Semi-supervised Crowdsourced Test Report Clustering via Screenshot-Text Binding Rules

SHENGCHENG YU, Nanjing University, China
CHUNRONG FANG, Nanjing University, China
QUANJUN ZHANG, Nanjing University, China
MINGZHE DU, Nanjing University, China
JIA LIU, Nanjing University, China
ZHENYU CHEN, Nanjing University, China

Due to the openness of the crowdsourced testing paradigm, crowdworkers submit massive spotty duplicate test reports, which hinders developers from effectively reviewing the reports and detecting bugs. Test report clustering is widely used to alleviate this problem and improve the effectiveness of crowdsourced testing. Existing clustering methods basically rely on the analysis of textual descriptions. A few methods are independently supplemented by analyzing screenshots in test reports as pixel sets, leaving out the semantics of app screenshots from the widget perspective. Further, ignoring the semantic relationships between screenshots and textual descriptions may lead to the imprecise analysis of test reports, which in turn negatively affects the clustering effectiveness.

This paper proposes a semi-supervised crowdsourced test report clustering approach, namely SemCluster. SemCluster respectively extracts features from app screenshots and textual descriptions and forms the structure feature, the content feature, the bug feature, and reproduction steps. The clustering is principally conducted on the basis of the four features. Further, in order to avoid bias of specific individual features, SemCluster exploits the semantic relationships between app screenshots and textual descriptions to form the semantic binding rules as guidance for clustering crowdsourced test reports. Experiment results show that SemCluster outperforms state-of-the-art approaches on six widely used metrics by 10.49% – 200.67%, illustrating the excellent effectiveness.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Semi-Supervised Clustering, Crowdsourced Test Report Clustering, Image Understanding, Text Analysis

ACM Reference Format:
Shengcheng Yu, Chunrong Fang, Quanjun Zhang, Mingzhe Du, Jia Liu, and Zhenyu Chen. 2024. Semi-supervised Crowdsourced Test Report Clustering via Screenshot-Text Binding Rules. *Proc. ACM Softw. Eng.* 1, FSE, Article 69 (July 2024), 24 pages. https://doi.org/10.1145/3660776

Authors' addresses: Shengcheng Yu, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, China, yusc@smail.nju.edu.cn; Chunrong Fang, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, China, fangchunrong@nju.edu.cn; Quanjun Zhang, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, China, quanjun.zhang@smail.nju.edu.cn; Mingzhe Du, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, China, 181250028@smail.nju.edu.cn; Jia Liu, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, China, liujia@nju.edu.cn; Zhenyu Chen, Nanjing University, State Key Laboratory for Novel Software Technology, Nanjing, China, zychen@nju.edu.cn.

ACM 2994-970X/2024/7-ART69
https://doi.org/10.1145/3660776

1 INTRODUCTION

Crowdsourced testing has been playing a dominant role in software testing, especially in the mobile application (app) domain [19]. Due to the fragmentation problem faced by mobile app testing [64], in-house testing technologies have to deal with the compatibility problems [19]. However, it is challenging for app developers to test apps in all kinds of environments. Crowdsourced testing makes up such a shortcoming with its openness [19]. In crowdsourced testing, testing tasks are distributed to a large number of crowdworkers, utilizing their different testing environments or devices. Then the app developers can collect reports from distinct testing environments.

However, the openness of crowdsourced testing also brings risks to app testing. One particular problem is duplicate reports. As presented in [55], there are 82% duplicate test reports. Also, from our investigation, the duplication ratio reaches 87.13% (see details in Table 1). Most reports report similar bugs, hindering app developers from effectively reviewing reports and handling bugs.

Clustering is widely used for handling test report duplication in an unsupervised way and based on extracted features from crowdsourced test reports [30]. It helps app developers focus on a cluster of reports describing the same bug at a time with the information in the reports supplementing each other. Most existing work [4, 5, 21, 24, 26, 41, 51] focuses on textual information of test reports to cluster duplicate reports. Currently, crowdsourced test reports consist of app screenshots and textual descriptions. App screenshots are a strong carrier of bug information. Several state-of-the-art studies [17, 18, 55] preliminary consider the combination of app screenshots and textual information. However, such approaches take app screenshots as sets of pixels, completely ignoring contained semantic information in app screenshots or textual descriptions [64]. Moreover, app screenshots and textual descriptions cooperate with each other to better present the bugs. Therefore, semantic relationships between app screenshots and textual descriptions can be utilized. However, none of the existing work considers such relationships to assist test report clustering, which leads to a great waste of information contained in the crowdsourced test reports.

In this paper, we propose **SemCluster**, a **Sem**i-supervised crowdsourced test report **Cluster**ing approach via screenshot-text **Sem**antic binding rules. SemCluster extracts features with semantics from app screenshots and textual descriptions to represent and characterize crowdsourced test reports. SemCluster further utilizes semantic relationships between app screenshots and textual descriptions to form semantic binding rules to correct the biases and guide the clustering process.

In particular, SemCluster first extracts features from both app screenshots and textual descriptions to represent crowdsourced test reports. For app screenshots, SemCluster extracts existing widgets and characterizes the app page structures. *Content Feature* and *Structure Feature* are used to represent the app screenshots. *Content Feature* is a micro perspective of the app screenshots, describing the contents of the app screenshots, including images, texts, *etc.* From the macro perspective, SemCluster presents the *Structure Feature*, which describes the general structure of the app screenshots and neglects the specific content details. Textual descriptions are another significant part of crowdsourced test reports. Crowdworkers describe the bugs intuitively. Textual descriptions are divided into *Bug Behavior* and *Reproduction Step*. *Bug Behavior* directly describes the app behaviors and may include expectations from the crowdworkers. *Bug Behavior* is a direct presentation from the crowdworkers. *Reproduction Step* presents the test events triggering the bugs and constructs a context of the bug.

With the extracted features, SemCluster starts the exploration of semantic relationships between app screenshots and textual descriptions. App screenshots and textual descriptions elaborate on bugs from different perspectives, and they corroborate each other. Based on different features with semantics, SemCluster mines the semantic relationships between app screenshots and textual descriptions. As a representation of the semantic relationships, we form the semantic binding rules.

Essentially, the semantic binding rules focus on specific features from app screenshots and textual descriptions if any two test reports are strongly linked to such features, and other features are neglected to eliminate the possible negative effect. Such rules include *MUST–LINK* and *CANNOT–LINK*, which show the strong links or strong antinomies of crowdsourced test reports, respectively. *MUST–LINK* and *CANNOT–LINK* respectively indicate that two reports are supposed to be assigned to the same cluster or to different clusters (Section 3.3).

Based on the semantic features and guided by the semantic binding rules, SemCluster adopts a semi-supervised method for report clustering. The specific clustering algorithm is based on the K-Medoids clustering [1, 44]. The K-Medoids algorithm keeps actual test reports as centers (medoids) during the clustering iterations. During the clustering, SemCluster calculates the distance of different features with adaptive strategies between two crowdsourced test reports for clustering.

To evaluate the proposed SemCluster, we construct a large-scale crowdsourced test report dataset, which contains 847 reports of 18 different mobile apps of various domains. Based on the dataset, we conduct an empirical evaluation. SemCluster achieves promising results over six widely used metrics, including Precision, Recall, F-measure, Purity, ARI [23, 48], and NMI [53], outperforming state-of-the-art baselines by 10.49% – 200.67%. Besides, the ablation study demonstrates that semantic feature extraction and semantic binding rule guidance both positively contribute to SemCluster.

The noteworthy contributions of this paper can be concluded as the following points:

- We propose a novel and effective mechanism for characterization and representation features for app screenshots and textual descriptions in crowdsourced test reports with intermediate semantic relationships.
- We introduce semantic binding rules to guide the semi-supervised clustering algorithm.
- We propose a novel approach for clustering crowdsourced test reports with image and text semantics, and we implement and open-source a tool for practice.
- We conduct an empirical study to evaluate the effectiveness of the proposed SemCluster.

Reproduction package is available on: https://sites.google.com/view/semcluster.

2 BACKGROUND & MOTIVATION

In this section, we first introduce the background of mobile app crowdsourced testing, and then we introduce two concrete examples from our real-world dataset to illustrate our motivation.

2.1 Mobile App Crowdsourced Testing

As a common workflow of crowdsourced testing, requesters publish test tasks on crowdsourced testing platforms, and then crowdworkers execute such test tasks and submit test reports to the platform [19]. Most mainstream crowdsourced testing platforms process test reports and give feedback to requesters [63].

On most of the mainstream crowdsourced testing platforms, crowdworkers are required to submit reports to describe the bugs occurring during the testing [19]. Each crowdsourced test report is supposed to consist of one app screenshot and textual descriptions [17, 18, 63]. Textual descriptions include reproduction steps that indicate the test events triggering the bugs and bug descriptions focusing on the bugs. App screenshots and textual descriptions assist app developers in inspecting the reports mutually.

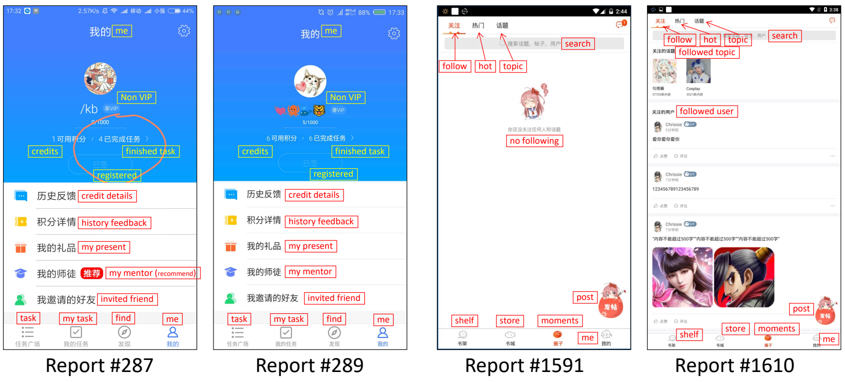
The most obvious advantage of crowdsourced testing is that it utilizes various testing environments of different crowdworkers [55, 58, 59]. However, the critical quality control challenge is also due to the openness [63]. Requesters need optimized test reports, and clustering is one of the most important strategies [30]. Report clustering groups similar test reports together based on common attributes or characteristics, and it improves the efficiency and accuracy of test report processing, allowing the identification of common patterns of bugs across multiple reports and simplifying the analysis. Existing approaches ignore the semantics contained in the crowdsourced test reports, which can assist the report characterization and representation. Further, the semantic relationship between app screenshots and textual descriptions is wasted.

2.2 Motivation

To better illustrate our motivation, we provide two typical examples.

In the first motivating example (Fig. 1(a)), the app screenshots are highly similar, while the textual descriptions are quite different. In Report #287, the crowdworker is reporting the differences between actual app activity and expectations. In Report #289, the crowdworker is reporting that the expected "unfinished task" is replaced with "finished task". The root cause of the Report #287 is confirmed as not being synchronized with the database, and the root cause of the Report #289 is confirmed as using wrong field on the web end. These two reports are reporting completely different bugs. However, they use almost the same app screenshots. For traditional approaches, the clustering results will be affected by the app screenshots and the reports may be mistakenly clustered into the same cluster. However, with our semantic binding rules, the large SemDistance satisfies the *CANNOT–LINK* rule, thus directly guiding the clustering algorithm to put these two reports into different clusters despite the high similarity of app screenshots.

Report #287: Click on the "me" button, the actual "finished task" number should be 0 instead of 4. The result is different from expectation.
Report #289: Click to accept the task, and choose to view unfinished tasks, while it shows finished tasks under the avatar.



(a) Different Textual Descriptions for Similar App Screenshots (b) Different App Screenshots for Similar Textual Descriptions

Fig. 1. Motivating Examples (Translations for important phrases are provided in red or yellow boxes.)

In the second motivating example (Fig. 1(b)), the app screenshots are quite different due to the contents of the posts, while the two reports actually report the same problem: extra refresh operation is required after following topics or users. However, similar to the first motivating example, traditional approaches may make mistakes if all features are considered. Therefore, based on the designed semantic binding rules, specifically the *MUST–LINK*, we can eliminate the influence of unimportant features and focus on "important" features.

Report #1591: Choose one topic to follow, contents will not appear until the app is refreshed.
Report #1610: Choose moments and follow two topics, while the app does not show the content, contents can be seen after refreshing.

As reflected in the above typical examples, we can find that in some cases, some of the features may have negative side effects on the report clustering. Unsupervised approaches are incapable of dealing with such situations robustly. Therefore, in this paper, we are trying to improve the clustering effectiveness by building a set of semantic binding rules to cluster crowdsourced test reports in a semi-supervised way.

3 APPROACH

In this section, we present our proposed approach, SemCluster, in detail. SemCluster is composed of three stages: feature extraction, report distance calculation, and report clustering (as shown in Fig. 2). Firstly, four features are extracted to represent the crowdsourced test reports consisting of app screenshots and textual descriptions. The adopted features include *Structure Feature*, *Content Feature*, *Bug Behavior*, and *Reproduction Step*. Secondly, with these features, different distance algorithms are used to calculate the SemDistance respectively, and we utilize a weighted algorithm to merge the distance as the SemDistance between each two crowdsourced test reports. Thirdly, during the clustering process guided by the semantic binding rules, the specific clustering algorithm is the K-Medoids algorithm, in order to prevent the centers from deviating from actual data points. During each iteration of clustering, the designed semantic binding rules will take effect to correct potential biases in the clustering and guide the clustering process.

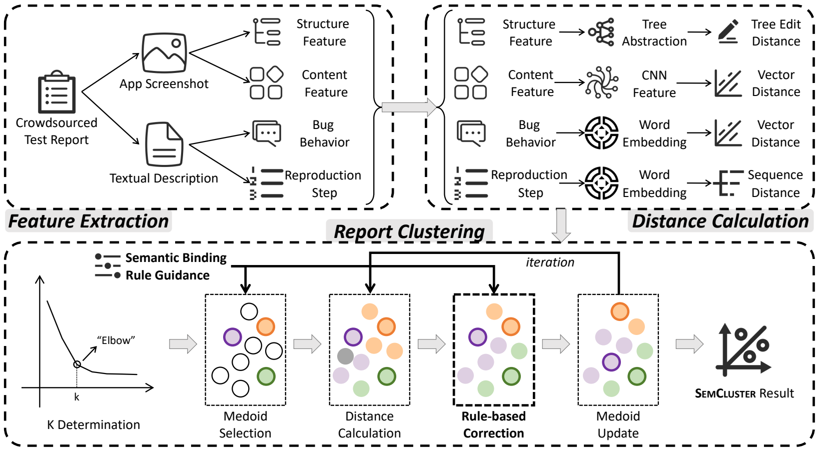


Fig. 2. SemCluster Framework

3.1 Feature Extraction

As the first step towards effectively clustering the crowdsourced test reports, representing the reports with features precisely is significant. In this section, we illustrate the feature design. Crowdsourced test reports consist of app screenshots and textual descriptions. For app screenshots, SemCluster considers the *Structure Feature* and the *Content Feature*. For textual descriptions, SemCluster considers the *Bug Behavior* and the *Reproduction Step*.

Textual Description Preprocessing. The *Structure Feature* and the *Content Feature* are directly extracted from app screenshots, while *Bug Behavior* and *Reproduction Step* are always mixed in the crowdsourced test reports. One critical prerequisite for extracting the features is to distinguish *Bug Behavior* from *Reproduction Step*. Therefore, we use a TextCNN model [27] to classify each sentence. Based on the pre-trained models, we utilize a corpus from [63] to make the fine-tuning. The corpus contains about 4k reports collected from Github, StackOverflow, and other resources, and is divided into the training set, validation set, and test set at the ratio of 6:2:2, as the common practice [63]. The TextCNN model first embeds the words into 128-dimension vectors with a Word2Vec model [34]. Afterward, the `Convolutional` layers and `MaxPooling` layers are connected to extract the text features. Finally, the `SoftMax` layer is used to predict whether the sentence is a *Bug Behavior* or a *Reproduction Step*. The overall *Precision* and *Recall* values reach 98%[1] on the test set.

3.1.1 Structure Feature. *Structure Feature* describes the GUI structure of the app screenshots. As many apps are refreshing their content constantly, *i.e.,* news apps and shopping apps, this may make it challenging to characterize their content accurately. In this situation, there is something else that is important to consider to better characterize the app GUI, which is the overall structure of the app screenshots. Also, in some cases, the app screenshots may be distinct from each other when the crowdworkers report the same bugs. However, if the reported bugs are within the same app activity, the GUI structure will be the same. Therefore, we consider the *Structure Feature* as a significant feature. In the crowdsourced testing scenario, we have to directly extract the GUI layout from the GUI images. Therefore, we combine traditional computer vision (CV) algorithms and deep learning (DL) technologies to process app screenshots.

To extract the GUI structure directly from app screenshots, we first extract the existing app widgets from the app screenshots with CV algorithms. Specifically, utilizing the Canny edge detection method [8] in conjunction with Optical Character Recognition (OCR) techniques, SemCluster can successfully segment app screenshots based on GUI widgets. The Canny algorithm identifies the edges of GUI widgets by first converting the images to grayscale. Through this process, the detected edges are obtained and transformed into bounding boxes, each defined by four coordinates, thereby outlining rectangular areas corresponding to the widgets. To further refine the segmentation, we integrate the outcomes from OCR and Canny, eliminating any overlapping bounding boxes. However, the extracted areas from app screenshots may be blank instead of actual widgets. Therefore, we further use a convolutional neural network (CNN) model [47, 63], VGG-16 (detailed in Section 3.1.2), to identify the widget category of each candidate widget. As a result, the blank areas are eliminated because they may be only image resources shown on the app GUI.

After acquiring all the valid widgets from the app screenshots, which refer to the extracted areas being actual widgets, we organize them in a four-level tree structure. The root node of the tree structure represents the app screenshot (the first figure in Fig. 3), *i.e.,* all the widgets. In level one, we name the nodes as **`Group`** (the second figure in Fig. 3), which is a rough horizontal operation on the app screenshot. Situations are common when some GUI widgets overlap other widgets in height (like cells occupying many rows in tables). In this level, such widgets should be in the same **`Group`**, and within a **`Group`**, some of the widgets may have different coordinates in height. In level two, we name the nodes as **`Line`** (the third figure in Fig. 3), which is a subtle horizontal operation on the app screenshot on the basis of the **`Group`**. Within a **`Group`**, we take each GUI widget as the smallest unit. Therefore, one **`Group`** may be divided into one or more **`Line`**s. Moreover, the widgets that overlap other widgets in height will be segmented into different lines. In level three, we name the nodes as **`Column`** (the fourth figure in Fig. 3), which is a vertical operation on the app screenshot. **`Column`** is done according to the vertical coordinates of the GUI widgets. An app screenshot is therefore represented by a GUI tree, and the leaf nodes represent GUI widgets. All the trees are of the same height. Such a design is because app developers tend to group together widgets with

[1]Detailed process and result data are omitted due to the page limit and can be found in our online package.

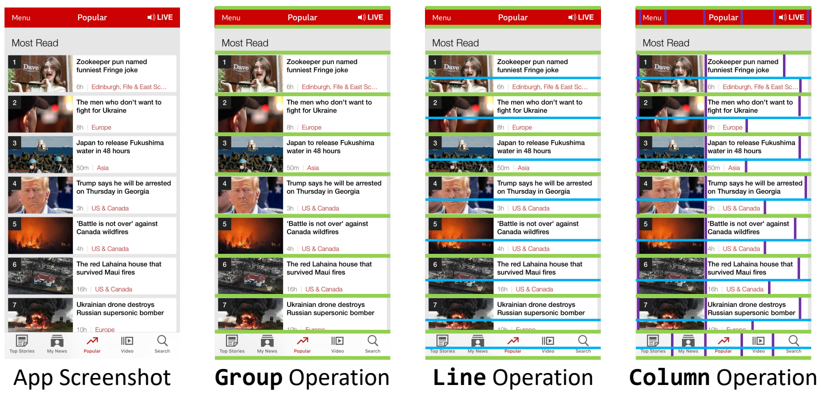


Fig. 3. Illustration Example of Layout Characterization

similar semantics and keep those with different semantics apart [43, 45]. The reason for two-level horizontal operations is that some widgets may overlap other widgets on different lines in height.

During the formation of the GUI structure tree, some particular processing steps are necessary. Some widgets are with the same coordinates, which means they should be in the same **`Line`** or **`Column`**. However, there may actually exist slightly different coordinates horizontally or vertically for such widgets due to the subtle bias of the widget recognition algorithm. We set a threshold, ϕ, to identify whether the widgets should be in the same **`Line`**. ϕ is a ratio of the app screenshots' height. If one **`Line`**'s height is smaller than ϕ, the **`Line`** will be merged to its upper **`Line`**. In our implementation, we set the ϕ as 0.05 as the literature [64]. The processing is similar to the **`Column`**. If the ratio between the width of recognized areas and the width of the app screenshot is smaller than ϕ, the area will be considered as mistakenly recognized GUI widgets. Mistakenly recognized widgets caused by the image texture of the contents are also eliminated to prevent needless overhead.

3.1.2 Content Feature. *Content Feature* depicts GUI widget contents, *e.g.,* images, and buttons of the app screenshots. Such "contents" can have a significant effect on calculating the SemDistance between every two reports. Different from the *Structure Feature*, which views app screenshots from a macro perspective, *Content Feature* concentrates on the micro perspective of the app screenshots. *Content Feature* pays attention to the details of the whole app screenshots and considers the situation that content refreshing can bring changes even if the overall structure remains unchanged.

Since the app screenshots have been characterized to construct the *Structure Feature*, we take all leaf nodes of the app screenshot structure, *i.e.,* the widget screenshots, to represent the *Content Feature* of the app screenshots. With the post-processing of the app screenshots in Section 3.1.1, SemCluster removes most invalid widgets (mistakenly recognized blank areas) and leaves valid ones, by removing the recognized areas whose width or height is smaller than specific ratios of the app screenshots. Such practice eliminates negative effects of the trivial parts of some widgets with complicated textures and can reduce the distance calculation overhead by reducing widget number.

To extract and represent the widget screenshots, we use a CNN model [47, 63] aiming at a classification task. For the training of the model, we use a dataset containing over 36k app widget screenshots from [63], evenly distributed in 14 widget types[2], *e.g.,* `Button`, `TextField`, `ImageView`, `Switch`, `CheckBox`, `RadioButton`, *etc.* Following the existing research [63], the dataset is split at the

[2]Common ones in https://developer.android.com/reference/android/widget/package-summary

ratio of 7:2:1 to form the training set, the validation set and the test set. The neural network is composed of `Convolutional` layers, `MaxPooling` layers, and `FullyConnected` layers. The `AdaDelta` algorithm is adopted as the optimizer and the `categorical_crossentropy` function as the loss function. The batch size is set as 32. These hyperparameters are kept the same as the original VGG-16 model [47]. The overall *Precision* and *Recall* values reach 90%, which means that the trained model can effectively extract widget screenshot features[3], especially in identifying features of widget contents.

After acquiring the model and the fixed parameters, we discard the last layer of the model, (*i.e.,* the `SoftMax` layer), and directly output the results of the last but one layer, the `FullyConnected` layer as a 4096-dimension vector [25, 29]. The vectors are used to represent the widget screenshots. Each app screenshot contains several widgets, and the quantity is different, so the length of representing vectors for the app screenshots is also variable. Therefore, *Content Feature* are variable-length sets of 4096-dimension vectors, representing the widgets within the app screenshots.

3.1.3 Bug Behavior. Bug Behavior directly depicts the bug situations and is often attached to the expectations of the crowdworkers. *Bug Behavior* is in the form of short sentences. Most *Bug Behavior* loosely follows a pattern to describe the app behaviors and user expectations: "apply ⟨*operation*⟩ on ⟨*widget*⟩, ⟨*behavior*⟩ happens, while ⟨*expectation*⟩ is expected". Therefore, as widely used in previous studies [55, 64], we adopt the Word2Vec model [34] to directly encode the *Bug Behavior*. The *Bug Behavior* in each crowdsourced test report is therefore transformed into a 100-dimension vector, as the practice in existing research [63]. In the scenario of crowdsourced test report processing, *Bug Behavior* is very short sentences, usually no more than 50 words. Therefore, Word2Vec is capable enough to process texts of such a length. Moreover, Word2Vec is lightweight and does not require extra training or fine-tuning, unlike some more advanced methods, *e.g.,* Bert. Third, Word2Vec is also with a lower overhead than other more advanced models and is more beneficial when applied to a practical scenario. To make the Word2Vec model more adaptive for mobile app crowdsourced testing, we use the domain-specific keyword list [63] to identify the synonyms, antonyms, and polysemies in mobile app testing.

3.1.4 Reproduction Step. Reproduction Step states the user operation sequence from the app launch to the bug occurrence. In mobile app testing, test operations are important, and different operation sequences will make app behaviors different even on the same app activity. In each operation, two parts are important, the concrete operation (op_i) and the object widget (obj_i) of the operation. The *Reproduction Step* can be abstracted into a formal sequence: $RP^t = \langle S^t_1(op_1, obj_1), S^t_2(op_2, obj_2), ..., S^t_n(op_n, obj_n)\rangle$.

To extract the operation and the object from the textual descriptions, we adopt the constituent parser algorithm [66] to analyze the part of speech for the text segments. Up to now, the informal *Reproduction Step* is formalized into an operation sequence consisting of operations and objects.

3.2 Report Distance Calculation

To cluster the crowdsourced test reports, we calculate the distance, SemDistance, between every two reports. However, the four features are in different forms, *i.e., Structure Feature* is in the form of trees, *Content Feature* is in the form of a set of image vectors, *Bug Behavior* is in the form of textual vectors, and *Reproduction Step* are semi-structured sequences, so different strategies are used.

3.2.1 Structure Feature. Structure Feature is in the form of tree structures. Therefore, we adopt the advanced tree edit distance algorithm, APTED (All Path Tree Edit Distance) [39, 40] to calculate the SemDistance on *Structure Feature*. The tree edit distance refers to the minimal operations required

[3]Detailed process and result data are omitted due to the page limit and can be found in our online package.

to transform one tree structure into another. APTED considers three kinds of operations: ① **Delete** one node and put its child nodes to its parent node. ② **Insert** one node between an existing one and its children. ③ **Rename** an existing node.

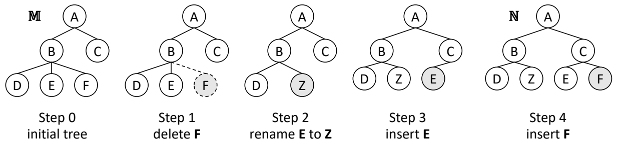


Fig. 4. Tree Edit Distance Example

We give an example in Fig. 4 to illustrate the operations. In order to transform tree $\mathbb{M}$ into tree $\mathbb{N}$, the first step is to delete the **F** node and then rename the **E** node into **Z**. Following, add **E** node and **F** node to **C** node in order. Therefore, the edit distance between tree $\mathbb{M}$ and tree $\mathbb{N}$ is 4.

In the distance calculation of *Structure Feature*, the **Rename** operation is neglected because we assign all nodes the same label. The intuition of this practice is that we only care about the general structure of the app screenshot, and the detailed contents are considered in the *Content Feature*.

In addition, to unify the distance data distribution, we normalize the distance to the range of [0, 1] with Min-Max-Normalization.

3.2.2 Content Feature. *Content Feature* is in the form of image vectors. Different from traditional practice, which directly calculates the distance of the whole app screenshots with the pixel granularity, we consider the widget granularity (Algorithm 1).

For the two app screenshots in two reports, we use the results from Section 3.1.1, which contains the widget number and the coordinate information of each widget. We consider the test report with fewer widgets on the app screenshot as the W_{source}, and the other as the W_{target} (Line 1–7). For each widget w_s in the W_{source}, we find the matching widget w_t^* from the W_{target}. The choice of w_t^* has two criteria: 1) the closest widget in the W_{target}, which is calculated according to the coordinates (Line 11–19); and 2) the IoU (Intersection over Union) value should be greater than a preset threshold λ (Line 20–23). In our implementation, we set the λ as 0.75 according to the experience. If the matching widget w_t^* is found, we calculate the Euclidean distance between the two widget vectors from the CNN model. The average distance of all the matching widget pairs is considered as the distance of the *Content Feature*. The range of the *Content Feature* distance is normalized to [0, 1] with the Min-Max-Normalization: $\frac{x-min}{max-min}$, where *max* is the maximum value of all results and *min* is the minimum value of all results.

3.2.3 Bug Behavior. *Bug Behavior* is in the form of text vectors. Following the common practice [30, 64], we use the Euclidean distance to the *Bug Behavior* distance in pair. Also, in order to unify values of different scales, we normalize each result to [0, 1] with the Min-Max-Normalization.

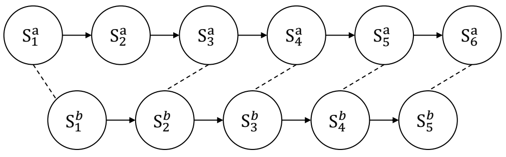


Fig. 5. Distance Calculation of Reproduction Step

3.2.4 Reproduction Step. *Reproduction Step* is in the form of operation node sequences. Therefore, in order to consider the semantics into the distance calculation, we do not directly transform the

Algorithm 1: Distance Calculation of Content Feature

```
Input: App Screenshot $S_1$, App Screenshot $S_2$;
Output: similarity $sim_{ct}$
1  $W_1 \leftarrow$ extractWidget($S_1$)    // Widget Set 1
2  $W_2 \leftarrow$ extractWidget($S_2$)    // Widget Set 2
3  if $W_1.size() \leq W_2.size()$ then
4      $W_{source} \leftarrow W_1$;    $W_{target} \leftarrow W_2$
5  else
6      $W_{source} \leftarrow W_2$;    $W_{target} \leftarrow W_1$
7  end
8  disSum $\leftarrow$ 0 // the sum of distance between all widget pairs
9  widgetNum $\leftarrow$ 0
10 for each widget $w_s \in W_{source}$ do
11     minDis $\leftarrow \infty$
12     $w_t^* \leftarrow$ Null
13     for each widget $w_t \in W_{target}$ do
14         coordinateDis $\leftarrow$ coordinateDis($w_s$, $w_t$)
15         // calculate the distance of 2 widgets' coordinates
16         if coordinateDis < minDis then
17             minDis $\leftarrow$ coordinateDis
18             $w_t^* \leftarrow w_t$
19         end
20     end
21     IoU $\leftarrow$ calIoU($w_s$, $w_t^*$)
22     if $IoU > \lambda$ then
23         disSum += EuclideanDistance($w_s$, $w_t^*$)
24         widgetNum += 1
25     end
26 end
27 return $dis_{ct} \leftarrow$ disSum / widgetNum;
```

texts into vectors. Instead, we consider the *Reproduction Step* as a sequence. With two sequences RP^a and RP^b, each node of the sequence S_i^t is composed of an *operation* and an *object*. For each S_i^a in RP^a and each S_i^b in RP^b, we calculate the text similarity between all the nodes. Then, we adopt the Dynamic Time Warping (DTW) [46] algorithm. The DTW algorithm is originally used for the distance calculation of temporal sequences. In the sequence, the DTW algorithm aims to find the most similar node pairs (Fig. 5), specifically, which means that it will figure out the similar passed app activities during the testing. The distance is also normalized to [0, 1] with the Min-Max-Normalization to unify the scale among different features.

3.2.5 Feature Aggregation. As the common practice of merging different features [18, 55, 63], we use a set of parameters, θ, to declare the weights of the features. The merging formula is illustrated as $SemDistance = (\sum_{i=1}^{4} \theta_i * dis(\mathbb{X}))$. The $\mathbb{X}$ represents for the four features, *Structure Feature*, *Content Feature*, *Bug Behavior* and *Reproduction Step*, and the sum of θ_i should be 1 ($\sum_{i=1}^{4} \theta_i = 1$). In our implementation, we set the weights for four features as 0.25 as a default setting. As the most effective method to merge different features, we also use the weighted sum of different features. However, we further introduce the semantic binding rules to indicate the semantic relationship between app screenshots and textual descriptions. The rules mine and utilize the semantic relationship as a guide to the clustering process, which reflects the "semi-supervised" idea.

3.3 Semantic Binding Rule Construction

The clustering results of traditional approaches greatly depend on the input parameters, *i.e.,* the number of clusters. Moreover, such methods require a large amount of effort on data labeling, which is limited and expensive in crowdsourced test reports, because domain knowledge or expertise is required. Traditional approaches neglect [17, 18, 30] the rich implicated semantics of the app screenshots and textual descriptions. However, such information can be utilized to guide the clustering process.

In this paper, we use two types of constraints to guide the clustering process. The constraints consider both app screenshots and textual descriptions, and the mutual semantic relationships. The two types of rules used to specify the relationship are:

- ***MUST–LINK***: two data instances (reports) are supposed to be assigned to the same cluster.
- ***CANNOT–LINK***: two data instances (reports) are supposed to be assigned to different clusters.

Rules can be extended. If Report A and Report B are *MUST–LINK*, and Report B and Report C are *MUST–LINK*, then Report A and Report C should be *MUST–LINK*; if Report A and Report B are *MUST–LINK*, and Report B and Report C is *CANNOT–LINK*, then Report A and Report C should be *CANNOT–LINK*, *etc.*

The screenshot-text semantic binding rules are used to implement the constraints, and can be concluded in Equation 1 (SF indicates *Structure Feature*, CF indicates *Content Feature*, BB indicates *Bug Behavior*, and RS indicates *Reproduction Step.*):

$$\textcircled{1}\, Dis(SF) < \alpha_1 \;\&\&\; Dis(BB) < \theta_1 \quad \textcircled{2}\, Dis(SF) < \alpha_2 \;\&\&\; Dis(RS) < \theta_2 \quad (\mathit{MUST–LINK}) \tag{1a}$$

$$\textcircled{1}\, Dis(CF) > \beta_1 \;\&\&\; Dis(RS) > \omega_1 \quad \textcircled{2}\, Dis(CF) > \beta_2 \;\&\&\; Dis(BB) > \omega_2 \quad (\mathit{CANNOT–LINK}) \tag{1b}$$

The $Dis(*)$ refers to the distance between two reports on one specific feature. In our implementation, as default, we set the α_i as 0.1, the θ_i as 0.3, the β_i as 0.7, and the ω_i as 0.8. The parameters are decided according to preliminary small-scale trials and following existing study [13].

We apply two groups of semantic binding rules, the *MUST–LINK* and *CANNOT–LINK*, which are designed to correct the possible bias caused by the coefficient-based similarity calculation. The proposed rules are based on a deep understanding of the crowdsourced test reports. We combine the domain knowledge of human testers on the semantics contained in crowdsourced test reports and the inner semantic relationships to construct the semantic binding rules. We give two direct examples in Section 2.2 to illustrate the necessity and effectiveness of the semantic binding rules. It should be highlighted that the semantic binding rules are designed to confirm that the clustering results can comply with the intuition of human testers without requiring large-scale human efforts to label the crowdsourced test reports.

3.4 Report Clustering

In this work, the specific clustering algorithm we use is the K-Medoids clustering [1, 44]. Compared with the more widely used and more fundamental algorithm, K-Means [32] algorithm, one advantage of the K-Medoids clustering is that it keeps choosing actual reports as centers (medoids) during all the clustering iterations. It keeps the actual reports that actually maintain the semantics. In contrast, the K-Means algorithm will generate new data instances ("reports") as centers. However, the generated centers do not have any semantics, which violates the design of SemCluster. For example, in the "Distance Calculation" step in Fig. 2, the grey circle represents the fake instance of one cluster, which is deviated from actual report instances, completely losing the semantics and making the following clustering meaningless.

Before the clustering process, we first build the semantic binding rule sets. For all report pairs, we query the distance of the four features and apply the semantic binding rules. Finally, we form two semantic binding rule sets, the *MUST–LINK* set, and the *CANNOT–LINK* set, containing all the *MUST–LINK* report pairs and all the *CANNOT–LINK* report pairs, respectively.

The first step of the clustering process is to determine the K value, *i.e.,* the cluster number. We use the *Elbow method* [52], which uses the WSS (within cluster sum of squares) to determine the K. The intuition is that when the increase of K will not bring enough benefits, the K should not be increased. This is one of the most widely used K value determination algorithms.

Then, we select medoids for the clustering. We adopt a guided-stochastic method to determine the medoids. The results of medoid selection should obey the semantic rules, which means that if two instances have *MUST–LINK* relationships, they should not be selected as medoids at the same time. Therefore, we first sample K instances out of all the crowdsourced test reports, then we query the selected instances to the *MUST–LINK* set to check whether there exist instances having *MUST–LINK* relationships. If so, only one of such instances will be left and others will be removed from the initial medoid list, and then new randomly selected instances will be added. The query to the *MUST–LINK* set will be repeated until no instances with *MUST–LINK* relationships exist in the initial medoid list at the same time. The semantic binding rules are expandable, and conflicts may appear, so the design of SemCluster needs to consider the conflict resolving of different *MUST–LINK* rules and *CANNOT–LINK* rules. If one coming instance satisfies different *MUST–LINK* rules with different instances in different clusters, the coming instance will be clustered into the cluster with the instance that has smaller distance with the coming instance. If one coming instance satisfies different *CANNOT–LINK* rules with different instances in different clusters, the coming instance will not be clustered to any of the clusters. If the binding rules conflict, which means two reports satisfy both *MUST–LINK* and *CANNOT–LINK*, SemCluster will discard both rules because the conflict means the reports have quite different similarity situations on different features, so they cannot be simply applied with the semantic binding rules. Therefore, we treat these two reports as common clustering instances with no *MUST–LINK* or *CANNOT–LINK*. This practice can help resolve the potential conflicts with different *MUST–LINK* rules or different *CANNOT–LINK* rules or between *MUST–LINK* and *CANNOT–LINK* rules.

Afterward, the iterative clustering begins. In each iteration, we determine which is the closest medoid of all instances that are not medoids calculated by SemDistance. Then, the semantic binding rule sets are queried to justify any instances that violate the rules.

The semantic binding rule set is used to correct the clustering results. For each target instance, if there exist clusters with instances having *MUST–LINK* and without instances having *CANNOT–LINK*, the target instance should be re-clustered to such cluster with the smallest SemDistance between the target instance and the medoid. If there exist clusters with instances having *CANNOT–LINK* and without instances having *MUST–LINK*, the target instance should be re-clustered to cluster having no *CANNOT–LINK* and with the smallest SemDistance between the target instance and the medoid. Otherwise, if none of the above clusters exist, which means the semantic binding rules do not apply to the target instance, the target instance will not be re-clustered.

With the guidance of semantic binding rules, we get the final result of this iteration, and the medoids are going to be updated. For each instance in one specific cluster, we calculate the sum of SemDistance of this instance to all other instances, and the instance with the smallest sum is updated to the new medoid.

4 EXPERIMENT

In order to evaluate the effectiveness of SemCluster, we set four research questions on a dataset containing 847 reports. Then we present and analyze the evaluation results.

4.1 Research Questions

We set four research questions (RQ) to comprehensively evaluate the effectiveness of SemCluster.

RQ1 (Baseline): How does SemCluster perform compared with baselines?

RQ2 (Configuration): How does SemCluster perform with different parameter configurations?

RQ3 (`Unspvsd`[4]): How does SemCluster perform compared with unsupervised configuration?

RQ4 (`Parameter`): How does SemCluster perform with different parameter settings of the semantic binding rules?

Table 1. Apps and Test Reports under Investigation

App	Domain	# Report	Bug Category	Report / Category
A1	Finance	134	9	14.89
A2	System	29	6	4.83
A3	Business	9	8	1.13
A4	Reading	13	3	4.33
A5	Reading	26	4	6.50
A6	Reading	152	8	19.00
A7	Reading	41	4	10.50
A8	System	75	7	10.71
A9	Finance	26	5	5.20
A10	Life	5	3	1.67
A11	Finance	10	2	5.00
A12	Travel	17	17	1.00
A13	Communi.	131	8	16.38
A14	Travel	88	15	5.87
A15	Music	51	8	6.38
A16	Education	4	2	2.00
A17	Life	12	3	4.00
A18	System	24	5	4.80
Sum / Avg		847	109	6.93

4.2 Experimental Subject

A dataset containing 847 crowdsourced test reports from 18 mobile apps (details can be seen in Table 1) is used for the experiment, covering different app categories, *e.g.*, Finance, Communication, Life, *etc.* The apps are labeled from A1 to A18. We cooperate with one of the most popular and representative crowdsourced testing platforms in China to build the dataset. This platform provides the crowdsourced test reports of real-world apps that delegate this platform to conduct crowdsourced testing. The platform where we collect the reports has one of the representative and most popular crowdsourced testing platforms in China and has supported many academic studies in crowdsourced testing (omitted due to anonymity policy). The platform has attracted over 20,000 users from 50 different countries, showing the great popularity and typicality of the platform. In order to better illustrate the representativeness of the platform, we have an in-depth investigation into other platforms all over the world. We find that such platforms adopt almost identical crowdsourced testing mechanisms and involve similar crowdsourced work capabilities.

[4]short for "unsupervised"

For the platform we use and other platforms, there are two kinds of participants, the requesters and the crowdworkers. The requesters publish their testing requirements as tasks on the platforms and attach the apps under test. Then crowdworkers choose to execute tasks with their own devices. If they find bugs, they will submit crowdsourced test reports, which consist of app screenshots and textual descriptions. In the textual descriptions, crowdworkers will state the steps triggering the bugs they find from the app launch to the bug occurrence and the descriptions of the bugs. Then the requesters can download all the reports within a fixed time period from crowdworkers. The capability of crowdworkers can affect the test reports. Thus, we believe that the results based on the platform can adequately reflect the generalizability of the proposed method. The number of reports for each app ranges from 4 to 152. We invite three experts in mobile app development and testing to label such test reports according to the revealed bugs. The labeling participants are three graduate students (no overlap with participants of the user study), with over five years of mobile app testing experience. They are familiar with the mechanism of crowdsourced testing. They are knowledgeable about the bugs presented in the crowdsourced test reports. The three participants are required to label all the reports independently and then discuss the disagreements until they reach a consensus on all reports. Their expertise and the cross-validation can ensure the correctness of the labeling. After the labeling and the cross-validation, the bug category (cluster) for each app ranges from 2 to 17, and the total number is 109. On average, the reports in each cluster of the whole dataset are 6.93. The duplication ratio of the crowdsourced test reports in the dataset reaches 87.13% $((1 - 109/847) \times 100\%)$. Our dataset is collected from real-world apps, some with a large number of reports but some with a few. Such a dataset shows the generalizability of SemCluster on different apps with different report quantities. We believe the clustering is necessary no matter what the report number is. First, we cannot predict the report number in crowdsourced testing. SemCluster, as an important part of crowdsourced testing, is designed to process all kinds of apps and to relieve app developers' burden by clustering similar reports to accelerate report reviewing. Second, even if the quantity is not large, clustering can group reports describing similar bugs together to save the time of report reviewing and can indicate which reports can be referred to. Actually, we cooperate with commercial platforms (hidden due to anonymity requirements) to collect reports, and the dataset of such a size is large enough to evaluate SemCluster per previous work.

4.3 Evaluation Metrics

Clustering is a mature algorithm in many fields, so we can use various metrics to make our evaluation sounder. In this paper, we use the six most widely used metrics [3, 61] that can internally or externally reflect the clustering effectiveness.

Before introducing the metrics used in this paper, we define the positive samples and negative samples. In this paper, one sample refers to a report pair consisting of two reports. For the report set containing n crowdsourced test reports to be clustered, we consider there are $C_n^2 = n(n-1)/2$ report pairs. We define report pairs that the reports should be in the same cluster as positive samples, and reports should be in different clusters as negative samples, then the true positive (TP) sample, false positive (FP) sample, true negative (TN) sample, and false negative (FN) sample are thereby defined.

Precision describes how accurate can SemCluster cluster the test reports. The range of Precision is [0, 1]. The Precision value is calculated as $TP/(TP + FP)$.

Recall describes how much SemCluster can find all the positive samples. The range of Recall is [0, 1]. The Recall value is calculated as $TP/(TP + FN)$.

F-measure is the harmonic mean of Precision value and Recall value, ranging in (0, 1]:

$$F\text{-}measure = \frac{2 * Precision * Recall}{Precision + Recall} \tag{2}$$

Purity measures the extent to which clusters contain a single class, and can be described as follows in Equation 3, where Ω refers to the clusters, C refers to the ground truth classes, and N refers to the total report number. The range of Purity is [0, 1].

$$Purity(\Omega, C) = \frac{1}{N} \sum_k \max_j \left| w_k \cap c_j \right| \tag{3}$$

ARI [23, 48] (Adjusted Rand Index) is an adjusted version (Equation 4) of RI, and ARI has a score close to 0.0 for random label assignment, and the range of ARI is [-1, 1].

$$RI = \frac{TP + TN}{TP + FP + FN + TN} \quad , \quad ARI = \frac{RI - E(RI)}{max(RI) - E(RI)} \tag{4}$$

NMI [53] refers Normalized Mutual Information (Equation 5). The I refers to the mutual information, which means the uncertainty of the changes given the class information C. The H refers to the entropy. The range of NMI is [0, 1].

$$NMI(\Omega, C) = \frac{I(\Omega; C)}{(H(\Omega) + H(C))/2} = \frac{H(\Omega) - H(\Omega|C)}{(H(\Omega) + H(C))/2} \tag{5}$$

4.4 Experiment Results

In this section, we show the experiment results of the SemCluster effectiveness evaluation. The improvement of SemCluster over baselines or different configurations is shown in the bottom part (calculated as $(SemCluster - X)/X$, X represents the results of baselines or different configurations).

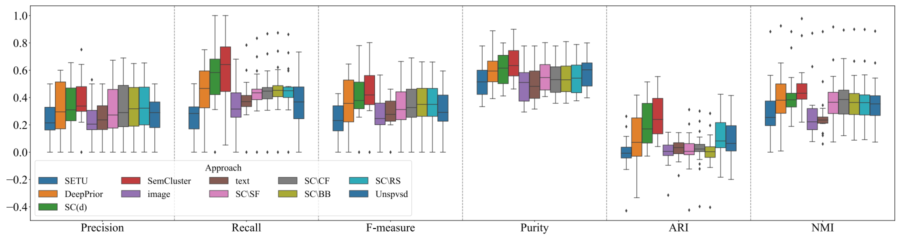


Fig. 6. Experiment Results

Table 2. Summary of Experiment Results (Improvement indicates the improvement of SemCluster over the corresponding baseline or configuration on the corresponding metric.)

		SETU	DeepPrior	SC(d)	SemCluster	Image	Text	SC \ SF	SC \ CF	SC \ BB	SC \ RS	Unspvsd
	Precision	0.23	0.31	0.36	0.39	0.24	0.25	0.30	0.32	0.32	0.32	0.28
	Recall	0.26	0.42	0.59	0.63	0.34	0.39	0.44	0.45	0.47	0.46	0.37
	F-measure	0.24	0.34	0.42	0.46	0.27	0.29	0.32	0.34	0.35	0.35	0.31
	Purity	0.52	0.60	0.61	0.66	0.51	0.50	0.56	0.54	0.54	0.55	0.58
	ARI	-0.01	0.09	0.21	0.26	0.01	0.02	0.01	0.02	0.00	0.11	0.10
	NMI	0.28	0.39	0.42	0.48	0.28	0.27	0.38	0.39	0.38	0.38	0.37
Improvement	Precision	72.76%	27.75%	8.24%	-	66.34%	57.99%	32.07%	24.83%	24.86%	22.51%	39.93%
	Recall	147.88%	49.93%	6.73%	-	88.56%	62.74%	43.72%	39.93%	36.04%	38.24%	73.73%
	F-measure	93.73%	32.77%	9.57%	-	71.93%	59.00%	41.65%	34.87%	31.24%	30.81%	46.37%
	Purity	26.38%	10.49%	8.19%	-	29.46%	32.21%	17.43%	22.71%	21.66%	20.58%	12.99%
	ARI	3051.89%	200.67%	23.90%	-	2963.72%	999.89%	2766.05%	1127.11%	8928.91%	126.34%	154.12%
	NMI	70.07%	24.25%	14.32%	-	73.42%	76.69%	25.26%	21.29%	26.73%	26.94%	30.79%

RQ1: Baseline. To compare SemCluster with the state-of-the-art approach, we implement the representative SETU algorithm proposed by Wang *et al.* in [55] and DeepPrior algorithm proposed

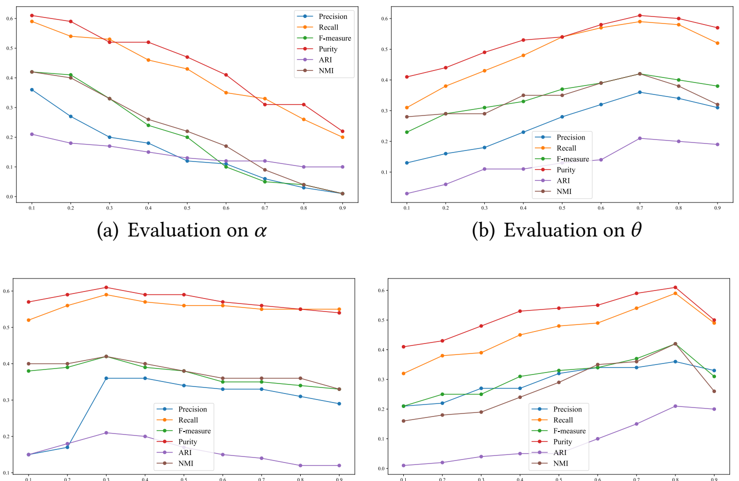
by Yu *et al.* in [63] via the reproduction packages. Our previous published demo version [16] is also taken as one baseline. The results are shown on the Fig. 6 and Table 2. According to the results, we can see that SemCluster performs better than the baselines. Compared with the three baselines, averagely in 18 apps, on Precision, SemCluster outperforms by 72.76%, 27.75%, and 8.24% over `SETU`, `DeepPrior`, and SC(d) (the same follows); on Recall, SemCluster outperforms by 147.88%, 49.93%, and 6.73%; on F-measure, SemCluster outperforms by 93.73%, 32.77%, and 9.57%; on Purity, SemCluster outperforms by 26.38%, 10.49%, and 8.19%; on ARI, SemCluster outperforms by 3051.89%, 200.67%, and 23.90%; on NMI, SemCluster outperforms by 70.07%, 24.25%, and 14.32%. According to the illustration of Fig. 6, SemCluster (the last box) is more superior compared with `SETU` (the first box). The calculation overhead is of no significant difference. Therefore, we hold that SemCluster is better than the baselines in effectively clustering the crowdsourced test reports.

RQ2: Configuration. SemCluster involves four different features extracted from the app screenshots and textual descriptions. In order to elaborate on the necessity of taking advantage of both the app screenshots and textual descriptions, we compare SemCluster with six different configurations, the `Image` and the `Text`, and the ablation configurations (SC \ SF, SC \ CF, SC \ BB, and SC \ RS represents the results of SemCluster excluding *Structure Feature*, *Content Feature*, *Bug Behavior*, and *Reproduction Step*, respectively). `Image` configuration abandons the two features of textual descriptions and only uses two features from app screenshots to characterize the similarity among test reports. Similarly, the `Text` configuration uses two features from textual descriptions to characterize the similarity among test reports. The ablation configurations mean deleting each feature and using the rest three features. The ablation results indicate how the features affect the clustering process. For example, when we compare SemCluster with SC \ SF, we consider the differences in the results indicate how the *Structure Feature* has effect during the clustering. Therefore, we use the ablation experiments to evaluate each component. According to the Fig. 6 and Table 2, we can observe that SemCluster outperforms all the configurations on all the metrics. Among different configurations, the `Image` and the `Text` perform worse than the four ablation configurations, worse than SemCluster by 19.66% – 2372.83% and 22.20% – 787.76%, respectively. This means the features are all important in our approach. Moreover, among the four ablation configurations, SC \ SF, SC \ CF, SC \ BB, and SC \ RS, the results are similar, worse than SemCluster by 8.54% – 2213.29%, 6.10% – 890.45%, 10.85% – 7187.54%, and 11.04% – 82.69%, respectively. The ablation configurations without image features are worse than those without text features, which means the image features are even more important. To sum up, considering both app screenshots and textual descriptions is necessary for effectively clustering the crowdsourced test reports, and the four features extracted from crowdsourced test reports designed in our approach SemCluster can accurately characterize and represent the features of the reports.

RQ3: `Unspvsd`. Semantic binding rules are considered to perform critical guidance to the SemCluster based on a deep understanding of app screenshots and textual descriptions. Therefore, we design experiments to evaluate the effectiveness of the semantic binding rules. The `Unspvsd` approach is the SemCluster without semantic binding rules. We compare `Unspvsd` with SemCluster and `SETU` respectively. `Unspvsd` outperforms the `SETU` by 23.46%, 42.68%, 32.36%, 11.85%, 1261.61%, 30.04% on six metrics, while compared with SemCluster, `Unspvsd` is less effective. The improvement of SemCluster over `Unspvsd` is also more than the improvement of `Unspvsd` over `SETU`, reaching 27.27%, 62.78%, 33.58%, 4.44%, 105.11%, 14.40%, on six metrics. Therefore, we believe that with the introduction of the semantic binding rules to the `Unspvsd` approach, which is SemCluster, the effectiveness will be greatly improved.

RQ4: Parameter. During the design of different semantic binding rules, we involve four parameters for four features extracted from app screenshots and textual descriptions: α, θ, β, and ω. For α, the evaluation metrics decrease with the increasing of parameter value, so we determine 0.1 for

α in SemCluster. For θ, the peak of the evaluation metric results reaches when θ is set as 0.8, so we determine 0.8 for α in SemCluster. For β, the peak of the evaluation metric results reaches when β is set as 0.3, so we determine 0.3 for α in SemCluster. For ω, the peak of the evaluation metric results reaches when ω is set as 0.9, so we determine 0.9 for α in SemCluster. Based on the evaluation results shown in Fig. 7, we can determine the optimal values for the four features in the semantic binding rule design.



(a) Evaluation on α (b) Evaluation on θ

(c) Evaluation on β (d) Evaluation on ω

Fig. 7. Evaluation Results of Different Parameter Configurations

Findings. According to the experiments, we have the following findings and in-depth insights. First, effective feature extraction will promote the characterization of the crowdsourced test reports and can help better capture the semantic distance (SemDistance). One of the important reasons for the success of SemCluster is that we mine the semantics of the app screenshots and textual descriptions of test reports instead of remaining on the simple pixel-granularity analysis. Second, mining the semantic relationships is necessary. As illustrated in the motivating examples in Section 2.2, a single utilization of app screenshots or textual descriptions will lead to a huge deviation of the final results, and will not precisely characterize the crowdsourced test reports nor calculate the similarity between them. Therefore, in this paper, we not only consider both app screenshots and textual descriptions but also focus on the screenshot-text semantic binding relationships to construct the rules that are used to guide the clustering process. Third, the role of rule-based guidance should not be ignored, and intermediate meanings among different features are important. They play a robust guiding role in the clustering and helping reach better results.

4.5 Threats to Validity

The apps involved may be a threat. We collect 847 reports on 18 different mobile apps. To evaluate the generalizability of SemCluster, our apps cover different domains, and the bug categories reach 109. Our approach design characterizes the app screenshots and textual descriptions with general strategies. **There is randomness in the process of clustering**. The clustering process may introduce randomness. Therefore, we run ten turns for each algorithm to eliminate the negative effects of randomness. **The labeling of the crowdsourced test reports may be a threat**. In order to eliminate such a threat, we invite experienced app developers and researchers focusing on mobile app testing to label the data independently. Also, their labeling data are cross-validated. If the labeling data are not consistent, they should discuss to reach a consensus. **The main language of the dataset is Chinese, which may pose a threat**. However, the data also involves English words. As to the baselines, including the SETU and DeepPrior, we have investigated their dataset (accessible part) and found that the reports they process are quite similar to ours in the language distribution, mainly Chinese and mixed with English words. Therefore, we do not have extra optimization on the baselines and directly use the code from their reproduction packages. We believe the language issue will not be an influential factor in the approach performance comparison. Moreover, the NLP techniques are strong. If we replace the feature characterization with an NLP model in another language, almost no negative impacts would occur in the effectiveness of SemCluster.

4.6 Real-world Impact Discussion

We use a user study to discuss the real-world impact of SemCluster in project maintenance. We invite six graduate students to inspect the reports for A2, containing 29 reports describing 6 bugs. The students are divided into two groups: one provided with reports with clustering results, and another provided with reports without clustering results. The participants are not provided the information about how many bugs are contained and no specific sequence of test reports is determined. We calculate the total inspected report numbers, inspecting time, and the inspected bugs. For the first group, all three participants choose to finish inspecting all 29 reports in an average time of 584 seconds, an average of 20.1 seconds for each report. The three participants inspect 6, 6, and 5 bugs, respectively. For the second group, the three participants choose to follow the clustering results and inspect one report in each cluster. They inspect the reports for an average time of 119 seconds, an average of 19.8 seconds for each report. The three participants inspect 6, 5, and 6 bugs, respectively. We conduct the significance test to show that the students of the two groups are with no significant capability differences. The average time is similar, showing that all participants have similar bug inspection capabilities, so they use similar time to review a report. For participants in the second group, which are provided with clustering results, they select one report in one cluster as a representative for this cluster. This is because reports in the same clusters (ideally) describe the same bugs, so inspecting one report is the best choice. Therefore, although participants in the second group only review 6 reports instead of all 29 reports, they can reveal as many bugs as participants in the first group do. This is the goal of clustering, which is to reduce the reports to be reviewed but keep bug-revealing capability. The results show that the participants can save much time in crowdsourced test report inspection, with the bug inspection effectiveness kept. SemCluster can greatly improve the test report inspection efficiency for project maintenance.

5 RELATED WORK

We first discuss the studies on crowdsourced testing and corresponding test report processing. Then, we discuss the GUI understanding-aided mobile testing techniques.

5.1 Crowdsourced Testing & Report Processing

Crowdsourced testing borrows the idea of crowdsourcing [22], and can be seen as a distributed paradigm to execute tasks. It has gained much popularity for the capability of testing apps in diverse environments or scenarios [19, 33]. However, the openness of crowdsourced testing is a double-edged sword. It also brings many risks and challenges, *i.e.,* crowdworker selection [56, 60], process optimization [58, 59], crowd-assisted automated testing. [20], test report optimization [55, 63], *etc.* One of the prominent risks is the crowdsourced test report quality control.

Many researchers have been focusing on crowdsourced test report processing, in order to assist the report review efficiency of developers. Feng *et al.* [17, 18] improve the test report prioritization with the feature extraction from both screenshots and texts. In addition, Yu *et al.* propose DeepPrior

[63], using image understanding towards crowdsourced test reports and better prioritizing the reports. Compared with the related studies, we process the textual descriptions similar to [63]. However, we adopt a completely different processing to the app screenshots. For SemCluster, we extract the *Structure Feature* and the *Content Feature*, which can better reflect the semantics of the app screenshots from both micro and macro perspectives. Targeted on the numerous duplicate and invalid test reports, Wang *et al.* [54] propose a test report selection technique, which classifies reports into positive ones and negative ones. Also, Wang *et al.* [57] propose an active learning method based on the local features of texts in reports to reduce the dataset scale of the report classification model. Chen *et al.* [14] model the test reports according to the report quality, and developers only need to review the high-quality ones. Report deduplication is another important direction for test report processing [2, 21, 36]. Sun *et al.* [49, 50] adopt two information retrieval models to analyze the textual information and some non-textual fields to measure the similarity of the crowdsourced test reports. Alipour [2] *et al.* and Hindle [21] *et al.* conduct a more comprehensive analysis of the context of the test reports to improve the duplicate detection accuracy, and further, they combine the contextual quality attributes, system-related topics, architecture terms, *etc.*, to improve the detection. Nguyen *et al.* [36] design a tool named DBTM, also using information-retrieval-based and topic-based features to detect duplicate reports.

SemCluster is presented in our previous demo paper [16], which describes a preliminary implementation of the proposed approach, focusing on the actual usage of the tool, and uses preliminary results to illustrate the effectiveness. Compared with the demo paper, this paper has the following contributions. First, the demo paper only considers one *MUST–LINK* rule and one *CANNOT–LINK* rule, which is preliminary, but we discuss more rules in this paper, together with conflict resolving with multiple rules. Second, we do more experiments with a sound design to illustrate the effectiveness, including the ablation experiments, the parameter determination experiments, and the extra baselines `DeepPrior` and the demo version SemCluster. Third, we conduct a user study to discuss the real-world impact of the proposed approach. Fourth, we have more detailed illustrations of the approach design and the corresponding intuition of the design to indicate the scientific contributions of our research.

The above work significantly boosts the report review efficiency via test report clustering, duplicate detection, prioritization, *etc.* However, some critical problems remain. Existing work can be divided into supervised ones and unsupervised ones. For the supervised approaches, a large amount of labeled data are required for the model construction, and the labeling work requires a large amount of human effort to ensure the correctness. Therefore, it is hard to collect large-scale labeled data and control the labeling quality. For the unsupervised approaches, the semantics of both app screenshots and textual descriptions of the crowdsourced test reports are neglected, which hinders the approaches from being effective and accurate. Therefore, we hold that a semi-supervised approach is necessary and more suitable in this scenario, which neither requires large-scale data labeling nor neglects the semantics of crowdsourced test reports. Besides, Existing studies, like [63], have preliminary considered the semantics of app screenshots and the textual descriptions. However, existing studies only consider the semantics within individual features, which completely ignore the semantic relationships among different features, like between app screenshots and textual descriptions. Since the app screenshots and textual descriptions are both utilized during the report clustering process, it is important to not only consider semantics of individual features, but also consider the semantic relationships among different features. Furthermore, although similarity measuring is one significant part for clustering, the clustering strategy also plays an important role. For SemCluster, the semantic binding rules can well guide the clustering process by correcting potential biases. We believe that compared with existing studies, this work has made much progress and contributions.

5.2 GUI Understanding-Aided Mobile Testing

GUI testing benefits much from the development of CV technologies. Many recent studies utilize CV technologies, both traditional ones and deep-learning-based ones, to solve the GUI testing obstacles from the pure visual perspective.

Some researchers focus on generating GUI structure codes with the analysis and learning of GUI images [9, 35, 37, 65]. Such tools do the reverse engineering for the GUI modeling. Cooper *et al.* [15] combines visual and textual information to detect duplicate video-based reports. GUI analysis is also widely used in the topic of recording and replaying mobile app test scripts. Behrang *et al.* [6] propose AppTestMigrator to migrate test cases between apps using the similarity among GUI widgets. Qian *et al.* [42] use CV technologies to recognize widgets and control robots to complete automated testing tasks. Bernal *et al.* [7] analyze the app GUI to replay video-based test reports. Yu *et al.* [64] utilize CV algorithms to characterize app GUIs and realize the cross-platform record and replay of mobile app test scripts. Pan *et al.* [38, 62] utilize the GUI understanding technologies to repair the mobile app test scripts. Further, Xu *et al.* [62] combines the analysis of GUI structure and visual GUI elements to more precisely repair the mobile app test scripts. Liu *et al.* [31] first design an augmenter to generate the app screenshots with issues and train a deep learning model to identify the UI display issues. Chen *et al.* [11] propose an approach that automatically predicts natural-language labels for mobile GUI components with a deep learning model. Chen *et al.* [12] discuss the applications of widget detection with state-of-the-art object detection algorithms. Chen *et al.* [10] propose a GUI design search engine to assist GUI designers. Li *et al.* [28] use a deep learning model to learn the app states and automatically generate interactions based on the analysis of the GUI information. The above work greatly enlightens us to explore the GUI information from the visual perspective to assist the crowdsourced testing. However, such work only extracts the information from the GUI images and neglects the in-depth semantics. Therefore, we are exploring the approaches to mine the existing semantics among app GUI and textual descriptions.

6 CONCLUSION

Faced with the redundancy problem of crowdsourced test reports, this paper proposes a novel test report clustering approach, SemCluster, based on semantic binding rules derived from the image and text semantics understanding. SemCluster utilizes such rules to guide the clustering as a semi-supervised mechanism. Also, this paper introduces a novel approach to characterize features of crowdsourced test reports, in order to more accurately describe the SemDistance among the crowdsourced test reports. This work is the first one that tries to explore semantic relationships between app screenshots and textual descriptions and uses semantic binding rules to guide the report clustering. Experiment results elaborate on the excellent performance of SemCluster, which outperforms the state-of-the-art approach on six metrics. An ablation study on four features illustrates that combining features from app screenshots and textual descriptions is a positive contribution. To check the real-world impact of the proposed SemCluster, we conduct a user study to show that SemCluster can greatly improve the report reviewing speed for project maintenance.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments. The work is partly supported by the National Natural Science Foundation of China (61932012, 62141215, 62272220, 62372228), the Science, Technology, and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003).

REFERENCES

[1] 1990. *Partitioning Around Medoids (Program PAM)*. Chapter 2, 68–125.

[2] Anahita Alipour, Abram Hindle, and Eleni Stroulia. 2013. A contextual approach towards more accurate duplicate bug report detection. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 183–192. https://doi.org/10.1109/MSR.2013.6624026

[3] Nejat Arinik, Vincent Labatut, and Rosa Figueiredo. 2021. Characterizing and comparing external measures for the assessment of cluster analysis and community detection. *IEEE Access* 9 (2021), 20255–20276. https://doi.org/10.1109/ACCESS.2021.3054621

[4] Sean Banerjee, Bojan Cukic, and Donald Adjeroh. 2012. Automated duplicate bug report classification using subsequence matching. In *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*. IEEE, 74–81. https://doi.org/10.1109/HASE.2012.38

[5] Sean Banerjee, Zahid Syed, Jordan Helmick, and Bojan Cukic. 2013. A fusion approach for classifying duplicate problem reports. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 208–217. https://doi.org/10.1109/ISSRE.2013.6698920

[6] Farnaz Behrang and Alessandro Orso. 2018. Test migration for efficient large-scale assessment of mobile app coding assignments. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 164–175. https://doi.org/10.1145/3213846.3213854

[7] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 309–321. https://doi.org/10.1145/3377811.3380328

[8] John Canny. 1986. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), 679–698. https://doi.org/10.1109/TPAMI.1986.4767851

[9] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676. https://doi.org/10.1145/3180155.3180240

[10] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. 2020. Wireframe-based UI design search through image autoencoder. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 3 (2020), 1–31. https://doi.org/10.1145/3391613

[11] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 322–334. https://doi.org/10.1145/3377811.3380327

[12] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object detection for graphical user interface: old fashioned or deep learning or a combination?. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1202–1214. https://doi.org/10.1145/3368089.3409691

[13] Songyu Chen, Zhenyu Chen, Zhihong Zhao, Baowen Xu, and Yang Feng. 2011. Using semi-supervised clustering to improve regression test selection techniques. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 1–10. https://doi.org/10.1109/ICST.2011.38

[14] Xin Chen, He Jiang, Xiaochen Li, Liming Nie, Dongjin Yu, Tieke He, and Zhenyu Chen. 2020. A systemic framework for crowdsourced test report quality assessment. *Empirical Software Engineering* 25, 2 (2020), 1382–1418. https://doi.org/10.1007/S10664-019-09793-8

[15] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2021. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 957–969. https://doi.org/10.1109/ICSE43902.2021.00091

[16] Mingzhe Du, Shengcheng Yu, Chunrong Fang, Tongyu Li, Heyuan Zhang, and Zhenyu Chen. 2022. SemCluster: a semi-supervised clustering tool for crowdsourced test reports with deep image understanding. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1756–1759. https://doi.org/10.1145/3540250.3558933

[17] Yang Feng, Zhenyu Chen, James A Jones, Chunrong Fang, and Baowen Xu. 2015. Test report prioritization to assist crowdsourced testing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 225–236. https://doi.org/10.1145/2786805.2786862

[18] Yang Feng, James A Jones, Zhenyu Chen, and Chunrong Fang. 2016. Multi-objective test report prioritization using image understanding. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 202–213. https://doi.org/10.1145/2970276.2970367

[19] Ruizhi Gao, Yabin Wang, Yang Feng, Zhenyu Chen, and W Eric Wong. 2019. Successes, challenges, and rethinking–an industrial investigation on crowdsourced mobile application testing. *Empirical Software Engineering* 24, 2 (2019), 537–561. https://doi.org/10.1007/S10664-018-9618-5

[20] Xiuting Ge, Shengcheng Yu, Chunrong Fang, Qi Zhu, and Zhihong Zhao. 2022. Leveraging android automated testing to assist crowdsourced testing. *IEEE Transactions on Software Engineering* 01 (2022), 1–18. https://doi.org/10.1109/TSE.2022.3216879

[21] Abram Hindle, Anahita Alipour, and Eleni Stroulia. 2016. A contextual approach towards more accurate duplicate bug report detection and ranking. *Empirical Software Engineering* 21, 2 (2016), 368–410. https://doi.org/10.1007/S10664-015-9387-3

[22] Jeff Howe et al. 2006. The rise of crowdsourcing. *Wired magazine* 14, 6 (2006), 1–4.

[23] Lawrence Hubert and Phipps Arabie. 1985. Comparing partitions. *Journal of classification* 2, 1 (1985), 193–218.

[24] Nicholas Jalbert and Westley Weimer. 2008. Automated duplicate detection for bug tracking systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 52–61. https://doi.org/10.1109/DSN.2008.4630070

[25] Seunghui Jang, Ki Yong Lee, and Yanggon Kim. 2020. An Approach to Improving the Effectiveness of Data Augmentation for Deep Neural Networks. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 1290–1295. https://doi.org/10.1109/COMPSAC48688.2020.00-78

[26] He Jiang, Xin Chen, Tieke He, Zhenyu Chen, and Xiaochen Li. 2018. Fuzzy clustering of crowdsourced test reports for apps. *ACM Transactions on Internet Technology (TOIT)* 18, 2 (2018), 1–28. https://doi.org/10.1145/3106164

[27] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1746–1751.

[28] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073. https://doi.org/10.1109/ASE.2019.00104

[29] Zenan Li, Xiaoxing Ma, Chang Xu, Chun Cao, Jingwei Xu, and Jian Lü. 2019. Boosting operational dnn testing efficiency through conditioning. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 499–509. https://doi.org/10.1145/3338906.3338930

[30] Di Liu, Yang Feng, Xiaofang Zhang, James Jones, and Zhenyu Chen. 2020. Clustering Crowdsourced Test Reports of Mobile Applications Using Image Understanding. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.3017514

[31] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl eyes: Spotting ui display issues via visual understanding. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 398–409. https://doi.org/10.1145/3324884.3416547

[32] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

[33] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. 2017. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software* 126 (2017), 57–84. https://doi.org/10.1016/J.JSS.2016.09.015

[34] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[35] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering* 46, 2 (2018), 196–221. https://doi.org/10.1109/TSE.2018.2844788

[36] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 70–79. https://doi.org/10.1145/2351676.2351687

[37] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 248–259. https://doi.org/10.1109/ASE.2015.32

[38] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. 2020. GUI-Guided Test Script Repair for Mobile Apps. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.3007664

[39] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems (TODS)* 40, 1 (2015), 1–40. https://doi.org/10.1145/2699485

[40] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Information Systems* 56 (2016), 157–173. https://doi.org/10.1016/j.is.2015.08.004

[41] Tomi Prifti, Sean Banerjee, and Bojan Cukic. 2011. Detecting bug duplicate reports through local references. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. 1–9. https://doi.org/10.1145/2020390.2020398

[42] Ju Qian, Zhengyu Shang, Shuoyan Yan, Yan Wang, and Lin Chen. 2020. Roscript: a visual script driven truly non-intrusive robotic testing system for touch screen applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 297–308. https://doi.org/10.1145/3377811.3380431

[43] Eric Steven Raymond. 2003. Applying the Rule of Least Surprise. *The Art of Unix Programming* (2003), 525.

[44] LKPJ Rdusseeun and P Kaufman. 1987. Clustering by means of medoids. In *Proceedings of the Statistical Data Analysis Based on the L1 Norm Conference, Neuchatel, Switzerland*. 405–416.

[45] Peter Seebach. 2001. The cranky user: The principle of least astonishment. *IBM DeveloperWorks* (2001).

[46] Diego F Silva and Gustavo EAPA Batista. 2016. Speeding up all-pairwise dynamic time warping matrix calculation. In *Proceedings of the 2016 SIAM International Conference on Data Mining*. SIAM, 837–845. https://doi.org/10.1137/1.9781611974348.94

[47] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[48] Douglas Steinley. 2004. Properties of the Hubert-Arable Adjusted Rand Index. *Psychological methods* 9, 3 (2004), 386.

[49] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 253–262. https://doi.org/10.1109/ASE.2011.6100061

[50] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 45–54. https://doi.org/10.1145/1806799.1806811

[51] Ashish Sureka and Pankaj Jalote. 2010. Detecting duplicate bug report using character n-gram-based features. In *2010 Asia Pacific Software Engineering Conference*. IEEE, 366–374. https://doi.org/10.1109/APSEC.2010.49

[52] Robert L Thorndike. 1953. Who belongs in the family? *Psychometrika* 18, 4 (1953), 267–276.

[53] Nguyen Xuan Vinh, Julien Epps, and James Bailey. 2010. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *The Journal of Machine Learning Research* 11 (2010), 2837–2854. https://doi.org/10.5555/1756006.1953024

[54] Junjie Wang, Qiang Cui, Qing Wang, and Song Wang. 2016. Towards effectively test report classification to assist crowdsourced testing. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10. https://doi.org/10.1145/2961111.2962584

[55] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. 2019. Images don't lie: Duplicate crowdtesting reports detection with screenshot information. *Information and Software Technology* 110 (2019), 139–155. https://doi.org/10.1016/J.INFSOF.2019.03.003

[56] Junjie Wang, Song Wang, Jianfeng Chen, Tim Menzies, Qiang Cui, Miao Xie, and Qing Wang. 2019. Characterizing crowds to better optimize worker recommendation in crowdsourced testing. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2918520

[57] Junjie Wang, Song Wang, Qiang Cui, and Qing Wang. 2016. Local-based active classification of test report to assist crowdsourced testing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 190–201. https://doi.org/10.1145/2970276.2970300

[58] Junjie Wang, Ye Yang, Rahul Krishna, Tim Menzies, and Qing Wang. 2019. iSENSE: Completion-aware crowdtesting management. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 912–923. https://doi.org/10.1109/ICSE.2019.00097

[59] Junjie Wang, Ye Yang, Tim Menzies, and Qing Wang. 2020. isense2. 0: Improving completion-aware crowdtesting management with duplicate tagger and sanity checker. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–27. https://doi.org/10.1145/3394602

[60] Junjie Wang, Ye Yang, Song Wang, Yuanzhe Hu, Dandan Wang, and Qing Wang. 2020. Context-aware in-process crowdworker recommendation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1535–1546. https://doi.org/10.1145/3377811.3380380

[61] Junjie Wu, Hui Xiong, and Jian Chen. 2009. Adapting the right measures for k-means clustering. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 877–886. https://doi.org/10.1145/1557019.1557115

[62] Tongtong Xu, Minxue Pan, Yu Pei, Guiyin Li, Xia Zeng, Tian Zhang, Yuetang Deng, and Xuandong Li. 2021. GUIDER: GUI structure and vision co-guided test script repair for Android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 191–203. https://doi.org/10.1145/3460319.3464830

[63] Shengcheng Yu, Chunrong Fang, Zhenfei Cao, Xu Wang, Tongyu Li, and Zhenyu Chen. 2021. Prioritize Crowdsourced Test Reports via Deep Screenshot Understanding. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 946–956. https://doi.org/10.1109/ICSE43902.2021.00090

[64] Shengcheng Yu, Chunrong Fang, Yexiao Yun, and Yang Feng. 2021. Layout and Image Recognition Driving Cross-Platform Automated Mobile Testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1561–1571. https://doi.org/10.1109/ICSE43902.2021.00139

[65] Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. 2021. GUIGAN: Learning to Generate GUI Designs Using Generative Adversarial Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 748–760. https://doi.org/10.1109/ICSE43902.2021.00074

[66] Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 434–443.

Received 2023-09-28; accepted 2024-04-16