Automated Classification of Actions in Bug Reports of Mobile Apps

Hui Liu*
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
Liuhui08@bit.edu.cn

Mingzhu Shen
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
3120181025@bit.edu.cn

Jiahao Jin
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
3220180713@bit.edu.cn

Yanjie Jiang
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
jiangyanjie@bit.edu.cn

ABSTRACT

When users encounter problems with mobile apps, they may commit such problems to developers as bug reports. To facilitate the processing of bug reports, researchers proposed approaches to validate the reported issues automatically according to the *steps to reproduce* specified in bug reports. Although such approaches have achieved high success rate in reproducing the reported issues, they often rely on a predefined vocabulary to identify and classify actions in bug reports. However, such manually constructed vocabulary and classification have significant limitations. It is challenging for the vocabulary to cover all potential action words because users may describe the same action with different words. Besides that, classification of actions solely based on the action words could be inaccurate because the same action word, appearing in different contexts, may have different meaning and thus belongs to different action categories. To this end, in this paper we propose an automated approach, called *MaCa*, to identify and classify action words in Mobile apps' bug reports. For a given bug report, it first identifies action words based on natural language processing. For each of the resulting action words, *MaCa* extracts its contexts, i.e., its enclosing segment, the associated UI target, and the type of its target element by both natural language processing and static analysis of the associated app. The action word and its contexts are then fed into a machine learning based classifier that predicts the category of the given action word in the given context. To train the classifier, we manually labelled 1,202 actions words from 525 bug reports that are associated with 207 apps. Our evaluation results on manually labelled data suggested that *MaCa* was accurate with high accuracy varying from 95% to 96.7%. We also investigated to what extent *MaCa* could further improve existing approaches (i.e., Yakusu and ReCDroid) in reproducing bug reports. Our evaluation results suggested that integrating *MaCa* into existing approaches significantly improved the success rates of ReCDroid and Yakusu by 22.7% = (69.2%-56.4%)/56.4% and 22.9%= (62.7%-51%)/51%, respectively.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Machine learning algorithms.*

KEYWORDS

Mobile Testing, Classification, Test Case Generation, Bug report

ACM Reference Format:
Hui Liu, Mingzhu Shen, Jiahao Jin, and Yanjie Jiang. 2020. Automated Classification of Actions in Bug Reports of Mobile Apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3395363.3397355

1 INTRODUCTION

Mobile applications are highly popular [31]. Apple store has 2.2 millions of mobile apps for download whereas Google Play has 2.8 millions. Such mobile apps, however, contain numerous defects that frustrate users [46]. When users encounter problems with such apps, they may commit such problems to developers as bug reports [9, 15] via bug tracking systems, like Bugzilla [1], Google Code Issue Tracker [4], and Github Issue Tracker [3]. To facilitate the resolution of the reported issues, such issue tracking systems often request and help users to specify detailed steps for reproducing the reported issues [20]. As a result, as suggested by existing empirical studies, such bug reports often contain necessary information in natural language for reproduction of the reported issues [15, 45].

Manual validation of the reported issues is often time consuming, and thus automated validation is highly desirable [15, 46]. Fazzini et al. [15] proposed an automated approach, called Yakusu, to translate bug reports into test cases for mobile apps. To build the mapping from textual bug reports to UI actions, Yakusu conducts lexicon normalization and object standardization on the textual bug

*corresponding author

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397355

reports. Lexicon normalization is to replace non-standard words with their canonical form to simplify parsing and understanding of actions. The key of lexicon normalization is to replace action words with one of the standard action words (vocabulary) that are selected manually by analyzing a corpus of bug reports. For example, Yakusu replaces the action word "*Tap*" in "*Tap on the Publish button*" with "*Click*". Object standardization is to replace phases that referring to target UI element with the ID of the corresponding UI element in app (called ontology in the paper). After the lexicon normalization and object standardization, Yakusu explores each clause in the bug report. If the root of a clause is a predefined action word, Yakusu extracts an abstract action for the clause based on the dependency tree of the clause. Otherwise, Yakusu computes the lexical similarity between the whole clause and properties of UI elements, and retrieves the most similar UI element. Finally, Yakusu generates test cases by mapping the resulting abstract actions to UI actions that are discovered by exploring mobile apps. More recently, Zhao et al. [46] proposed another automated approach, call ReCDroid, to automatically reproduce Android application crashes from bug reports. Similar to Yakusu, ReCDroid also employs a predefined (manually built) vocabulary of action words. ReCDroid classifies such action words into three categories, i.e., CLICK, EDIT, and GESTURE. ReCDroid identifies and classifies events/actions in bug reports by matching the reports against the predefined vocabulary. For different types of events, ReCDroid employs different grammar patterns to match the sentence, and creates abstract actions based on the matching patterns. Finally, ReCDroid maps the resulting abstract actions to UI actions by dynamic exploration of the associated app with dynamic ordered event tree.

Both Yakusu and ReCDroid are accurate, achieving high success rate in reproducing the reported issues. However, both Yakusu and ReCDroid rely on a predefined vocabulary to identify and classify actions in bug reports. Depending on manually constructed vocabulary (and classification) results in significant limitations. First, it is challenging for the vocabulary to cover all potential action words. Users may describe the same action with different words, i.e., new words out of the predefined vocabulary could be employed by users to describe existing actions. For example, "*Touch the Menu Button*", "*Press the Menu Button*", and "*Click the Menu Button*" employed different action words to represent the same action. Second, the classification of actions solely based on the action words could be inaccurate. The same action word, appearing in different contexts, may have different meaning and thus belongs to different action categories. For example, the same action word '*change*' has different meaning and belongs to different action categories in the following sentences: "*change orientation ...*" (belonging to category ROTATE), "*Change the dollar amount ...*" (belonging to category TYPE), and "*Change the language ...*" (belonging to category CLICK). Inaccurate identification and classification of action words could result in failure of the automated validation approaches like Yakusu and ReCDroid. For example, ReCDroid will apply improper patterns to match clauses if the action words are recognized or classified incorrectly, which in turn results in failure of action discovery.

To this end, in this paper we propose a Machine learning based automatic approach (called *MaCa*) to identifying and Classifying actions in Mobile apps' bug reports. For a given bug report, *MaCa* identifies action words based on natural language processing and dependency trees. For each of the resulting action words, *MaCa* extracts its contexts, i.e., its enclosing segment, the associated UI target, and the type of its target element. The extraction is conducted based on both natural language processing and static analysis of the associated apps. The action word and its contexts are then fed into a machine learning based classifier that predicts the category of the given action word in the give contexts. To train the classifier, we manually labelled 1,202 actions words from 525 bug reports that are associated with 207 apps. The manual labelling classifies the actions words into the following five categories: *CLICK*, *TYPE*, *ROTATE*, *SWIPE*, and *SCROLL*. Our evaluation results on manually labelled data suggested that *MaCa* was accurate with high accuracy varying from 95% to 96.7%. We also investigated to what extent the proposed approach can further improve existing approaches (i.e., Yakusu and ReCDroid) in reproducing bug reports by integrating *MaCa* into existing approaches. Evaluation results suggested that integrating *MaCa* into existing approaches significantly improved the success rates of ReCDroid and Yakusu by 22.7% = (69.2%-56.4%)/56.4% and 22.9%= (62.7%-51%)/51%, respectively. Evaluation results also suggested that the proposed approach was accurate in identifying and classifying such action words (called *unseen words*) that did not appear in the training data.

In summary, our paper makes the following contributions:

- An automated technique for identifying and classifying actions in Mobile apps' bug reports written in natural languages.
- A publicly available implementation of the technique, and a manually labelled dataset for training of the approach [39].
- An empirical study providing initial evidence of the effectiveness of our approach and its potential in improving existing approaches in automated validation of bug reports. The replication package is available on Github [39].

2 MOTIVATING EXAMPLES

One reason for machine learning based automated classification of action words is that users may leverage different words to represent essentially the same actions. The following bug report (issue #2523 of app *MvvmCross* [7]) is a typical example:

> 1. **Select** the second tab on the Bottom Navigation View; 2. **Click** the button with the text 'Click Me'; 3. **Press** the hardware back button

We notice that the reporter leverages three different words, i.e., "*select*", "*click*", and "*press*", to describe the same action (clicking UI elements). Such diversified description makes it challenging to collect and classify all potential action words manually.

Another reason for machine learning based classification of action words is that the same word may have different meaning in different contexts. For example, the word "*change*" in the following bug reports represents different actions:

(1) Issue #783 of app *Lockwise*[6]: "*... 4.**Change** the language to an unsupported one ...*";
(2) Issue #22 of app *LibreNew*[5]: "*... **Change** server address to an invalid one, e.g., xxyyzz ...*";
(3) Issue #34330 of app *Flutter*[2]: "*... 3.**Change** orientation a few times ...*".

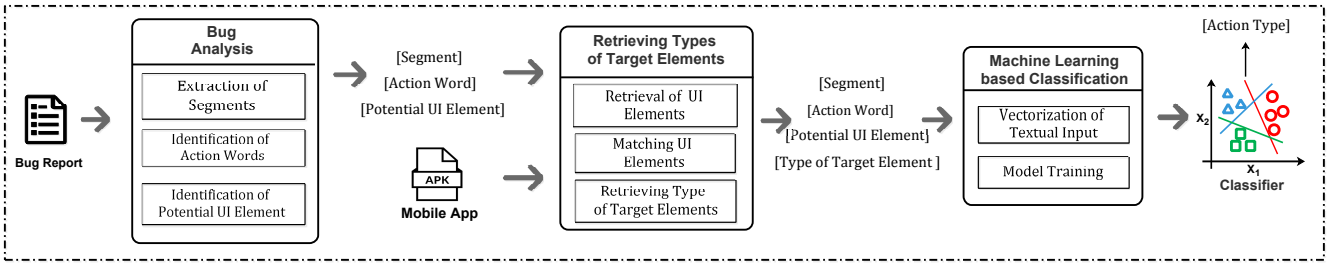

Figure 1: Overview

The first "*change*" [6] represents a click action to change/select a specific language from a given list. The second "*change*" [5] represents a typing action, e.g., to type in server address "*xxyyzz*". The third "*change*" [2] represents rotation of the mobile device. Such polysemous words make it challenging to classify action words solely based on the words themselves. Additional information, like the contexts of the words, should be considered as well.

We conclude based on the preceding examples that it may result in significant limitations if we manually construct vocabularies of actions words and classify them based action words only. To this end, the proposed approach employs machine learning techniques to identify and classify actions, and thus has the potential to handle unknown/new action words because machine learning techniques have the potential to handle new items that are not contained in training data [18, 33]. Besides that, it leverages the contexts of action words for classification, and thus has the potential to classify polysemous words correctly.

3 APPROACH

3.1 Overview

An overview of the proposed approach (MaCa) is presented in Fig. 1. The approach takes bug reports as input and finally generates a list of actions as well as their action types. Three main steps of the approach are bug analysis, retrieving types of target elements, and machine learning based classification.

Bug analysis decomposes steps to reproduce (notated as s2r) into segments that represent individual actions in reproduction of the reported issues. For example, s2r "*Press a custom server, and input an invalid server address*" should be decomposed into two segments "*Press a custom server*" and "*input an invalid server address*". For each of the resulting segments, the proposed approach leverages natural language processing techniques to identify action word, and potential UI elements associated with the action. For example, from segment "*Press a custom server*", we should identify action word "*Press*" and its associated potential UI element "*a custom server*".

Retrieving types of target elements takes input of an action word, its enclosing segment, its associated potential UI elements, and the associated mobile app. From the associated mobile app, the proposed approach leverages static analysis to identify all UI elements that may be created statically or dynamically by the app [42]. The proposed approach leverages word2vec [27] to encode properties of the resulting UI elements as well as the textual description (in bug report) of potential UI elements. From the resulting UI elements, the proposed approach leverages lexical similarity to identify most likely target UI element for the given action, and returns the type of the target element. Taking the action word "*Press*" in segment "*Press a custom server*" for example, the proposed approach matches potential UI element ("*a custom server*") against all UI elements in app *NewsBlur*, which results in a matching UI element whose ID is *login_custom_server*. Consequently, the proposed approach returns the type of this element, i.e., *TextView*, as the type of the target element for action word "*Press*".

Machine learning based classification takes input of an action word, its enclosing segment, its associated potential UI elements, and the type of its target element. It leverages machine learning based classification techniques to predict the type of the action. Considering the significant imbalance of data, the proposed approach computes weights for each of the categories (labels), and optimizes the training process with the resulting weights. The proposed approach employs powerful BERT [14] to turn the textual input into a single numeric vector. Finally, it leverages a logistic regression model [10] to predict the action type of the resulting numeric vector. Notably, the training of the employed classifier requires labelled training data, and thus we manually created a training dataset based on reports from Github.

3.2 Analyzing Bug Reports

We analyze bug reports to extract action words and their contexts from steps to reproduce (s2r). As the first step on the processing of s2r, we replace all hyperlinks in bug reports with a special symbol "*URL*" because the hyperlinks are often lengthy and detailed addresses (contents of the links) are often irrelevant to parsing of natural languages or the classification of actions. For example "*Go on https://download.lineageos.org/*" would be revised as "*Go on URL*".

After the preprocessing, we partition s2r into segments by *spaCy* [8], a widely used natural language processing toolkit. Basically, *spaCy* decomposes sentences according to punctuation marks, and the decomposition is often accurate. However, for clauses containing conjunctions, e.g., "*and*", the decomposition could be incomplete. For example, *spaCy* takes "*Alternatively press the mail and from contextual menu click on delete*" as a single segment although it should be partitioned into two segments: "*Alternatively press the mail*" and "*from contextual menu click on delete*". To this end, we further decompose the segments returned by *spaCy* according to a list of conjunctions, i.e., "*and*", "*or*", "*after*", and "*before*".

For each of the resulting segments, we parse it into a dependency tree by *spaCy* [8]. Two sample dependency trees are presented in Fig. 2 where the associated segments are "*click your library*" and

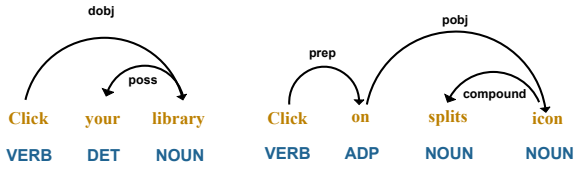

Figure 2: Dependency Trees of Segments

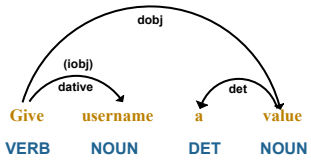

Figure 3: Dependency Tree and Indirect Object

"*click on splits icon*", respectively. From the dependency trees, we locate the verbs (the roots of the trees), e.g., "*click*" for the sample trees in Fig. 2. This verb is then taken as an action word. To locate the target UI elements associated with the discovered action word, we select the nouns/noun phases with the following special syntactic dependency with the action word: direct object (dobj), preposition object (pobj), and indirect objects (iobj). Direct object (dobj, as shown on the left part of Fig. 2) is the noun phrase which is the accusative object of the verb. The object of a preposition (pobj, as shown on the right part of Fig. 2) is the noun or pronoun governed by a preposition. Pobj is usually the noun or pronoun to the right of the preposition. The indirect object (iobj) of a verb, as shown in Fig. 3, is any nominal phrase that is a core argument of the verb but is not its subject or direct object.

We collect all such nouns and noun phases as the potential UI elements for the action. For the example on the left part of Fig. 2, the action word is "*click*" and its associated potential UI element is {"*you library*"}. For the example on the right part of Fig. 2, the action word "*click*" is associated with potential UI element {"*splits icon*"}.

3.3 Retrieving Types of Target Elements

In the preceding section, we extract action words as well as their associated potential UI elements. In this section, we search for the most likely UI element (notated as *actualElement*) in the associated app that the give action word *aw* may act on. Detailed algorithm is presented in Algorithm 1. Input of the algorithm includes the given action word *aw* and the associated mobile app (notated as *app*). Output of the algorithm is the element type of the target of the action word.

To retrieve the most likely UI element in the associated app that the action targets, we collect all UI elements that are created by the mobile app (Line 3). We analyze the resource configuration files (XML documents) and collect a set of UI elements that are created at design time [36]. Besides that, we also analyze the source code of the app to retrieve a set of UI elements that could be created by the app at runtime (with the help of Gator [42]). All such UI elements are returned and assigned to collection *UES*.

Algorithm 1: : Retrieving Type of Action Target

```
Input: aw; // action word
       app // the associated app
Output: ElementType // Type of the target UI element
1  maxSimilarity = 0 ;
2  actualElement = null ;
3  UES = RetrieveUIelements(app) ;
4  for each pElement in aw.PotentialUIelements do
5      for each element in UES do
6          similarity = ComputeSim(element, pElement+aw) ;
7          if similarity > maxSimilarity then
8              maxSimilarity = similarity ;
9              actualElement = element ;
10         end
11     end
12     appSim = Sim(app.getName(), pElement+aw);
13     if appSim > maxSimilarity then
14         maxSimilarity = appSim;
15         actualElement = app;
16     end
17 end
18 if actualElement ≠ null AND maxSimilarity > β then
19     ElementType = actualElement.type;
20 else
21     ElementType = null ;
22 end
23 return ElementType
```

The algorithm then goes into the main iteration (Lines 4-17). On each iteration (Lines 5-16), the algorithm selects the most likely UI element in the associate app for a given potential UI element (notated as *pElement*) associated with the action word *aw*. The selection is based on the lexical similarity between *pElement* and UI elements within the app. The ID and display text (notated as *dText*) of UI element (notated as *element*) are leveraged for the computation of lexical similarity:

$$ComputeSim(element, pElement + aw) = max(Sim(element.ID, pElement + aw), Sim(element.dText, pElement + aw)) \quad (1)$$

The action word *aw* (a verb) is appended to *pElement* for similarity computation because developers may name UI elements with verbs on some special cases. Method *Sim* computes the lexical similarity between two texts by representing them as digital vectors (also known as embedding) by well-known word2vec [27], and computing the cosine similarity of the two vectors. Notably, word2vec converts a single word into a vector, and thus we present the whole text as the average of the vectors for words within the text. On Line 12, the algorithm also computes the similarity between the potential UI element and the name of the associated app. It is likely that some users may describe the launch of the app as the first step to reproduce, and in this case the target element is the app itself.

The two iterations select the most likely UI element as the target of the action word, noted as *actualElement*. If its lexical similarity with one of the potential UI element descriptions is higher than β,

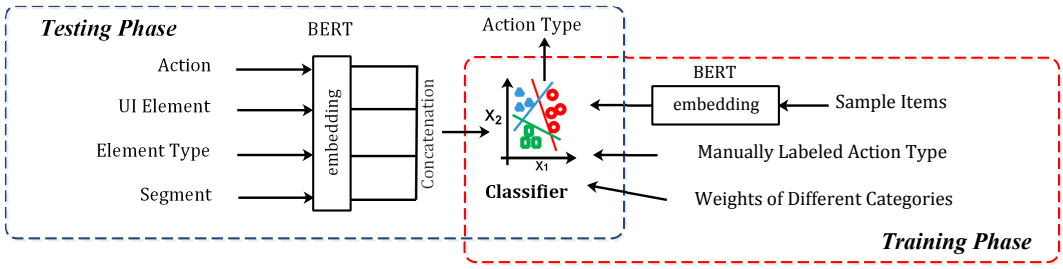

Figure 4: Machine Learning Based Classification

the selection succeeds and the type of *actualElement* is returned (Lines 18-19). Otherwise, the selection fails and the algorithm returns a special symbol *null*. Notably, we empirically set the minimal similarity β to 0.5 that is frequently employed as default values in machine learning based classifiers.

3.4 Machine Learning Based Classification

The last part of the approach is a machine learning based classifier that classifies an action word aw into one of the categories $CaS = \{c_1, c_2 \dots c_n\}$ where n is the total number of categories. The structure of the classifier and the training/testing process are presented in Fig. 4.

Although we have tried to retrieve the types of target elements (as specified in Section 3.3), a machine learning based classifier is still needed because of the following reasons. First, the retrieved type of the target element alone could be insufficient to determine the action type. For example, on "TextView" users may perform CLICK or SCROLL (even more action types on the latest iPhone). Consequently, we cannot determine the action type solely based on the type of associated target element. Second, there is no guarantee that the static analysis can successfully retrieve the type of the target element. The second step of the approach ("retrieving types of target elements") returns NULL when it fails to retrieve the target element based on the textual description of UI elements in S2R. It is challenging to match UI elements according to the textual description of UI elements. This is also the major reason for existing approaches (like Yakusu and ReCDroid) to recognize/classify action words first and then match associated UI elements according to the identified action types (not the other way around).

It is likely that the data (both training data and testing data) are not balanced, i.e., some categories are significantly larger than others. However, unbalanced data will cause machine learning models to focus on major categories (with the larger numbers of items) and to undervalue other minor categories. Machine learning based classifiers often leverage the weights (distribution) of different categories to cope the imbalance [38, 41]. To this end, we compute the weights of each category as follows:

$$wt_i = \frac{1}{log(ItemNumber(c_i))} \tag{2}$$

where wt_i is the weight for category c_i, and $ItemNumber(c_i)$ is the number of times belonging to category c_i. With the resulting weights wt, we compute the loss of the classifier as the weighted average of the loss for individual items during the training phase. According to the definition of the weights, the smaller a category is, the larger its weight would be. Consequently, the weighted loss enlarges the penalty for the misclassification on items belonging to small categories.

Input of the classifier is a tuple:

$$item =< aw, EnclosingSegment(aw), UIelements(aw), ElementType(aw) > \tag{3}$$

aw is an action word extracted in Section 3.2, $EnclosingSegment(aw)$ is the segment where aw is extracted, $UIelements(aw)$ is a list of potential UI elements associated with aw (as extracted in Section3.2), and $ElementType(aw)$ is the type of the most likely target of aw.

The input *item* is essentially a sequence of texts, and such texts should be vectorized before they could be fed into classifiers. We convert each part of the *item* into a vector by BERT (Bidirectional Encoder Representations from Transformers) [14] (as shown in the embedding layer of Fig. 4). BERT was proposed recently by researchers from Google, and has been proved effective in vectorizing texts [17, 21, 35]. After converting each of the texts into a numeric vector, we concatenate the resulting vectors as a longer one that is finally fed into the machine learning based classifier:

$$numericInput = [BERT(aw), BERT(EnclosingSegment(aw)), BERT(UIelements(aw)), BERT(ElementType(aw))] \tag{4}$$

For the machine learning based classifier, we empirically employ a logistic regression model [10]. As revealed in Section 4, logistic regression results in higher accuracy of the classification than other common machine learning techniques, including Naive Bayesian [23], Random Forest [24], Support Vector Machine [40], AdaBoost [34], Multi-Layer Perceptron [30], Convolutional Neural Network [22], Long Short-Term Memory [19], and Bi-directional Long Short-Term Memory [37]. Notably, the proposed approach is not confined to logistic regression, and it can work with any classification techniques that label numeric vectors with one of the predefined labels. The selected classifier employs a *softmax* function that could be formalized as follows:

$$P(x;\theta) = \frac{1}{\sum_{i=1}^{n} e^{w_i \cdot x + b_i}} \begin{bmatrix} e^{w_1 \cdot x + b_1} \\ e^{w_2 \cdot x + b_2} \\ \dots \\ e^{w_n \cdot x + b_n} \end{bmatrix} \tag{5}$$

A training process should optimize the parameters $w =< w_1, w_2 \dots w_n >$ and $b =< b_1, b_2 \dots b_n >$ on training data. After training, the model would compute the probability that item $x = numericInput$

belongs to jth category as follows:

$$P(y = j|x) = \frac{e^{w_j \cdot x + b_j}}{\sum_{i=1}^{n} e^{w_i \cdot x + b_i}} \quad (6)$$

4 EVALUATION

In this section, we evaluate the proposed approach by investigating the following research questions:

- **RQ1**: Is *MaCa* accurate in classifying actions in Mobile apps' bug reports?
- **RQ2**: Is *MaCa* accurate in identifying and classifying action words that are unseen in training data?
- **RQ3**: To what extent do different classification techniques affect the performance of *MaCa*?
- **RQ4**: To what extent do the contexts of action words affect the performance of *MaCa*?
- **RQ5**: To what extent can *MaCa* improve the performance of automated bug report validation by integrating it into existing approaches (i.e., ReCDroid and Yakusu)?
- **RQ6**: Dose *MaCa* outperform keyword match method based on pre-defined vocabulary?

4.1 Dataset

The dataset is composed of three parts. The first part of the dataset is the 51 bug reports created by Zhao et al. [46] for the evaluation of ReCDroid, notated as *icseData*. They randomly crawled 300 bug reports containing keywords "*crash*" and "*exception*" from Github, included 15 bug reports from FUSION [28], and 25 bug reports from the 62 reproducible bug reports collected by Fazzini et al. [15]. They applied manual filtration on such bug reports to exclude non-reproducible bug reports as well as those that did not result in actual app crashes. Finally, 51 bug reports were selected for their evaluation.

The second part of the dataset is the 39 bug reports collected by Fazzini et al. [15], notated as *isstaData*. Fazzini et al. [15] retrieved issues from Github using keywords "*android*", "*crash*", "*reproduce*", and "*version*". From the resulting issues, they randomly selected 100 for manual analysis, and finally picked up 62 reproducible bug reports for the evaluation of Yakusu. However, only 39 of them were still available and reproducible on the day we accessed it, and thus *isstaData* includes 39 bug reports only.

The third part, notated as *newData*, contains 525 bug reports of mobile apps. We collected such bug reports from Github that contain "*Android (or iOS)*" and "*reproduce*". We manually checked the resulting top 2,000 bug reports, and manually excluded those that did not contain steps to reproduce and those that have been included by either *icseData* or *isstaData*. Finally, 525 bug reports containing useful steps to reproduce were kept for the evaluation. For each of the bug reports, the first two authors manually identified steps to reproduce, and manually classified the action words in such steps. We also manually labelled the UI types for those whose associated Apps are not available. The participants achieved high agreement on the classification, resulting in a high Cohen's Kappa coefficient [13] of 0.7866 that suggests excellent inter-rater agreement.

4.2 Experiment Design

4.2.1 RQ1: Accuracy of MaCa*:* To measure the accuracy of *MaCa*, we conducted ten-fold cross validation on *newData* that we created manually. We partitioned the 525 bug reports in *newData* into ten groups with approximately equal size (either 52 or 53). On each fold, we selected one group as the testing data whereas the others were taken as training data. Notably, each group was taken as testing data for once. We trained *MaCa* on the selected training data, applied the resulting classifier to the testing data, and computed the accuracy of the classification on the testing data:

$$accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ items\ in\ testing\ data} \quad (7)$$

where a prediction was correct only if the predicted action type was exactly the same as manually labelled. The denominator of the equation was the total number of items instead of the number of predictions. Consequently, action words that MaCa failed to identify (and thus fails to classify) was counted in by the denominator.

Besides accuracy, we also computed some common performance metrics for multiple-classification, i.e., precision and recall for each category [32], and *macro F*1 [25] over the whole dataset. Notably, for our task (where each of the items would be assigned a single label), *micro F*1 equals accuracy, and thus we do not present *micro F*1 in the rest of the paper.

To further evaluate its performance, we also conducted cross-dataset validation: we trained MaCa with the *newData* and evaluated the resulting MaCa on *icseData* and *isstaData*, respectively.

4.2.2 RQ2: Unseen Words: One of the weakness of predefined vocabulary and classification is that they cannot identify or classify new action words outside the vocabulary (we call them unseen words). To investigate how well MaCa can handle such unseen words, we identified the rarest actions words (notated as *Raws*) in *NewData*, removed bug reports containing such words (resulting in a new data *NewData'*), trained MaCa with *NewData'* from scratch, and applied the resulting MaCa to *Raws*.

4.2.3 RQ3: Effect of Classification Techniques: As specified in Section 3.4, the proposed approach depends on classification techniques to predict action types. Although the approach selects logistic regression as the default classification technique, the approach itself is open to any of the classification techniques that can label a numeric vector with one of the predefined labels. Consequently, we investigated how different classification techniques may affect the performance of the proposed approach, and whether the default setting, i.e., logistic regression, results in the best performance. To this end, we replaced the logistic regression-based classifier in MaCa with different classifiers, and repeated the evaluation as introduced in Section 4.2.1. More specifically, we have tried the following classification techniques: Random Forest [24], Support Vector Machine (SVM) [40], Long Short-Term Memory (LSTM) [19], Bi-directional Long Short-Term Memory (Bi-LSTM) [37], Convolutional Neural Network (CNN) [22], Multi-Layer Perceptron (MLP) [30], Naive Bayesian [23], and AdaBoost [34]). Such techniques were selected because they are accurate and have been widely used.

4.2.4 RQ4: Effect of Contexts: As specified in Section 3.4, *MaCa* leverages various contexts (i.e., its target UI element, type of its

Table 1: Performance of MaCa (Ten-Fold Cross-Validation)

Metrics	1#	2#	3#	4#	5#	6#	7#	8#	9#	10#	Avg
Accuracy	97.5%	96.7%	96.7%	94.2%	96.7%	95.8%	99.2%	97.5%	95.0%	96.7%	96.6%
Macro F1	92.7%	94.8%	95.2%	92.1%	86.7%	91.5%	92.0%	89.6%	86.2%	93.1%	91.4%

target UI element, and its enclosing segment) of the action word to be classified. To investigate to what extent such contexts may affect the performance of the classification, we employed different contexts of the words and repeated the ten-fold evaluation on *newData* (as specified in Section 4.2.1). Notably, as introduced in Section 3.4, the contexts of an action word include its enclosing segment, its potential UI elements, and the type of the target. Consequently, we tried all of the possible combination of such contexts to evaluate the effect of different parts of the contexts.

4.2.5 RQ5: Effect on Existing Approaches: To investigate to what extent *MaCa* can improve the performance of ReCDroid and Yakusu, we integrated MaCa into ReCDroid and Yakusu, respectively. The evaluation on Yakusu was conducted as follows:

- First, we applied Yakusu to dataset *icseData* that Zhao et al. [46] manually built to evaluate ReCDroid, and computed its performance (success rate).
- Second, we applied *MaCa* to dataset *icseData*. *MaCa* classified the action words in *icseData*. Based on the classification, we replaced the out-of-vocabulary action words in *icseData* with their equivalent words (i.e., words belonging to the same category) from the vocabulary of Yakusu. The updated dataset was notated as *icseData′*
- Third, we applied Yakusu to *icseData′* and computed its performance (success rate).

Notably, to evaluate the effect of MaCa on Yakusu, we leveraged dataset *icseData* instead of *newData* or *isstaData* because of the following reasons. First, *newData* may contain reports that do not results in crashes. As a result, we could not apply Yakusu to *newData* because Yakusu requires bug reports that are associated with crash issues. Second, we did not leverage *isstaData* because this dataset had been specially created for the evaluation of Yakusu, and all of the action words within this dataset had been manually added to the predefined vocabulary of Yakusu. The evaluation on ReCDroid was done similarly on dataset *isstaData* that Fazzini et al. [15] manually created to evaluate Yakusu.

4.2.6 RQ6: Compared against Static Vocabulary based Approach: An intuitive alternative to the proposed approach is to match (and classify) action words according to a pre-defined vocabulary (we call it *vocabulary-based approach*). To compare *MaCa* against vocabulary-based approach, we conducted the following evaluation:

- We partitioned the 525 bug reports in *newData* into ten groups with approximately equal size (either 52 or 53), and conducted ten-fold cross-validation on the resulting ten groups.
- On each fold, we selected one group as the testing data whereas the others were taken as training data. We built a static vocabulary of action words according to the action words (and their labels) in the training data. If a single word (appearing in different reports) belonged to different categories, we classified it into the category it most frequently belonged to. With the resulting vocabulary, we classified action words in the testing data and compared the classification against manual labeling. We also trained *MaCa* with the training data and tested it with the testing data.

Table 2: Results of Cross-Dataset Evaluation

Metrics \ Evaluation Dataset	newData (Ten-Fold)	isstaData	icseData
Accuracy	96.6%	96.7%	95.0%
Macro F1	91.4%	91.1%	89.8%

4.3 Results and Analysis

4.3.1 RQ1: MaCa *Is Accurate:* Results of the ten-fold cross validation are presented in Table 1. From this table, we make the following observations:

- First, MaCa was highly accurate. The accuracy varied from 94.2% to 99.2%, with an average of 96.6%. The high accuracy suggested that MaCa succeeded frequently in predicting action types. We also noticed that MaCa achieved a high Macro F1 score as well: the average *macro F*1 was up to 91.4%.
- Second, MaCa was stable. In different folds, the evaluation data were partitioned (into training and testing datasets) in different ways. However, the accuracy varied slightly from 94.2% to 99.2%, with a small standard deviation (σ) of 0.014. The same was true for *macro F*1 whose standard deviation was 0.031.

The results of the cross-dataset evaluation also suggested that MaCa was accurate. The results are presented in Table 2. From this table, we observe that training MaCa with *newData* and testing the resulting MaCa on *isstaData* resulted in high accuracy of 96.7% and high F1 of 91.1%. Such performance was comparable with the within-dataset ten-fold cross-validation (i.e., the performance presented in the second column of Table 2). Evaluation results on *icseData* (the last column of Table 2) also confirmed the conclusion that MaCa was accurate.

We also analyzed the performance of MaCa on different categories. On the evaluation data (notated as *newData*), there were five different categories/labels: *CLICK*, *TYPE*, *ROTATE*, *SWIPE*, and *SCROLL*. The performance on different categories was presented in Table 3. From this table, we make the following observations:

- First, MaCa was accurate for all of the categories. The accuracy varied from 92.7% (*TYPE*) to 99.4% (*SWIPE*). The minimal precision, recall, and F1 score were 87.9% (*ROTATE*), 81.9% (*SCROLL*), and 87.1% (*SCROLL*), respectively.

Table 3: Performance on Different Categories

Metrics	CLICK	TYPE	ROTATE	SWIPE	SCROLL
Accuracy	94.3%	92.7%	98.2%	99.4%	98.5%
Precision	96.1%	90.5%	87.9%	95.2%	93.1%
Recall	95.9%	93.9%	89.9%	90.9%	81.9%
F1	96.0%	92.2%	88.9%	93.0%	87.1%

- Second, MaCa achieved the best performance on *CLICK* and the lowest performance on *SCROLL*. One possible reason for the relatively low performance (especially recall) on *SCROLL* is the small size of this category. Only 32 out of the 1,202 actions belong to this category, accounting for 2.7% of the actions in the dataset. It is challenging for the employed classifier to learn effective features of the category from such a small number of training items. In contrast, *CLICK* accounts for 70.4% of the actions in the dataset, which resulted in much bigger training data (thus better performance).

We noticed that the evaluation data were not evenly distributed over different categories. 70.4% of the items in the dataset belong to *CLICK*, 21% belong to *TYPE*, 3.1% belong to *ROTATE*, 2.8% belong to *SWIPE*, and 2.7% belong to *SCROLL*. The first two categories, i.e., *CLICK* and*TYPE*, are much more popular than others. To this end, the proposed approach assigns different weights to different categories while computing loss of the model in training phase (as introduced in Section 3.4). To investigate the effect of such weights, we removed the weights and repeated the evaluation. Evaluation results suggested that the average accuracy reduced significantly from 96.6% to 86.1%, and the *macro F*1 also reduced significantly from 91.4% to 76.4%. Table 4 presents its performance on different categories, where cells in gray represent increased performance (compared to Table 3) and others represent decreased performance. By comparing this table (where weights are disabled) against Table 3 (where weights are provided), we make the following observations:

- Disabling the weights resulted in significant reduction in accuracy on all of the categories. The reduction varied from 5.6% (*CLICK*) to 20% (*ROTATE*).
- Disabling the weights also resulted in similar reduction in precision and F1 scores.
- Concerning the recall of MaCa, disabling the weights had significantly different effect on different categories. On one side, it increased recall on the largest category (*CLICK*) slightly from 95.9% to 97.1%. On the other side, it significantly reduced recall one ROTATE (from 89.9% to 33.3%), SWIPE (from 90.9% to 81.8%), and SCROLL (from 81.9% to 66.7%). For the second biggest category (TYPE) that accounted for 21% of the whole data, disabling weights did not result in significant change in recall (93.9% vs. 92.8%). One possible reason for the difference is the significant difference in their size: CLICK accounts for 70.4% of the whole data whereas ROTATE, SWIPE and SCROLL all together account for less than ten percentage of the whole data. Without weights, the classifier would be significantly bias for the dominating category (e.g., CLICK), i.e., more likely to predict items as CLICK.

Table 4: Performance of MaCa (Without Weights)

Metrics	CLICK	TYPE	ROTATE	SWIPE	SCROLL
Accuracy	88.7%	86.3%	78.2%	84.9%	92.3%
Precision	89.6%	90.5%	64.2%	89.9%	68.7%
Recall	97.1%	92.8%	33.3%	81.8%	66.7%
F1	93.2%	91.6%	43.9%	85.7%	67.7%

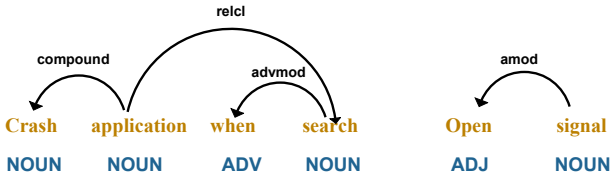

Figure 5: Missed Action Words

Based on the analysis, we conclude that providing the weights for different categories leads to significant increase of performance on smaller categories whereas the performance on the largest categories could be slightly reduced.

To reveal the reasons for failed cases, we manually analyzed the actions that MaCa failed to classify correctly. Analysis results suggest that one of the major reasons for the failure is that the employed NLP toolkit failed to identify action words. Two typical examples are presented in Fig. 5. For the example on the left, action word "*search*" was recognized as a noun. For the example on the right, the action word "*open*" was recognized as an adjective. In total, such incorrect identification of action words led to 9 out of the 40 failed cases.

Based on the preceding analysis, we conclude that the proposed approach is accurate. With improvement in natural language processing, the proposed approach could be further improved.

4.3.2 RQ2: Succeed on Unseen Words: From *newData*, we identified 20 rarest action words, removed bug records containing any of these words from *newData*, trained MaCa from scratch with the resulting shrunken dataset *newData′*, and applied MaCa to bug reports containing the 20 rarest words. The rarest words had been removed from *newData′*, and thus they were unseen words for MaCa trained on *newData′*. Such unseen words belonged to different categories: 11 CLICK, 8 TYPE, and 1 SWIPE. Eighteen of them appeared once whereas the other two appeared twice in *newData*.

Evaluation results suggested that MaCa was accurate in identifying and classifying unseen action words. On 22 steps to reproduce (s2r) containing unseen tokens, MaCa accurately identified and classified 20 action words, resulting in a high accuracy of 90.9%=20/22.

We conclude based on the preceding analysis that MaCa has the potential to identify and classify unseen action words, which is a significant advantage over predefined static vocabulary.

4.3.3 RQ3: Influence of Classification Techniques: To investigate to what extent the employed classification techniques can affect the performance of the proposed approach, we replaced the logistic regression based classifier with other common classification techniques and repeated the evaluation. Evaluation results are presented in Table 5. The first column presents different classification

Table 5: Influence of Classification Techniques

Techniques \ Dataset	newData		isstaData		icseData	
	Accuracy	Macro F1	Accuracy	Macro F1	Accuracy	Macro F1
Logistic Regression (*Default*)	96.6%	91.4%	96.7%	91.1%	95.0%	89.8%
LSTM	89.8%	81.7%	85.9%	79.4%	88.3%	81.7%
Bi-LSTM	89.1%	81.3%	89.4%	83.7%	87.4%	81.9%
MLP	86.0%	43.5%	84.1%	54.6%	82.9%	50.2%
SVM	85.0%	56.6%	82.6%	57.8%	85.6%	61.4%
Random Forest	83.4%	80.1%	83.5%	79.9%	81.7%	78.6%
Naive Bayes	80.8%	72.3%	78.8%	68.3%	79.9%	72.3%
AdaBoost	78.4%	20.6%	80.2%	33.4%	75.2%	30.3%
CNN	45.6%	37.2%	48.4%	39.7%	52.3%	44.1%

Table 6: Effect of Contexts

	Potential UI Elements	Type of Target Element	Enclosing Segment	newData		isstaData		icseData	
				Accuracy	Macro F1	Accuracy	Macro F1	Accuracy	Macro F1
1	✓	✓	✓	96.6%	91.4%	96.7%	91.1%	95.0%	89.8%
2	✓	✓		90.6%	49.3%	91.0%	60.4%	92.7%	57.7%
3	✓		✓	82.4%	29.7%	82.3%	28.5%	80.1%	29.6%
4		✓	✓	90.2%	41.3%	88.5%	59.3%	86.4%	54.8%
5	✓			77.9%	17.9%	73.4%	24.9%	75.3%	20.4%
6		✓		84.5%	36.6%	84.7%	39.4%	83.6%	38.3%
7			✓	80.8%	24.9%	80.3%	22.6%	77.6%	20.5%
8				71.4%	13.8%	71.2%	17.6%	69.3%	19.7%

techniques where the first one is logistic regression (default setting of MaCa). The other columns present performance of the proposed approach when different classification techniques were employed to replace the default setting. Notably, we evaluated the influence of classification techniques in different datasets, i.e., *newData* (ten-fold cross-validation, on the second and third columns), *isstaData* (the fourth and fifth columns), and *icseData* (the last two columns).

From Table 5, we make the following observations:

- First, the default setting (i.e., logistic regression) resulted in the best performance. On *newData*, it led to the highest accuracy (96.6%) as well as the highest *Macro F1* (91.4%). On *isstaData* and *icseData*, logistic regression also resulted in the highest performance.
- Second, although LSTM and Bi-LSTM resulted in significantly lower performance compared to logistic regression, they significantly outperformed other classification techniques, i.e., Random Forest, SVM, CNN, MLP, Naive Bayesian, and AdaBoost. On one side, LSTM and its variant Bi-LSTM are good at handling texts whereas the input of the classifiers was essential a sequence of texts. Consequently, both LSTM and Bi-LSTM led to good performance. One the other side, LSTM and Bi-LSTM often require a large number of training data that were not available for our task. As a result, they failed to outperform logistic regression that had weaker requirements on the size of training data. In future, however, if we can expand the training dataset manually or automatically, LSTM and Bi-LSTM have the potential to outperform logistic regression.

To validate the potential of LSTM and Bi-LSTM on our task, in Fig. 6 we depict the quantitative relation between size of labelled data and the performance of the proposed approach (on *newData*). From this figure, we observe that the performance increases stably with the increased data size.

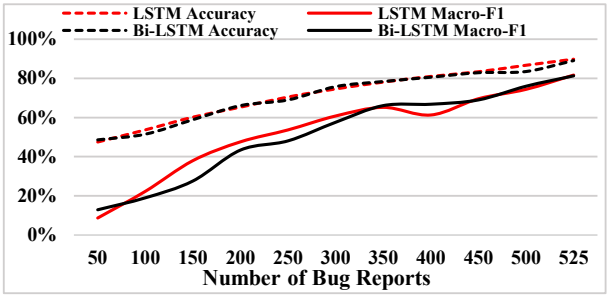

Figure 6: Potential of LSTM-based Classifiers

We conclude based on the preceding analysis that the default setting (logistic regression) outperformed alternative classification techniques for our task. However, with the increase of training data, other techniques, e.g., LSTM and Bi-LSTM, have the potential to further improve the performance.

4.3.4 RQ4: Contexts Are Critical: To reveal the effect of different contexts of the action words, we employed different combination of the contexts, i.e., the enclosing segment, the potential UI elements, and the type of target element. Results of the evaluation are presented in Table 6. Columns 2-4 presents the absence or presence of different contexts. The other columns present the performance of the proposed approach with the selected contexts on given testing data (i.e., *newData*, *isstaData*, and *icseData*, respectively). Each row

Table 7: MaCa Enhanced Existing Approaches

Approaches	#Bug Reports where the approach succeeded	Success Rate
Yakusu	26	50.98%
Yakusu+MaCa	32	62.7%
ReCDroid	22	56.41%
ReCDroid+MaCa	27	69.2%

represents a unique combination of the contexts. From the table, we make the following observations:

- Disabling any of the contexts resulted in significant reduction in performance. Compared to the default setting where all of the contexts were presented, other combinations resulted in significant reduction in both accuracy and *macro F*1.
- Type of target element was critical for the performance of the proposed approach. For example, on *newData*, disabling it decreased accuracy significantly from 96.6% to 82.4% and *macro F*1 from 91.4% to 29.7%. Disabling other contexts, i.e., the enclosing segment and the potential UI elements, resulted in smaller, but still significant, reduction in performance.
- Effect on *macro F*1 was more significant than that on accuracy (that was equal to *micro F*1). The reason is that the data were imbalanced, and thus significant changes in performance on small categories would not significantly change overall performance (e.g., accuracy). However, such significant changes in performance on small categories had obvious effect on *macro F*1 that was the arithmetic mean of *F*1 scores over different categories.

We conclude based on the preceding analysis that all of the selected contexts, i.e., the enclosing segment, the potential UI elements, and the type of target element, are useful for the classification. Disabling any of them would result in significant reduction in performance.

4.3.5 RQ5: MaCa Enhanced Existing Approaches: As an initial application of the proposed approach, we applied it to the automated validation of bug reports by integrating it into existing bug report validation approaches, i.e., Yakusu and ReCDroid. Evaluation results are presented in Table 7. The first column of the table presents the evaluated approaches. *Yakusu* and *ReCDroid* represent the original approaches proposed by Fazzini et al. [15] and Zhao et al. [46], respectively. *Yakusu + MaCa* and *ReCDroid + MaCa* represent approaches enhanced by *MaCa*. Notably, Yakusu and ReCDroid are evaluated on different datasets, and thus their denominators (i.e., the total numbers of bug reports) in calculating percentages (success rates) are different. As a result, the higher absolute number of Yakusu does not necessarily lead to a higher percentage (success rate) because Yakusu is applied to the larger dataset. As specified in Section 4.2, Yakusu and Yakusu+MaCa are evaluated on *icseData* that were created by Zhao et al [46] to evaluate ReCDroid. In contrast, ReCDroid and ReCDroid+MaCa are evaluated on isstaData that were created by Fazzini et al. [15] to evaluate Yakusu.

From Table 7, we observe that *MaCa* significantly improved the performance of both Yakusu and ReCDroid. It improved the success rate of Yakusu from 50.9% to 62.7%, with a relative increase of 22.9%=(62.7%-50.98%)/50.98%. The same was true for ReCDroid whose success rate was improved significantly by MaCa from 56.41% to 69.2%, resulting in a relative increase of 22.7%=(69.2%-56.41%)/56.41%. The evaluation results provide initial evidence of the usefulness of the proposed approach.

Table 8: Bug Reports Where MaCa Rescued Yakusu

	App	Issue No.	Out-of-vocabulary action words	Failure
1	Olam	1	tipying	No matching
2	LibreNews	22	Change	No matching
3	LibreNews	27	Change	Incorrect matching
4	ventriloid	1	left	Incorrect matching
5	Obd-Reader	22	start	No matching
6	Markor	194	–	action word "Insert" is misclassified

To further reveal the reasons for the success of MaCa, we manually analyzed the bug reports where the original approaches (i.e., Yakusu or ReCDroid) failed but the enhanced approaches (i.e., *Yakusu + MaCa* or *ReCDroid + MaCa*) succeeded. Table 8 presents the details of the six bug reports where MaCa rescued Yakusu. From this table, we observe that such bug reports come from five different apps. Five out of the six bug reports contain action words that are out of the predefined vocabulary of Yakusu (called out-of-vocabulary action words). If a s2r does not contain predefined action words (defined by the vocabulary), Yakusu leverages semantic comparison between s2r and the elements in the ontology of the relevant app to match UI elements. However, compared to the action word based heuristics, such semantic based matching is much more risky. As shown in Table 8, the semantic based matching for the six bug reports either failed to match any UI elements or resulted in incorrect matching (i.e., wrong UI elements). MaCa successfully identified such action words, classified them correctly, and replaced them with equivalent words from the given vocabulary. As a result, MaCa helped Yakusu generate test cases successfully for such bug reports. We take the third bug report of Table 8 (issue #27 of app *LibreNews*) as an example to explain how MaCa helped. The failed step was "*Change the server.*" Because the action word "*Change*" was out of vocabulary, Yakusu compared the whole segment (i.e., "*Change the server.*") against properties of ontology of the app *LibreNews*. The semantic similarity between the segment and the correct target element (notated as E2 whose ID is "*server_url*") was 0.6187. In contrast, its similarity with another element (notated as E2 whose ID was "*server_status*") was 0.6347. Consequently, Yakusu mapped the segment to wrong UI element (whose ID was "*server_status*"), which resulted in failure on this bug report. MaCa classified the action as typing, and thus Yakusu searched for editable UI elements only as the potential target of the action. As a result, element E2 (a *TextView* component) was excluded because it was not editable. In contrast, the right one, E1 (an *EditText* component) was editable and thus it was finally retrieved (correctly) as the target of the action.

Table 9 presents the details of the five bug reports where MaCa rescued ReCDroid. From the table, we also observe that four out of the five involved bug reports contain out-of-vocabulary action

Table 9: Bug Reports Where MaCa Rescued ReCDroid

	App	Issue no.	Out-of-Vocabulary Action Words	Failure
1	K-9 mail	1910	expand	timeout
2	ODK Collect	360	specify	timeout
3	Kandroid	2	makes	timeout
4	K-9 mail	2540	configuring	ignored
5	BeHe ExploreR	35	–	action word "go" is misclassified

words. As a result, ReCDroid failed to reproduce the reported crashes. We take the fourth bug report (issue #2540 of app *K-9 mail*) in Table 9 to illustrate how MaCa helped. ReCDroid failed to reproduce the following step "*Choose the inbox after configuring an account*". ReCDroid took it as a single step because it contained a single action word "*choose*" (that came from the predefined vocabulary), and thus another action "*configuring an account*" was missed. MaCa identified these two action words successfully because it decomposed the sentence into two segments according to conjunction "*after*" and identified "*configuring*" as a typing action. As a result, it helped ReCDroid reproduce both of the steps.

The last bug report on Table 9 (issue #35 of app *BeHe ExploreR*)) does not contain out-of-vocabulary action words. The bug report says "*Go on https://download.lineageos.org/. Try to download a rom. The app will crash*". The action word "*go*" was classified as CLICK in the vocabulary. However, in this case, the word represented a typing action instead of a click action: Users should type in the URL ended with an *Enter* key. As a result, ReCDroid failed to reproduce this step because of the misclassification. MaCa accurately classified the action word based on its contexts, replaced it with "*type*", and thus helped ReCDroid reproduce this step. The same is true for the last bug report on Table 8 where "*Insert*" in s2r "Insert image" was misclassified by the vocabulary as TYPE whereas MaCa classified it correctly as CLICK.

Notably, one possible reason for the effect of MaCa on Yakusu and ReCDroid is that the training data of MaCa might contain the out-of-vocabulary action words listed in Table 8 and Table 9. For example, the vocabulary of ReCDroid did not contain keyword "*expand*" (and thus it failed on the first report of Table 9) whereas this word appeared in the training data for MaCa. To investigate this issue, from the training data, we removed all of the items containing any of the out-of-vocabulary action words on Table 9. After that, we re-trained MaCa from scratch with the clean training data, and applied the resulting MaCa to the first four bug reports on Table 9. Evaluation results suggested that MaCa successfully identified and classified all of the unseen/out-of-vocabulary action words in the bug reports. For bug reports in Table 8, we conducted similar evaluation, and results also suggested that MaCa successfully identified and classified all of the out-of-vocabulary action words on Table 8 even if such words had been removed from the training data. This is one of the strengths of machine learning based approaches that they may handle action words that have never been manually labelled in the training data.

4.3.6 RQ6: Static Vocabulary based Approach: As specified in Section 4.3.6, we compared *MaCa* against the alternative approach (called *vocabulary-based approach*) on *newData*. Our evaluation results suggest that *vocabulary-based approach* was significantly less accurate than *MaCa*. Its accuracy was 72.2%, significantly lower than that (96.6%) of *MaCa*. Its F1 (53.6%) was also significantly lower than that (91.4%) of *MaCa*.

One reason for the low accuracy of *vocabulary-based approach* is that new actions words are emerging, and thus pre-defined vocabularies may miss some new words. Another reason for the low accuracy is that the same action word may belong to different categories, depending on its contexts.

4.4 Threats To Validity

The primary threat to external validity is the representativeness of the involved apps and bug reports. To reduce the threat, we crawled bug reports from Github, and reused bug reports collected by other researchers. The number (615) of involved bug reports is limited because of the complexity involved in manual analysis and labelling of the bug reports. Another reason for the limited number of bug reports is that only a small number of the publicly available bug reports explicitly specify steps to reproduce the reported issues. For example, among the 2,000 bug reports we manually analyzed, only 525 of them explicitly specify the reproduction steps.

The second threat to external validity is that only two existing approaches (i.e., Yakusu and ReCDroid) were involved to evaluate the effect of the proposed approach on automated validation of bug reports. The results could not be generalized to other approaches in automated validation of bug reports. To reduce the threat, we selected Yakusu and ReCDroid that represent the state of the art in this field. Another reason for the selection is that these two approaches came from different research groups.

A threat to constructive validity is that the manual labelling of the dataset could be inaccurate. The authors, instead of the developers who handled the reported issues, labelled the dataset because we failed to request the original developers to label the data. Without in-depth knowledge of the involved apps, however, the authors may misclassify some of the actions. To reduce the threat, the manual labelling was conducted by two authors with rich experience in app development and bug issue handling. A high Cohen's Kappa coefficient [13] of 0.7866 suggests excellent inter-rater agreement.

5 RELATED WORK

Analysis on bug reports for various tasks has been a hot research topic in the field of software engineering [15, 29, 46]. For space limitation, we only discuss such work that are most closely related to our technique.

The most closely related work is *Yakusu* [15] and ReCDroid [46]. Fazzini et al. [15] proposed an automated approach, called *Yakusu*, to translate Android bug reports into executable test cases. *Yakusu* parses textual segments into dependency trees and identifies abstract actions from the resulting trees according to a predefined vocabulary of UI actions. Finally, *Yakusu* generates executable test cases by mapping the abstract actions into a sequence of UI actions that are available for the given mobile app. Zhao et al. [46] proposed

another automated approach, called ReCDroid, to reproduce Android application crash from bug reports that are specified in natural language. Notably, our static analysis to retrieve types of target elements is inspired by the static analysis in ReCDroid [46]. ReCDroid differs from *Yakusu* in that ReCDroid extracts input values from bug reports for editable events whereas *Yakusu* generates random values for such events. Besides that, ReCDroid reproduces crashes directly whereas *Yakusu* generates test cases that could be executed (and modified if needed) to reproduce reported crashes. Both *Yakusu* and ReCDroid employ predefined (manually constructed) vocabularies of action words. As introduced in Section 1, such manually constructed vocabularies result in significant limitations in action recognition and classification. Our approach that could recognize and classify actions automatically helps such approaches to avoid predefined vocabularies. Evaluation results in Section 4 validated the effectiveness of our approach in enhancing *Yakusu* and ReCDroid.

Motwani and Brun [29] proposed an automatic approach, called Swami, to generate precise oracles and executable test cases from API specifications (i.e., ECMA-262 JavaScript specification). Different from the proposed approach, Swami takes the assumption that the textual input is structured natural language specifications. Based on this assumption, Swami identifies parts of the specification relevant to the implementation to be tested with a regular expression. The regular expression concerns the section number and the method name in the structured document. After that, Swami employs a series of regular expression-based rules to extract information of the syntax for the methods to be tested. Besides bug reports and API specification, various resources have been leveraged to generate test cases, e.g., police reports on self-driving cars [16], automatically generated error messages [44], app states [26], and log messages collected from the field [43].

Oscar and Carlos [11] proposed an approach, called EULER, to assess the quality of the steps to reproduce in bug reports. In case of ambiguous steps, steps described with unexpected vocabulary, and steps missing in the report, EULER provides actionable feedback to reporters who may improve the quality of bug reports according to the suggestions. As a preprocessing step of quality assessment, EULER identifies and classifies action words. Similar to *Yakusu* and ReCDroid, EULER employs a predefined vocabulary and determines the category of a given action word by comparing its lemma against those in the vocabulary. Our approach may further improve EULER in the same as it helps *Yakusu* and ReCDroid.

Oscar et al. [12] proposed an approach, called DEMIBuD, to identify missing information in bug descriptions. Among different implementations of DEMIBuD, DEMIBuD-ML is based on machine learning techniques. DEMIBuD-ML is essentially a linear Support Vector Machines (SVM) based binary classifier that classifies bug reports as missing or not missing expected behaviors (or steps to reproduce). S2RMiner, proposed by Zhao et al. [45] also apply machine learning techniques to bug reports. Different from DEMIBuD-ML, S2RMiner classifies sentences in bug reports as steps to reproduce (S2R) or non-S2R. Our approach is similar to DEMIBuD-ML and S2RMiner in that all of them apply machine learning based classification techniques to bug reports. However, our approach differs from DEMIBuD-ML and S2RMiner in that our approach classifies action words whereas DEMIBuD-ML and S2RMiner classify either the whole bug report or sentences in bug reports.

6 CONCLUSIONS AND FUTURE WORK

Automated validation of bug report is highly desirable. However, existing approaches rely on manually constructed vocabulary and classification of action words, which prevents such automated approaches from achieving their maximal potential. To this end, in this paper, we propose an automated approach to identify and classify action words in mobile apps' bug reports. The key insight of the approach is that the contexts of the action words are critical for the classification of action words. Consequently, the proposed approach leverages natural language processing, semantic dependency trees, and static source code analysis to identify action words as well as their contexts, i.e., the enclosing segments, potential UI elements, and types of target elements. We also manually construct a training dataset for the training and evaluation of the proposed approach. Evaluation results suggest that the proposed approach is accurate. Evaluation results on publicly available datasets suggest that the proposed approach can significantly enhanced the state-of-the-art approaches in automated validation of bug reports.

As future work, we intend to expand the manually labelled training data. To build the data, we manually analyzed only 525 bug reports. In contrast, to construct vocabularies, Zhao et al. [46] manually analyzed 813 bug reports whereas Fazzini et al. [15] analyzed 400 tutorials on mobile apps. In future, if we can analyze more bug reports to expand the dataset, we may further improve the performance of the proposed approach. We also intend to extend the proposed approach to handle bug reports of traditional GUI applications (e.g., Windows applications).

ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China under Grant No.: 61690205, 61772071

REFERENCES

[1] 2019. Bugzilla. https://www.bugzilla.org/.
[2] 2019. Flutter's bug report, no.34330. https://github.com/flutter/flutter/issues/34330.
[3] 2019. Github Issue Tracker. https://github.com/issues.
[4] 2019. Google Code Issue Tracker. https://code.google.com/archive/.
[5] 2019. LibreNew's bug report, no.22. https://github.com/milesmcc/LibreNews-Android/issues/22.
[6] 2019. Lockwise's bug report, no.783. https://github.com/mozilla-lockwise/lockwise-android/issues/783.
[7] 2019. MvvmCross's bug report, no.2532. https://github.com/MvvmCross/MvvmCross/issues/2532.
[8] 2019. spaCy. https://spacy.io/.
[9] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*. IEEE, 133–143.
[10] Dankmar Böhning. 1992. Multinomial Logistic Regression Algorithm. *Annals of the Institute of Statistical Mathematics* 44, 1 (1992), 197–200.
[11] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. Assessing the Quality of the Steps to Reproduce in Bug Reports. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 86–96.
[12] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting Missing Information in Bug Descriptions. In *Proceedings of the 2017 11th Joint Meeting*

on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017. ACM, 396–407.

[13] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. 4171–4186.

[15] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically Translating Bug Reports into Test Cases for Mobile Apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. ACM, 141–152.

[16] Alessio Gambi, Tri Huynh, and Gordon Fraser. 2019. Generating Effective Test Cases for Self-driving Cars from Police Reports. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 257–267.

[17] Jianfeng Gao, Michel Galley, and Lihong Li. 2018. Neural Approaches to Conversational AI. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL'18), Melbourne, Australia, July 15-20, 2018, Tutorial Abstracts*. 2–7.

[18] Jerzy W. Grzymala-Busse. 1991. On the Unknown Attribute Values in Learning from Examples. In *Methodologies for Intelligent Systems, 6th International Symposium, ISMIS '91, Charlotte, N.C., USA, October 16-19, 1991, Proceedings*. 368–377.

[19] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[20] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real Challenges in Mobile App Development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*. IEEE, 15–24.

[21] Liadh Kelly, Hanna Suominen, Lorraine Goeuriot, Mariana L. Neves, Evangelos Kanoulas, Dan Li, Leif Azzopardi, René Spijker, Guido Zuccon, Harrisen Scells, and João R. M. Palotti. 2019. Overview of the CLEF eHealth Evaluation Lab 2019. In *Proceedings of the 10th International Conference of the CLEF Association, CLEF2019, Lugano, Switzerland, September 9-12*. 322–339.

[22] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*. 1746–1751.

[23] Igor Kononenko. March 6-8, 1991. Semi-Naive Bayesian Classifier. In *European Working Session on Machine Learning (EWSL'91), Porto, Portuga*. 206–219.

[24] Andy Liaw, Matthew Wiener, et al. 2002. Classification and Regression by RandomForest. *R News* 2, 3 (2002), 18–22.

[25] Xiao Luo and A. Nur Zincir-Heywood. 2005. Evaluation of Two Systems on Multi-class Multi-label Document Classification. In *Foundations of Intelligent Systems, 15th International Symposium, ISMIS 2005, Saratoga Springs, NY, USA, May 25-28, 2005, Proceedings*. Springer, 161–169.

[26] Yun Ma, Yangyang Huang, Ziniu Hu, Xusheng Xiao, and Xuanzhe Liu. 2019. Paladin: Automated Generation of Reproducible Test Cases for Android Apps. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications, HotMobile 2019, Santa Cruz, CA, USA, February 27-28, 2019*. ACM, 99–104.

[27] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of 27th Annual Conference on Neural Information Processing Systems*. 3111–3119.

[28] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-completing Bug Reports for Android Applications. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, Bergamo, Italy, August 30 - September 4*. 673–686.

[29] Manish Motwani and Yuriy Brun. 2019. Automatically Generating Precise Oracles from Structured Natural Language Specifications. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE Press, 188–199.

[30] Sankar K. Pal and Sushmita Mitra. 1992. Multilayer Perceptron, Fuzzy Sets, and Classification. *IEEE Trans. Neural Networks* 3, 5 (1992), 683–697.

[31] Thanasis Petsas, Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos, and Thomas Karagiannis. 2013. Rise of the Planet of the Apps: A Systematic Study of the Mobile App Ecosystem. In *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013*. ACM, 277–290.

[32] David Martin Powers. 2011. Evaluation: from Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation. *Journal of Machine Learning Technologies* 2, 1 (2011), 37–63.

[33] J. Ross Quinlan. 1989. Unknown Attribute Values in Induction. In *Proceedings of the Sixth International Workshop on Machine Learning (ML 1989), Cornell University, Ithaca, New York, USA, June 26-27, 1989*. 164–168.

[34] Gunnar Rätsch, Takashi Onoda, and Klaus-Robert Müller. 1998. Regularizing AdaBoost. In *Advances in Neural Information Processing Systems (NIPS), Denver, Colorado, USA, November 30 - December 5*. 564–570.

[35] Siva Reddy, Danqi Chen, and Christopher D. Manning. 2019. CoQA: A Conversational Question Answering Challenge. *Transactions of the Association for Computational Linguistics* 7 (2019), 249–266.

[36] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*. IEEE, 143.

[37] Mike Schuster and Kuldip K. Paliwal. 1997. Bidirectional Recurrent Neural Networks. *IEEE Trans. Signal Processing* 45, 11 (1997), 2673–2681.

[38] scikit-Learn. 2019. API Specification on Weight of Categorie. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=class_weight.

[39] Minzhu Shen. 2020. Replication Package for Maca. https://github.com/sakura182/Maca.

[40] Johan AK Suykens and Joos Vandewalle. 1999. Least Squares Support Vector Machine Classifiers. *Neural Processing Letters* 9, 3 (1999), 293–300.

[41] Hongliang Yan, Yukang Ding, Peihua Li, Qilong Wang, Yong Xu, and Wangmeng Zuo. 2017. Mind the Class Weight Bias: Weighted Maximum Mean Discrepancy for Unsupervised Domain Adaptation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. 945–954.

[42] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE, 658–668.

[43] Tingting Yu, Tarannum S. Zaman, and Chao Wang. 2017. DESCRY: Reproducing System-level Concurrency Failures. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 694–704.

[44] Cristian Zamfir and George Candea. 2010. Execution Synthesis: A Technique for Automated Software Debugging. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*. ACM, 321–334.

[45] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. 2019. Automatically Extracting Bug Reproducing Steps from Android Bug Reports. In *Reuse in the Big Data Era - 18th International Conference on Software and Systems Reuse, ICSR 2019, Cincinnati, OH, USA, June 26-28, 2019, Proceedings*. Springer, 100–111.

[46] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE Press, 128–139.