggio. 1998. Example-based learning for view-based human face detection. *TPAMI* 20, 1 (1998), 39–51.
[77] Saghar Talebipour, Yixue Zhao, Luka Dojcilović, Chenggang Li, and Nenad Medvidović. 2021. UI test migration across mobile platforms. In *ASE*. 756–767.
[78] Najam us Saqib and Sara Shahzad. 2018. Functionality, performance, and compatibility testing: a model based approach. In *FIT*. 170–175.
[79] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*. 6000–6010.
[80] Xiaolong Wang, Abhinav Shrivastava, and Abhinav Gupta. 2017. A-Fast-RCNN: hard positive generation via qdversary for object detection. In *CVPR*. 2606–2615.
[81] Junfang Wu, Chunyang Ye, and Hui Zhou. 2021. BERT for sentiment classification in software engineering. In *ICSS*. 115–121.
[82] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *FASE*. 250–265.
[83] Shengcheng Yu, Chunrong Fang, Yexiao Yun, and Yang Feng. 2021. Layout and image recognition driving cross-platform automated mobile testing. In *ICSE*. 1561–1571.
[84] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for Android: are we really there yet in an industrial case?. In *FSE*. 987–992.
[85] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. FrUITeR: a framework for evaluating UI test reuse. In *ESEC/FSE*. 1190–1201.
[86] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated test input generation for Android: towards getting there in an industrial case. In *ICSE-SEIP*. 253–262.
[87] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. 2019. Wuji: automatic online combat game testing using evolutionary deep reinforcement learning. In *ASE*. 772–784.
[88] Tianyang Zhong, Yaonai Wei, Li Yang, Zihao Wu, Zhengliang Liu, Xiaozheng Wei, Wenjun Li, Junjie Yao, Chong Ma, Xiang Li, et al. 2023. ChatABL: abductive learning via natural language interaction with ChatGPT. *arXiv preprint arXiv:2304.11107* (2023).