Crowdsourced Test Report Prioritization Based on Text Classification

YUXUAN YANG, XIN CHEN
School of Computer Science, Hangzhou Dianzi University (e-mail: yangyuxuan@hdu.edu.cn, chenxin4391@hdu.edu.cn)
Corresponding author: Xin Chen (e-mail: chenxin4391@hdu.edu.cn).

This work is partially supported by the Science and Technology of Zhejiang Province (Grant No. LY21F020020), the National Natural Science Foundation of China (Grants Nos. 61902096), and Key Project of Science and Technology of Zhejiang Province (Grants No. 2017C01010).

ABSTRACT In crowdsourced testing, crowd workers from different places help developers conduct testing and submit test reports for the observed abnormal behaviors. Developers manually inspect each test report and make an initial decision for the potential bug. However, due to the poor quality, test reports are handled extremely slowly. Meanwhile, due to the limitation of resources, some test reports are not handled at all. Therefore, some researchers attempt to resolve the problem of test report prioritization and have proposed many methods. However, these methods do not consider the impact of duplicate test reports. In this paper, we focus on the problem of test report prioritization and present a new method named DivClass by combining a diversity strategy and a classification strategy. First, we leverage Natural Language Processing (NLP) techniques to preprocess crowdsourced test reports. Then, we build a similarity matrix by introducing an asymmetric similarity computation strategy. Finally, we combine the diversity strategy and the classification strategy to determine the inspection order of test reports. To validate the effectiveness of DivClass, experiments are conducted on five crowdsourced test report datasets. Experimental results show that DivClass achieves 0.8887 in terms of APFD (Average Percentage of Fault Detected) and improves the state-of-the-art technique DivRisk by 14.12% on average. The asymmetric similarity computation strategy can improve DivClass by 4.82% in terms of APFD on average. In addition, empirical results show that DivClass can greatly reduce the number of inspected test reports.

INDEX TERMS Crowdsourced testing, test report prioritization, text classification, diversity strategy, asymmetric similarity.

I. INTRODUCTION

These years have witnessed that mobile applications have become more and more important and powerful in our daily lives and work, such as transportation, shopping, and payment. However, due to the characteristics of the seamless cycle and continuous evolution, mobile applications pose great challenges to software testing activities. To meet the challenges, many companies and organizations recently adopt crowdsourced testing to detect the post-release bugs in software. Crowdsourced testing is an emerging software testing technology which is based on the concept of crowdsourcing proposed by Howe and Robinson in 2006 [1]. Different from traditional software testing, crowdsourced testing is performed by a large number of online crowd workers (who may be not professionals) from different places [2]. It can effectively reduce test cost and test cycle, and improve test efficiency [3]. By simulating the real scenario, crowdsourced testing can provide developers with real feedback information, function requirements, and user experiences [4]. Therefore, crowdsourced testing has become a widely used technology for software testing and attracted a lot of attention from industry and academic. Many crowdsourced platforms have sprung up, such as uTest1, Testin2, and TestBirds3.

In crowdsourced testing, crowd workers help developers perform testing and submit test reports recording the abnormal behaviors of software [5]. Typically, a test report is composed of four fields, including environment, input, description, and screenshot [6], which can provide some critical information for developers to understand and fix bugs. However, workers are recruited from open platforms, it is hard to guarantee their expertise about software testing.

1www.utest.com
2wwwittestin.com
3www.testbirds.com
Some workers may be unfamiliar with and unexperienced in software testing, thus the submitted test reports are usually highly redundant (i.e., many test reports reveal the same bug using different natural language information) and their quality may vary greatly [7]. Meanwhile, crowdsourced testing will produce a large number of test reports in a short time. These test reports are simultaneously submitted to developers who need to read through the content, reproduce the bug, and make a decision for debugging [8]. In such a way, due to the large number and the widely varied quality, some test reports are dealt with extremely slowly or not at all [9].

To help developers handle test reports more efficiently, researchers have conducted extensive studies on reducing the number of inspected test reports [10]. Based on the ideal of test case prioritization [11], [12], some researchers tried to leverage text-mining techniques to prioritize test reports so that developers inspect test reports in an order that reveals bugs earlier [13]. However, they did not consider the impacts of duplicate test reports. That is, duplicate test reports may be selected when determining the next test report in generating the inspection order. In the literature, Jiang et al. [4] attempted to resolve the problem of clustering crowdsourced test reports. They partitioned test reports into different clusters in which test reports belonging to the same cluster reveal the same bug. They discovered that identifying duplicate test reports can significantly reduce the number of inspected test reports. In this study, we focus on resolving the problem of test report prioritization by considering the impact of duplicate test reports.

In this paper, we propose a new method named DivClass by combining a diversity strategy and a classification strategy for test report prioritization. First, we leverage Natural Language Processing (NLP) techniques to preprocess crowdsourced test reports and extract some important words to form a keyword dictionary. Then, a vector space model is built based on the keyword dictionary. We calculate the risk value of each test report and construct a similarity matrix for all the test reports based on an asymmetric similarity computation strategy. Finally, we combine a diversity strategy and a classification strategy to prioritize test reports. On the one hand, we leverage the diversity strategy to select the test report (for inspection) who has the maximum distance with all the already inspected test reports. On the other hand, the classification strategy is applied to identify the duplicate test reports for the currently selected test report to avoid that the duplicates are selected again in the early phase.

To validate the effectiveness of DivClass, we run experiments on five collected datasets with 1728 crowdsourced test reports. We investigate four research questions and implement the widely used evaluation metric APFD (Average Percentage of Faults Detected) to evaluate the effectiveness of DivClass. The state-of-the-art method DivRisk is selected as a baseline method for comparison. Experimental results show that DivClass can achieve 0.8887 in terms of APFD and improve DivRisk by 14.12% on average. The results also demonstrate that the asymmetric similarity computation strategy can effectively improve DivClass in terms of APFD. Meanwhile, we empirically validate whether DivClass can reduce the number of inspected test reports by detecting a given percentage of bugs. Empirical results show that DivClass obviously outperforms DivRisk. When detecting 100% of bugs, DivClass can reduce the number of inspected test reports by 45.31%, 43.42%, 40.38%, 44.81%, and 85.09% compared with DivRisk on these five datasets, respectively.

In this paper, we make the following contributions:

1) To the best of our knowledge, we first apply the concept of test report classification to resolve the problem of crowdsourced test report prioritization for improving the effectiveness of existing methods.

2) We propose a new method named DivClass by combining a diversity strategy and a classification strategy for test report prioritization. Meanwhile, an asymmetric similarity computation method is adopted to overcome the impact of multi-bug test reports.

3) To evaluate the effectiveness of DivClass, we run extensive experiments on crowdsourced test report datasets. Experimental results show that DivClass has good performance in prioritizing test reports and outperforms the baseline method DivRisk.

The rest of this paper is structured as follows. Section 2 details the background and motivation for this study. The framework of DivClass is presented in Section 3. In Section 4 and Section 5, we show the experimental setup and experimental results, respectively. The discussion about the threats to validity is described in Section 6 and the related work is reviewed in Section 7. Finally, we conclude this paper in Section 8.

II. BACKGROUND AND MOTIVATION

In this section, we describe the background of crowdsourced testing and the motivation for this work.

Differing from traditional software testing, crowdsourced testing recruits not only professional testers, but also end users for testing [2]. These testers are geographically decentralized and called crowd workers. In crowdsourced testing, companies or organizations prepare the software under test and design test tasks. Then, test tasks are released in crowdsourced platforms. Workers passing an evaluation select test

FIGURE 1: The procedure of crowdsourced testing.
tasks based on their test environments and perform testing. When detecting a bug, the worker should edit a test report using descriptive natural language according to the given template. Figure 1 presents the procedure of crowdsourced testing. In our experiments, test reports are mainly defined by four parties, namely environment, input, description, and screenshot. Table 1 gives several examples of crowdsourced test reports. In the table, Environment refers to the test environment, including software and hardware configuration. Input is test data and operation steps. Workers follow the corresponding input to perform testing. Description is natural language information which records the abnormal behavior of software. Screenshot contains some pictures that may capture the states of software when occurring a bug.

Compared with traditional testing, crowdsourced testing has its own characteristics. First, to attract more workers for testing, test tasks are usually financially compensated. Many workers tend to quickly complete test tasks and submit more test reports. In such a way, the number of submitted test reports is large, obviously outperforming the available resources of developers. Thus, many test reports are not dealt with timely and some important bugs may be not detected and fixed before releasing. Second, workers tend to detect easily discovered errors in software rather than critical bugs or bugs that are hard to reproduce. Therefore, the submitted test reports are highly redundant. Inspecting duplicate test reports will spend unnecessary time and resources of developers. Third, writing long and descriptive test reports may be more challenging on mobile software than client software [5]. For convenience, some workers may report multiple bugs in the same test report which is called a multi-bug test report. As shown in Table 1, TR1 and TR2 reveal one bug, respectively. TR3 is a multi-bug test report which involves two bugs and uses the serial numbers to distinguish them.

To help developers reduce the inspection cost, researchers have conducted extensive studies on resolving the task of test report prioritization that aims to detect more bugs by inspecting fewer test reports. Two state-of-the-art techniques DivRisk [13] and Text&ImageDiv [5] are proposed. DivRisk leverages the textual content to prioritize test reports. It combines a diversity strategy which is used to determine the candidate set and a risk strategy which is applied to select the final test report from the candidate set for inspection. Text&ImageDiv combines both the textual information and the screenshot information for test report prioritization. Then, a balanced formula is designed to calculate the distances between test reports by combining the textual distance and the image distance. Although these two methods have made efforts for this task, the effectiveness is not promising, especially when detecting 100% of bugs. In addition, these methods do not exclude the impact of duplicate test reports on forming the inspection order.

In this paper, we propose a new method by combining a diversity strategy and a classification strategy to generate the inspection order for test reports. The diversity strategy is applied to determine the next test report for inspection and the classification strategy is applied to identify duplicate test reports by a propagation based classification method. Meanwhile, we take multi-bug test reports into consideration and design an asymmetric similarity computing method.

### III. METHODOLOGY

In this section, we describe the implementation details of DivClass. As shown in Figure 1, DivClass is composed of three procedures. First, we adopt NLP to preprocess crowdsourced test reports and build a keyword dictionary. Then,
an asymmetric similarity metric is constructed by employing the Jaccard similarity coefficient. Finally, we combine a
diversity strategy and a classification strategy to generate the
inspection order for test reports.

Notably, test reports contain two fields of natural language
information, namely the input and the description. By an
investigation on test reports, we observe that the input in-
formation of test reports is highly similar and the content of
input is more than that of the description. For example, in
Table 1, TR1 and TR2 contain many of the same words.
Although the description information of TR1 is obviously
different from that of TR2, their similarity is high when
combining the textual contents of both the input and the
description. In such a way, it is hard to distinguish whether
these two test reports reveal the same bug or not. Therefore,
we only leverage the description information to calculate the
similarities of test reports.

Runtime example. To clearly show the procedure of Di-
vClass, we select five crowdsourced test reports presented
in Table 1 as an example to display the execution result
corresponding to each procedure. Five test reports reveal four
different bugs. TR1 is a duplicate of TR4. TR3 is a multi-
bug test report and reveals a same bug with TR2. TR2 and
TR5 do not provide screenshots.

A. PREPROCESSING

In our experiments, test reports are composed of Chinese
with a few English words. Compared with English or other
Latin languages, Chinese is extremely different [4]. There-
fore, we need a Chinese NLP tool to process crowdsourced
test reports. Fortunately, there are many efficient Chinese
NPL tools, such as Language Technology Platform (LTP)4,
ICTCLAS5, and IKAnalyzer6. These tools have been widely
adopted in some tasks related to text processing [13], [14].
In this study, we select LTP to preprocess crowdsourced
test reports since LTP can provide online cloud services for
processing natural language and implementing simple pro-
cessing for English. The preprocessing consists of four pro-
cedures, including word segmentation, stop word removal,
synonymy replacement, and test report representation.

TABLE 2: The representation for these five test reports

| No. | Words                                      |
|-----|--------------------------------------------|
| TR1 | check, end, horizontal, pictur, screen, scroll, show, system |
| TR2 | circl, fail, friend, pictur, share, wechat |
| TR3 | applic, circl, download, fail, friend, open, pictur, recommend, share, show, system, wechat |
| TR4 | function, keyword, miss, pictur, remind, show |
| TR5 | correspond, horizont, itself, keyword, pictur, result, search |

Word Segmentation. Different from English or other
Latin languages that use spaces to segment words, Chinese
does not contain spaces. Chinese word segmentation aims to
divide a series of continuous Chinese characters into vocab-
ularies with independent semantics according to the human’s
understanding to Chinese. LTP can accurately implement
word segmentation for Chinese documents. Meanwhile, we
remove punctuation and numbers based on the segmentation
results since punctuation and numbers are useless for simi-
lariry measurement. In this study, we only retain vocabularies
consisting of Chinese characters.

Stop word removal. Test reports usually contains some
meaningless words which may produce negative impacts on
calculating the similarities of test reports. In addition, stop
words will increase the scale of features and thus need to
spend extra computing time. Therefore, we need to remove
these stop words by leveraging a Chinese stop word list7 with
1204 common words.

Synonymy replacement. In crowdsouced testing, differ-
ent workers have their own preferences to words or phrases,
they may use different expressions to describe the same bug
when editing test reports, thus the similarity computation
may be not accuracy. For example, the words “select” and
“choose” convey the same semantics. To alleviate the seman-
tic gap between test reports containing the same bug, we im-
plement the synonymy replacement operation. In this study,
we employ the thesaurus to perform synonymy replacement
based on the LTP platform.

Test report representation. Since test reports are present-
ed in an unstructured form with different free-form texts, we
need to represent them in a structured form for similarity
measurement. In this paper, we adopt the bag of words

4 http://www.ltp-cloud.com/
5 http://ictclas.nlpir.org/
6 http://www.oschina.net/p/ikanalyzer
7 https://github.com/hdudbsinlp/stopwords.
model [15] to represent test reports, which is a commonly used method for unstructured document representation. For a given test report, we represent it as a set of words, namely $TR_i = \{W_1^i, W_2^i, \ldots, W_{i_m}^i\}$, where $i_m$ represent the number of words within the $i$-th test report.

Although the preprocessing based on NLP is designed to process crowdsourced test reports written in Chinese, it is also suitable for processing documents written English or other Latin languages. By using other NLP tools, such as the Stanford NLP toolkit\(^8\), we can process documents written in English. However, different languages have their own characteristics and thus need to employ different operations. For example, English words are naturally segmented by spaces and the same word may be presented in different forms, such as “select” and “selected”. We would remove the operation word segmentation and add the operation lemmatization in the preprocessing.

**Example.** After removing the stop words, each test reports can be represented as a set of words, as shown in Table 2.

### B. SIMILARITY COMPUTATION

After the preprocessing, we need to calculate the similarities between test reports for identifying duplicate test reports. In literature, the cosine similarity [16], [17] and the Jaccard index [5] are widely applied to calculate the similarity between a pair of documents. In this study, we employ the Jaccard index to calculate the similarity between a pair of test reports. The main reason is that the Jaccard index is more suitable for calculating the similarity between two vectors which only considers whether one keyword occurs and neglects the number of its occurrences.

As mentioned above, multi-bug test reports include more natural language information while some single-bug test reports (i.e., a test report only reveals a bug) may contain several keywords. In such a way, tradition symmetrical similarity computation methods may lead to suboptimal inspection orders. The reason is that if the similarity between a multi-bug test report and a single-bug test report outperforms the similarity threshold, they are regarded duplicate test reports. Thus, when the single-bug test report is selected for inspection, the multi-bug test report containing additional bug information is identified as a duplicate. Actually, the single-bug test report may be similar with the multi-bug test report, but the multi-bug test report is not very similar to the single-bug test report. For example, $TR_3$ is a multi-bug test report which reveals the same bug with $TR_2$. $TR_2$ contains all these 6 words within $TR_2$, while 6 out of 13 words within $TR_3$ can be found in $TR_2$. Intuitively, $TR_3$ is very similar to $TR_3$, but $TR_3$ is not very similar with $TR_2$. Therefore, we introduce an asymmetric strategy for similarity measurement of test reports.

The Jaccard index is used to evaluate the difference between two finite sample sets. The greater the index value is, the higher the similarity is. In this work, each test report is regarded as a set of keywords. Given two test reports $TR_i$ and $TR_j$, the original formula for calculating the similarity is as follows:

$$Sim( TR_i, TR_j ) = \frac{| TR_i \cap TR_j |}{| TR_i \cup TR_j |} \quad (1)$$

In this work, we define that $Sim( TR_i, TR_j )$ is the similarity between $TR_i$ and $TR_j$, and $Sim( TR_i, TR_j )$ refers to the similarity between $TR_j$ and $TR_i$. Meanwhile, $Sim( TR_i, TR_j )$ is not equal to $Sim( TR_j, TR_i )$. Based on Formula 1, $Sim( TR_i, TR_j )$ is equal to $Sim( TR_j, TR_i )$. Therefore, to distinguish $Sim( TR_i, TR_j )$ and $Sim( TR_j, TR_i )$, we define the following formula:

$$Sim( TR_i, TR_j ) = \frac{| TR_i \cap TR_j |}{| TR_i |} \quad (2)$$

**Example.** With the asymmetric similarity computation strategy, we calculate the similarities between test reports, as shown in Table 3. $Sim( TR_2, TR_3 )$ and $Sim( TR_3, TR_2 )$ are 1 and 0.5, respectively. In such a way, if we set the similarity threshold to 0.7, $TR_2$ is the duplicate of $TR_3$, but $TR_3$ is not the duplicate of $TR_2$.

### C. PRIORITIZATION TECHNIQUE

In this subsection, we describe how DivClass works in prioritizing test reports.

In general, the inspection order of test reports is formed one by one. Undoubtedly, it is important to select the first test report. In the study, Feng et al. adopted a risk strategy to determine the first test report for inspection [13]. They defined the number of different keywords contained in a test report as the risk value of the test report. That is, the more keywords a test report contains, the greater the risk value is. The risk strategy dynamically reduced the risk values of keywords not related to a true bug. However, this strategy needs additional manual efforts for determining whether the test report reveals a true bug or not. In addition, multi-bug test reports generally involve more natural language information and should be inspected first since inspecting one test report can detect multiple bugs. Therefore, we also use risk values to represent the degree of risk of test reports, but do not dynamically change the risk values of keywords not related to a true bug.

**DivClass.** DivClass consist of two phases. In the first phase, given a set of $n$ crowdsourced test reports $TR = \{ TR_1, TR_2, \ldots, TR_n \}$, we first select the test report $TR_s$ with the highest risk value for inspection. Let $QTR$ be the

| Table 3: The asymmetric similarity matrix for these five test reports |

|       | $TR_1$ | $TR_2$ | $TR_3$ | $TR_4$ | $TR_5$ |
|-------|--------|--------|--------|--------|--------|
| $TR_1$| 1      | 0.1429 | 0.4286 | 0.2857 | 0.1429 |
| $TR_2$| 0.1667 | 1      | 1      | 0.1667 | 0.1667 |
| $TR_3$| 0.25   | 0.5    | 1      | 0.0833 | 0.0833 |
| $TR_4$| 0.3333 | 0.1667 | 0.1667 | 1      | 0.3333 |
| $TR_5$| 0.1429 | 0.1429 | 0.1429 | 0.2857 | 1      |

---

\(^8\)http://nlp.stanford.edu/software/
Based on the diversity strategy, we select the test report (from $CTR$) who has the maximum distance with $QTR$ as the next inspected test report. This test report is removed from the corresponding class. We select another new test report from this class and add it to $CTR$. The procedure is repeated until all the classes become empty sets.

**Example.** We calculate the risk values for these five test reports by counting the number of keywords. The risk values of $TR_1, TR_2, TR_3, TR_4,$ and $TR_5$ are 7, 6, 12, 6, and 7, respectively. Thus, $TR_3$ should be first selected for inspection. We remove $TR_3$ from $TR$ and create a class $G_1 = \{TR_3\}$. When $\delta$ is set to 0.7, $TR_2$ is identified as the duplicate of $TR_3$ based on the similarity matrix. We remove $TR_2$ from $TR$ and put it into $G_1$. Then, the diversity strategy is applied and $TR_5$ is selected for inspection. There is not a duplicate for $TR_5$. We proceed to select $TR_4$ for inspection, followed by $TR_1$. Finally, $TR_2$ is selected for inspection. The complete inspection order is $\{TR_3, TR_5, TR_4, TR_1, TR_2\}$.

**IV. EXPERIMENT SETUP**

This section details the experimental setup. First, we present the research questions (RQs). Then, the evaluation metric is introduced. Third, the details of crowdsourced test report datasets are described. Next, we clarify the selected baseline methods for comparison. Finally, we detail the experimental platform and parameter settings.

**A. RESEARCH QUESTIONS**

In this paper, we attempt to apply test report classification to assist the task of test report prioritization. We proposed a new method DivClass which combines the diversity strategy and the classification strategy to implement the prioritization. The classification strategy involves a similarity threshold to determine the duplicates of the currently selected test report for inspection. In addition, to address the problem of the inaccurate similarity computation between multi-bug test reports and single-bug test reports, we introduce the asymmetric similarity computation strategy. Therefore, we run experiments for the following targets: investigating the impacts of the similarity threshold on DivClass, evaluating the effectiveness of DivClass, exploring the role of the asymmetric similarity computation strategy, and validating whether DivClass can help developer reduce the inspection cost. In this study, the research questions includes:

- **RQ1.** How does the similarity threshold $\delta$ impact DivClass?
- **RQ2.** Can DivClass outperform some baseline methods in prioritizing test reports?
- **RQ3.** Can the asymmetric similarity computation strategy improve the effectiveness of DivClass?
- **RQ4.** Can DivClass help developers reduce the number of inspected test reports?

RQ1 is designed to reflect the impacts of the parameter $\delta$
on DivClass and determine the best parameter value for $\delta$. RQ2 is designed to evaluate the effectiveness of DivClass in test report prioritization, which can help developers choose suitable techniques. RQ3 is designed to investigate whether the asymmetric similarity computation strategy can improve the effectiveness of DivClass. RQ4 is designed to validate whether DivClass can help developers reduce the inspection cost of test reports.

**B. EVALUATION METRICS**

Test report prioritization simulates the ideal of test case prioritization which aims to rank test cases to reveal bugs earlier [18]. In evaluating the effectiveness of various test case prioritization techniques, the Average Percentage of Fault Detected (APFD) [12] is widely adopted to measure how rapidly a prioritized test suite detects defects when executing the test suite [19]. Therefore, we also employ APFD to evaluate the effectiveness of DivClass.

For each bug, the first test report revealing this bug is recorded. Based on the prioritization order generated by DivClass and the bug information revealed by test reports, we can calculate the APFG to measure the effectiveness of DivClass. For APFD, the values vary from 0 to 1. The higher the value is, the faster bug detection rate the method has. The formula for calculating the APFD is as follows:

$$\text{APFD} = 1 - \frac{T_{f1} + T_{f2} + \cdots + T_{fm}}{n \times m} + \frac{1}{2 \times n}$$

where $n$ is the number of test reports, $m$ denotes the number of bugs revealed by test reports, $T_{fi}, i = 1, 2, \ldots, m$ represents the index of the first test report revealing the $i$-th bug.

**C. BASELINE METHODS**

To validate the effectiveness of DivClass, we need to select some state-of-the-art test report prioritization techniques as baselines. Actually, test report prioritization is an important task in software maintenance. Researchers have proposed two state-of-the-art techniques for prioritizing test reports, namely DivRisk and Text&ImageDiv.

DivRisk mainly adopted a text-mining based method to prioritize test reports. The method first extracts keywords from the input information and the description information in test reports to construct a key dictionary. Then, each test report is represented by a vector and its risk value is calculated according to the keyword dictionary. Finally, the prioritization technique combining a diversity strategy and a risk strategy is adopted to prioritize test reports. The diversity strategy is applied to determine the candidate test report set, and then the risk strategy is adopted to select a test report from the candidate set for inspection.

Text&ImageDiv combines both the textual information and the screenshot information for test report prioritization. This method measures the Jaccard distance and the Chi-square distance between test reports by leveraging the text-mining based technique and the image processing technique. Then, a balanced formula is designed to calculate the distance of a pair of test reports. Compared with DivRisk, Text&ImageDiv employs two different kinds of information for test report prioritization. In some projects, Text&ImageDiv achieves better results than DivRisk.

In this study, we only select DivRisk as a baseline method to validate the effectiveness of DivClass. We do not select Text&ImageDiv since it is hard to implement. First, Text&ImageDiv needs the screenshot information, but many test reports do not contain screenshots. Second, image processing is beyond the scope of software engineering. Meanwhile, we also select the Random method and the Best method as baselines for comparison. The Random method randomly selects a test report for inspection to form the inspection order. The Best method is an ideal prioritization technique.

**D. DATASETS**

To validate the effectiveness of test reports, we run experiments on five crowdsourced test report datasets. From October 2015 to January 2016, we perform crowdsourced testing for five mobile applications, namely UBook, Justforfun, CloudMusic, SE-1800, and iShopping. The brief descriptions are presented as follows:

- **Justforfun**: which is a photo sharing mobile applications developed by Dynamic Digit. Users can share and exchange photos with others online by this application.
- **SE-1800**: which is an electrical monitoring application developed by Panneng. It can provide various monitoring solutions for electricity substations.
- **iShopping**: which is an online shopping guideline mobile application developed by Alibaba. Users can search and buy what they want by this application.
- **CloudMusic**: which is a music playing and sharing mobile application developed by Netease. Users can share their music with others by this application.
- **UBook**: which is an online education mobile application developed by New Orientation. It provides a lot of course resources for users and allows them to download.

In our experiment, we release the test tasks on the crowdsourced platform kikbug.net and recruit students from different universities to perform testing. These students have passed an evaluation. They have three years of programming experience on average and are familiar with software testing. We require that they complete crowdsourced testing within two weeks and submit test reports by a small mobile application installed in their mobile phones. All the students use descriptive natural language to describe the detected bug and are required to fill the test steps that are helpful to reproduce the bug. Sometimes, some screenshots may be uploaded by the application. Meanwhile, the application can automatically record the test environment. Finally, a test report is generated and delivered to the platform.

9https://guang.taobao.com
10http://music.163.com
11http://www.pgyer.com/y44v
12http://kikbug.net
TABLE 4: Five crowdsourced test report datasets

| Dataset    | Version | #R | #R_i | #B | R_m |
|------------|---------|----|------|----|-----|
| CloudMusic | 1.3.0   | 238| 149  | 21 | 8   |
| iShopping  | 2.5.1   | 408| 193  | 65 | 28  |
| JustForFun | 1.8.5   | 291| 61   | 25 | 55  |
| SE-1800    | 2.5.1   | 348| 141  | 32 | 35  |
| UBook      | 2.1.0   | 443| 238  | 30 | 53  |
| Totals     |         | 1728| 782 | 173 | 179 |

We collect five mobile crowdsourced test report datasets including 1728 test reports. The developers of these five applications are invited to evaluate the submitted test reports. They reproduce the bugs following the corresponding input and validate whether a test report reveal a true bug. The details are presented in Table 4. In the table, #R is the number of test reports. #B represents the number of validated bugs. #R_i and #R_m denotes the number of invalid test reports and multi-bug test reports, respectively. Due to the poor experience of workers, five datasets involves many invalid test reports. Actually, there are only 230, 207, 205, 79, and 205 valid test reports that reveal 25, 32, 65, 21, and 30 bugs, respectively, which indicates that test reports are highly redundant. The degree of redundancy (which is calculated by #R/#B) of five datasets reaches 9.20, 6.47, 3.15, 3.76, and 6.83, respectively. In this study, for a fair comparison, we do not remove these invalid test reports since developers also take time to exclude them in a real scenario. We regard that all the invalid test reports reveal the same bug.

E. EXPERIMENTAL PLATFORM AND PARAMETER SETTINGS

All the experiments are conducted with Java JDK 13.0 and complied with Eclipse 4.5.1. We run DivClass on a PC with 64-bit Win 10, a Intel(R) Core(TM) i5-7500 CPU, and a 8G memory.

DivClass involves a parameter, namely the similarity threshold $\delta$. Given the impact, we experimentally tune this parameter and determine the final value 0.7. Section V.A will present the tuning details.

V. EXPERIMENTAL RESULTS

In this section, we answer the mentioned above four RQs and present the experimental results.

A. INVESTIGATION TO RQ1

Motivation. DivClass involves a parameter, namely the similarity threshold $\delta$. It may have great impacts on identifying duplicate test reports, thus influencing the effectiveness of DivClass in prioritizing test reports. In this RQ, we mainly investigate how $\delta$ influences DivClass and attempt to seek a suitable parameter value for $\delta$ to ensure that it can be applied to different datasets.

Approach. Notably, when $\delta$ is equal to 0, the prioritization technique will transform into the technique based on the diversity strategy since all test reports will be identified as duplicates. Therefore, In this experiment, we set the tuning step to 0.1 and gradually change the value of $\delta$ from 0.1 to 1. We select two datasets, namely CloudMusic and iShopping, to tune the parameter and determine the best value. We present the experimental results of DivClass with respect to different $\delta$ on other datasets.

FIGURE 3: Results of DivClass with different $\delta$ values on CloudMusic.

FIGURE 4: Results of DivClass with different $\delta$ values on iShopping.

FIGURE 5: Results of DivClass with different $\delta$ values on JustForFun.
\[ \delta \text{ and the vertical axis represents the value of APFD. As shown from the figures, when } \delta \text{ varies from 0 to 1, DivClass achieves different results in terms of APFD. In Figure 3, with the continuous growth of } \delta, \text{ the curve falls from 0.8988 to 0.8966 and rises from 0.8966 to 0.9196. When } \delta = 0.4 \text{, the curve keeps stable and DivClass achieves the best result 0.9196 in terms of APFD. In Figure 4, the curve behaves a basically upward trend except the point } \delta = 0.7. \text{ When } \delta \text{ is set to 0.5 and 0.7, DivClass achieves the best value 0.8455 in terms of APFD. Based on the results from Figure 3 and Figure 4, we choose 0.7 as the default parameter value of } \delta. \]

As seen from Figures 5 to 7, DivClass achieves the best results in terms of APFD when } \delta \text{ is equal to 0.7. Therefore, 0.7 may be a good choice for } \delta. \text{ The curves on JustForFun and SE-1800 present the basically similar tendency with that on CloudMusic, and the curve on UBook shows an obvious upward trend when } \delta \text{ varies from 0.1 to 0.4 and keeps stable when } \delta \text{ is greater than 0.4. In addition, when } \delta \text{ is greater than 0.6, all the curves keen basically stable and achieve the best results, which indicates that the classification accuracy will be improved greatly when } \delta \text{ is greater than 0.6, thus the identified duplicated test reports for the currently inspected test report are more accurate.} \]

**Conclusion.** DivClass achieves different results in terms of APFD and can obtain good result when } \delta \text{ is set to a great value. Based on the experiment results, we choose 0.7 as the default value of } \delta. \]

**B. INVESTIGATION TO RQ2**

**Motivation.** As mentioned above, some studies focused on crowdsourced test report prioritization and proposed state-of-the-art techniques for this problem, such DivRisk [13] and Text&ImageDiv [5]. In this RQ, we try to investigate whether DivClass outperforms these comparative methods. Notably, we only select DivRisk as a baseline since Text&ImageDiv involves image processing techniques that is beyond the scope of software engineering. In addition, we also select the Random method as a baseline and compare DivClass with the Best method to investigate the gap in test report prioritization.

**Approach.** DivRisk contains two parameters, namely the increment parameter of risk value and the scale of candidate set. Based on the parameter settings of the study [13], we set these two parameter to 0.2 and 8, respectively. In addition, given that the used text of DivRisk includes the input information and the description information of test reports, this experiment uses the combination of both the input and the description to run DivRisk, we select the best result as the default result of DivRisk.

**Result.** Given the nature of the Random method, we run the Random method 20 times on each dataset and calculate the average result. Table 5 shows the comparative results of DivClass and the baseline methods. In the table, “Improvement” represents the improvement achieved by DivClass compared with DivRisk. The computation formula is \( \frac{\text{DivClass} - \text{DivRisk}}{\text{DivRisk}} \). “Gap” denotes the gap between DivClass and Best. The computation formula is \( \frac{\text{Best} - \text{DivClass}}{\text{DivClass}} \). As seen from the table, we observe that the results achieved by DivClass outperform those of DivRisk and Random, and are approximate to those of Best. For example, DivClass achieves 0.9245 and 0.8724 in terms of APFD on CloudMusic and JustForFun, and improves DivRisk by 13.64% and 33.87%, respectively. However, there exist 5.98% and 3.82% of gap compared with Best. On these five datasets, DivClass improves DivRisk by 14.12% on average. The reason may be that the asymmetric similarity computation strategy and the propagation-based classification strategy play the critical role. When adopting the asymmetric similarity measurement strategy, if a multi-bug test report is selected as the next test report for inspection, some single-bug test reports will be identified as the
duplicates. In contrast, if a single-bug test report is selected as the next test report for inspection, multi-bug test reports will not be identified as the duplicates. In such a way, multi-bug test reports containing additional bug information can proceed to be selected for inspection. When adopting the propagation-based classification strategy, test reports (revealing the same bug) with different text can be identified as the duplicates, which can effectively avoid the semantic gap between test reports and reduce the impacts of duplicate test reports. Compared with Best, there still is 6.87% of gap on average. Comparatively, DivRisk can achieves better results in terms of APFD than Random on all the datasets but JustForFun. The reason may be that the nature of randomness, Random may occasionally achieve better results in terms of APFD.

**Conclusion.** DivClass achieves better results than DivRisk and Random. The performance of DivClass approximates to that of Best. The results also demonstrate that the classification strategy can effectively improve the effectiveness of test report prioritization.

**C. INVESTIGATION TO RQ3**

**Motivation.** In this study, considering that multi-bug test reports contain more natural language information than single-bug test reports, we adopt the asymmetric similarity computation strategy to calculate the similarities between test reports. In such a way, many single-bug test reports revealing different bugs may be similar to the same multi-bug test report, but the multi-bug test report is not similar to these single-bug test reports. Actually, we do not ensure whether the the asymmetric similarity computation strategy can improve the effectiveness of test report prioritization. In this RQ, we mainly investigate the role of the asymmetric similarity computation strategy.

**Approach.** In this experiment, we adopt the symmetrical similarity computation strategy to calculate the similarities of test reports, i.e., we replace Formula 2 with Formula 1 for similarity measurement. Notably, in our method, we only change the similarity computation strategy and keep other phases unchanged. For convenience, we use DivClassWS to represent the method of DivClass with the symmetrical similarity computation strategy.

**Result.** Figure 8 shows the comparative results between DivClass and DivClassWS. As shown in the figure, we can observe that DivClass significantly outperforms DivClassWS on these five datasets. That is, when introducing the asymmetrical similarity computation strategy, DivClass improves DivClassWS by 2.67%, 3.85%, 5.67%, 5.57%, and 6.34%, respectively. The main reason may be that the asymmetric similarity computation strategy can reflect the relationship between multi-bug test reports and single-bug test reports, i.e., although a single-bug test report is very similar with a multi-bug test report, the multi-bug test report may be not similar to the single-bug test report since the multi-bug test report contains some information related to other bugs. When a multi-bug test report is selected for inspection, some similar single-bug test reports should be identified as the duplicates since these single-bug test reports do not involve other bugs. In contrast, when a single-bug test report is selected for inspection, a multi-bug test report should be not identified as a duplicate.

**Conclusion.** When introducing the asymmetric similarity computation strategy, DivClass achieves better results in terms of APFD. The experimental results also demonstrate that the asymmetric similarity computation strategy may be suitable for measuring the similarities between single-bug test reports and multi-bugs test reports.

**D. INVESTIGATION TO RQ4**

**Motivation.** Actually, in a real scenario, developers are hard to inspect all the test reports due to the limited resources and time. Based on this motivation, researchers issue the problem of test report prioritization based on the ideal of test case prioritization to reveal different bugs earlier by inspecting an optimized order of test reports. Ideally, developers only need to inspect a test report for each bug. However, it is hard to obtain the best inspection order by automated prioritization techniques. That is, sometimes developers have to inspect duplicate test reports. In this RQ, we experimentally investigate whether DivClass can help developers reduce the number of inspected test reports.

**Approach.** To validate the efficiency of DivClass, this study employs the number of inspected test reports as an evaluation metric when detecting a given number (such as 25%, 50%, 75%, and 100%) of bugs. To do that, we introduce the linear interpolation method [20] to determine the number of inspected test reports. The procedure of the linear interpolation method is detailed as follows:

Step 1. $m$ represents the number of bugs revealed by all the test reports.

Step 2. $p$ refers to the percentage of detected bugs. In this experiment, we set the percentage to 25%, 50%, 75%, and 100%, respectively.

Step 3. $Q = m \times p$ is the number of bugs corresponding to a percentage. Suppose that $\text{int}(Q)$ and $\text{frac}(Q)$ are the integer part and the fractional part of $Q$, respectively, if
TABLE 6: Five crowdsourced test report datasets

| Dataset    | p  | DivClass | DivRisk | Random | Best |
|------------|----|----------|---------|--------|------|
| CloudMusic | 25%| 12.5     | 10.25   | 17.7   | 2.25 |
|            | 50%| 31       | 34      | 58.68  | 7.5  |
|            | 75%| 44.5     | 50.5    | 114.2  | 12.25|
|            | 100%| 70 | 128     | 198.5  | 18   |
| iShopping  | 25%| 11       | 13      | 31.06  | 6    |
|            | 50%| 24       | 39      | 80.2   | 18   |
|            | 75%| 46       | 79      | 167.79 | 35   |
|            | 100%| 129 | 228     | 333.5  | 52   |
| JustForFun | 25%| 3.25     | 3.25    | 13.63  | 1.25 |
|            | 50%| 18.5     | 47.5    | 61.55  | 8    |
|            | 75%| 38.75    | 117.5   | 125.18 | 14.75|
|            | 100%| 124 | 208     | 251.15 | 21   |
| SE-1800    | 25%| 17.5     | 18      | 14.4   | 4    |
|            | 50%| 46       | 47      | 55.95  | 10   |
|            | 75%| 79       | 92      | 127.85 | 18   |
|            | 100%| 133 | 241     | 298.75 | 26   |
| UBook      | 25%| 8.25     | 8.5     | 21.18  | 2    |
|            | 50%| 17.5     | 34      | 62.65  | 7    |
|            | 75%| 23.5     | 92      | 137.28 | 14.5 |
|            | 100%| 34  | 228     | 345.9  | 22   |

where \( \frac{Q}{2} \neq 0 \), we need to leverage the linear interpolation method.

Step 4. Let \( i \) and \( j \) represent the indexes of test reports when detecting at least \( Q \) and \( Q + 1 \) bugs, respectively. The number of inspected test reports is determined by the formula \( i + (j - i) \times \frac{Q}{2} \).

In this experiment, we also select DivRisk, Random, and Best as the comparative methods.

**Result.** Table 6 presents the number of inspected test reports when detecting 25%, 50%, 75%, and 100% of bugs on these five datasets. Notably, multi-bug test reports containing multiple bugs should be inspected first. When inspecting a multi-bug test report, multiple bugs are detected. Therefore, the number of inspected test reports may be lower than that of the detected bugs. As shown in the table, Best only needs to inspect a few test reports. For example, when detecting 25%, 50%, 75%, and 100% of bugs on iShopping, the Best method only needs to inspect 6, 18, 35, and 52 test reports, respectively. Correspondingly, DivClass needs to inspect 11, 24, 46, and 129 test reports, DivRisk needs to inspect 13, 39, 79, and 228 test reports, and Random needs to inspect 31.05, 80.2, 167.79, and 333.5 test reports.

The results from Table 6 also demonstrate that DivClass outperforms DivRisk and its performance is approximate to Best. For example, DivClass needs to inspect 8.25, 17.5, 23.5, and 34 test reports on UBook, and reduces the number of inspected test reports compared with DivRisk by 2.94%, 48.53%, 74.46%, and 85.09%, respectively. In addition, we observe that DivRisk needs to inspect more test reports than DivRisk on CloudMusic when detecting 25% of bugs. The potential reason is that DivRisk can identify test reports revealing different bugs in the early stage. When the order includes most the test reports, DivRisk is hard to identify the test reports revealing different bugs. Actually, according to the results, DivClass achieves greater improvement compared with DivRisk when detecting 75% and 100% of bugs, which also demonstrates this fact.

**Conclusion.** DivClass needs to inspect fewer test reports than DivRisk and Random on most datasets when detecting 25%, 50%, 75%, and 100% of bugs. Its performance is approximate to that of Best.

VI. THREATS TO VALIDITY

This section discusses some threats to validity, including multi-bug test reports, parameter selection, natural language selection.

**Multi-bug test reports.** In this study, we do not remove multi-bug test reports. Due to that multi-bug test reports reveal multiple bugs, the obtained APFD may not really reflect the true performance of prioritization techniques. Therefore, this may produce a threat to DivClass in prioritizing test reports. However, when removing multi-bug test reports from the datasets, our method also achieves 0.8921 in terms of APFD on average and outperforms DivRisk by 16.42%. In addition, in a real scenario, there may be multi-bug test reports. This technique can simulate the real scenario to handle crowdsourced test reports. Therefore, this threat will be reduced greatly.

**Parameter selection.** In DivClass, we set a similarity threshold to determine the duplicates for the currently inspected test report. The experimental results show that DivClass achieves different results in terms of APFD with the change of the parameter value. Therefore, this may influence the application and generalization of DivClass on other datasets. However, this study has conducted an experiment to reflect the impacts and seek a suitable parameter value. The results also demonstrate that DivClass can achieve the optimal or approximate optimal results on these five datasets when using the selected parameter value. Meanwhile, the experimental results also reveal that DivClass keeps relatively stable on all the datasets when \( \delta \) is greater than 0.6. Therefore, this impact is negligible.

**Natural language selection.** In the experiment, all test reports are written in Chinese. Admittedly, Chinese is extremely different from English or other Latin languages [21]. This may produce a threat to apply our method to other natural languages. However, NLP techniques have been widely applied to various text processing tasks which may involve different natural languages. Actually, the preprocessing used in this study is not limited our task. Similarly, when adopting other NLP tools, such as the Stanford NLP toolkit, we can process documents written in English or French. Certainly, as mentioned above, for different natural languages, we need adopt specific operations. Therefore, we can decrease this threat.

VII. RELATED WORK

In this section, we review some related work, including crowdsourced testing, crowdsourced test report processing, and duplicate bug report detection.
A. CROWDSOURCED TESTING

The concept of crowdsourcing is first proposed by Howe and Robinson in 2006 [1]. It is a new solution for large-scale distributed tasks by organizing ubiquitous online individuals and machines to resolve problems [2], [22]. Due to the obvious advantages, such as cost-effectiveness, impartiality, diversity, and high device and configuration coverage [3], crowdsourcing has been successfully applied to a variety of fields.

Recently, crowdsourced testing has become a new trend in software engineering [23]. Dolstra et al. first introduced crowdsourced testing for GUI testing [24]. By running the system under test in virtual machines, they recruited crowd workers to conduct remote semi-automated testing for the system [24]. To demonstrate that oracle problems can be solved by crowdsourced testing, Pastore et al. divided the problems into a series of sub-problems and distributed these sub-problems to different workers [25]. Nebeling developed a toolkit named CrowdStudy to conduct usability testing for web sites [26]. They not only recruited a large number of workers for testing, but also designed different conditions of web site evaluations. To overcome the challenges of mobile application testing, Wu et al. proposed AppCheck which crowdsourced the event trace collection online, and captured various touch events from end users when they interacted with the app [27]. Zhang et al. provided a comprehensive guidance on crowdsourced test services for mobile applications [28]. They presented a clear comparison between crowdsourced testing and traditional testing for mobile application testing.

Besides, some studies focused on worker selection [29]. For example, Cui et al. presented a new hybrid method named ExReDiv to select a set of workers for a test task [29]. The method was composed of three strategies, including a experience strategy, a relevance strategy, and a diversity strategy. Some studies tended to improve the procedure of crowdsourced testing. For example, Alyahya and Arugebh identified a set of limitations in crowdsourced testing to improve assigning crowd manager, improving building test team, monitoring testing progress. They evaluated the process improvements by questionnaire and workshop [30]. In addition, some studies attempted to reveal the potential of crowdsourced testing to empirical studies. Liu et al. implemented the crowdsourced testing and traditional laboratory testing for software usability [31]. They found that crowdsourced testing has obvious advantages of low cost and high efficiency, but may lead to poor test results. Yang et al. tried to mine the relationship between the crowdsourced ability of students and their performance in software testing course for designing a better teaching model and improving teaching quality [32].

B. CROWDSOURCED TEST REPORT PROCESSING

Crowdsourced testing will produce a large number of test reports which take developers a lot of time to inspect. To reduce the inspection cost of test reports, researchers have conducted extensive studies to reduce the number of inspected test reports or improve the quality of inspected test reports [33]. To help developers detect more bugs by inspecting fewer test reports, Feng et al. attempted to resolve the problem of test report prioritization by the simulation of test case prioritization [13]. They combined a diversity strategy and a risk strategy to prioritize test reports. However, this method is a semi-automated method since it needs additional manual efforts. After that, considering that screenshots are important information, they integrated the text information and the screenshot information for test report prioritization [5]. Given that test reports may be highly redundant, Jiang et al. proposed the TERFUR framework to partition test reports into different clusters. Ideally, developers only need to inspect a test report from each cluster [4]. To avoid the unnecessary inspection on invalid test reports, Wang et al. tried to distinguish them from raw data by the active learning technique [6]. On the contrary, some studies focus on the quality of test reports. Taking the role of duplicate test reports into consideration, Chen et al. leveraged the additional useful information contained in duplicate test reports to augment the main test report, thus developers can understand and reproduce the bug better [8]. In addition, Gao et al. also believed that duplicate test reports play key roles in assisting developers understanding the revealed bugs [34]. To assist developers to judge the value of test reports, Chen et al. adopted a taxonomy of indicators to measure the quality of test reports by the step transformation functions [7] and the logistical regression technique [9].

Besides, some studies concentrate on assisting the generation of test reports. For example, Yu proposed a new auxiliary method named CroReG to generate crowdsourced test reports by employing image understanding techniques to analyze screenshots [35]. Liu et al. adopted computer-vision techniques to analyze the screenshots to generate descriptions for screenshots. However, this method usually required that test reports are written by professional workers [36]. In crowdsourced testing, it is hard to generate test requirements since it requires that the issuers have rich domain knowledge [37]. To do that, Guo et al. developed a tool named KARA based on the knowledge graph method [37]. KARA first analyzed the results of crowdsourced testing of Android applications and then built the knowledge graph of the target application.

C. DUPLICATE BUG REPORT DETECTION

In software development, developers tend to adopt bug tracking systems to manage bug reports. Over time, more bug reports are delivered to the bug tracking system. To help developers reduce the inspection time to bug reports, researchers have proposed various methods to resolve the problem of duplicate bug report detection [38], [39].

In existing studies, NLP techniques are one of the most widely used methods for detecting duplicate bug reports [40]. For example, Runeson et al. first leveraged NLP techniques to identify duplicate bug reports by sequentially performing...
tokenization, stemming, stop words removal, vector space representation, and similarity calculation [40]. Another important method is Information Retrieval (IR) techniques. For example, Wang et al. combined natural language and execution information for duplicate bug report detection [41]. However, this method needed to take more cost to collect the execution information. Machine learning techniques are also a popular method for duplicate bug report detection. For example, Kukkar et al. adopted Convolutional Neural Network (CNN) to extract relevant features and calculate the similarities of bug reports. Compared with traditional methods, machine learning techniques achieved significant improvement [42].

Although duplicate bug report detection is a hot research topic in software engineering, it is often not deployed [43]. Developers usually depend on the search abilities of the bug tracking systems by some simple SQL strings or IR-based words. This often leads to the creation of duplication bug reports. Hindle and Onuczko attempted to adopt continuously querying to query the bug database for searching duplicate bug reports [43]. Considering that many techniques either require additional bug information or use complex retrieval-based or learning-based methods, Chaparro et al. proposed to allow users to reformulate the queries. They presented three query reformulation strategies for users to refine the retrieval [38].

VIII. CONCLUSION
To obtain more high-quality test report prioritization orders, this study apply test report classification to the task of test report prioritization. We propose a new prioritization technique combining the diversity strategy and the classification strategy. With the diversity strategy, DivClass will select the test report with the greatest distance with already inspected test reports as the next test report for inspection. Meanwhile, with the classification strategy, DivClass can easily identify the duplicates of the currently selected test reports. Experiments are performed on five mobile crowdsourced test report datasets. Four research questions are investigated to validate the performance of DivClass. The experiment results show that DivClass has good performance in prioritizing test reports and outperforms the DivRisk technique. Our prioritization technique can effectively help developers reduce the inspection cost.

In the future, we will develop a practicable tool implementing our method for test report prioritization. We also collect more test reports to validate the effectiveness of DivClass and investigate the impact of the similarity threshold. In addition, we will attempt design new test report classification techniques and apply these techniques to resolve other tasks related to handling test reports.

IX. REFERENCES

[1] J. Howe, “The rise of crowdsourcing,” Wired magazine, vol. 14, no. 6, pp. 1–4, 2006.

[2] K. Mao, L. Capra, M. Harman, and Y. Jia, “A survey of the use of crowdsourcing in software engineering,” Journal of Systems and Software, vol. 126, pp. 57–84, 2017.

[3] F. Guarnieri and H. Muccini, “Crowd and laboratory testing, can they co-exist? an exploratory study,” in 2nd IEEE/ACM International Workshop on CrowdSourcing in Software Engineering, CSI-SE 2015, Florence, Italy, May 19, 2015, 2015, pp. 32–37.

[4] H. Jiang, X. Chen, T. He, Z. Chen, and X. Li, “Fuzzy clustering of crowdsourced test reports for apps,” ACM Trans. Internet Techn., vol. 18, no. 2, pp. 18:1–18:28, 2018.

[5] Y. Feng, J. A. Jones, Z. Chen, and C. Fang, “Multi-objective test report prioritization using image understanding,” in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, pp. 202–213.

[6] J. Wang, S. Wang, Q. Cui, and Q. Wang, “Local-based active classification of test report to assist crowdsourced testing,” in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, pp. 314–324.

[7] X. Chen, H. Jiang, X. Li, T. He, and Z. Chen, “Automated quality assessment for crowdsourced test reports of mobile applications,” in 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campanobasso, Italy, March 20-23, 2018, pp. 368–379.

[8] X. Chen, H. Jiang, Z. Chen, T. He, and L. Nie, “Automatic test report augmentation to assist crowdsourced testing,” Frontiers Comput. Sci., vol. 13, no. 5, pp. 943–959, 2019.

[9] X. Chen, H. Jiang, X. Li, L. Nie, D. Yu, T. He, and Z. Chen, “A systemic framework for crowdsourced test report quality assessment,” Empir. Softw. Eng., vol. 25, no. 2, pp. 1382–1418, 2020.

[10] S. Yu, C. Fang, Z. Cao, X. Wang, T. Li, and Z. Chen, “Prioritize crowdsourced test reports via deep screenshot understanding,” CoRR, vol. abs/2102.09747, 2021.

[11] S. Eghbali and L. Tahvildari, “Test case prioritization using lexicographical ordering,” IEEE Trans. Software Eng., vol. 42, no. 12, pp. 1178–1195, 2016.

[12] D. D. Nucci, A. Panichella, A. Zaidman, and A. D. Lucia, “A test case prioritization genetic algorithm guided by the hypervolume indicator,” IEEE Trans. Software Eng., vol. 46, no. 6, pp. 674–696, 2020.

[13] Y. Feng, Z. Chen, J. A. Jones, C. Fang, and B. Xu, “Test report prioritization to assist crowdsourced testing,” in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE’15. Bergamo, Italy: ACM, 2015, pp. 225–236.

[14] W. Zhang, T. Yoshida, X. Tang, and Q. Wang, “Text clustering using frequent items,” Knowl. Based Syst., vol. 23, no. 5, pp. 379–388, 2010.

[15] H. Liu, H. Tang, W. Xiao, Z. Guo, L. Tian, and Y. Gao, “Sequential bag-of-words model for human action classification,” CAAI Trans. Intell. Technol., vol. 1, no. 2, pp. 121–138, 2016.

[16] H. Jiang, J. Zhang, Z. Ren, and T. Zhang, “An unsupervised approach for discovering relevant tutorial fragments for apps,” in Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, pp. 38–48.

[17] H. Jiang, J. Zhang, X. Li, Z. Ren, D. Lo, X. Wu, and Z. Luo, “Recommending new features from mobile app descriptions,” ACM Trans. Softw. Eng. Methodol., vol. 28, no. 4, pp. 22:1–22:29, 2019.

[18] Q. Peng, A. Shi, and L. Zhang, “Empirically revisiting and enhancing IR-based test-case prioritization,” in ISSTA ‘20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020, 2020, pp. 324–336.

[19] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, “FAST approaches to scalable similarity-based test case prioritization,” in Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27-June 03, 2018, 2018, pp. 222-232.

[20] Y. Ledru, A. Petrenko, and S. Borodov, “Using string distances for test case prioritisation,” in ASE 2009. 24th IEEE/ACM International Conference on Automated Software Engineering, AUCKLAND, NEW ZEALAND, November 16-20, 2009, 2009, pp. 510–514.

[21] X. Zhou, X. Wan, and J. Xiao, “Cminer: Opinion extraction and summarization for chinese microblogs,” IEEE Trans. Knowl. Data Eng., vol. 28, no. 7, pp. 1650–1663, 2016.

[22] K. Mao, L. Capra, M. Harman, and Y. Jia, “A survey of the use of crowdsourcing in software engineering,” RN, vol. 15, no. 01, 2015.

[23] S. Alyahya, “Crowdsourced software testing: A systematic literature review,” Inf. Softw. Technol., vol. 127, p. 106363, 2020.
[24] E. Dolstra, R. Vliegendhart, and J. A. Pouwelse, “Crowdsourcing gui tests,” in Sixth IEEE International Conference on Software Testing, Verification and Validation, ser. ICST’13. Luxembourg: IEEE Computer Society, 2013, pp. 332–341.

[25] F. Pastore, L. Mariani, and G. Fraser, “Crowdoracles: Can the crowd solve the oracle problem?” in Sixth IEEE International Conference on Software Testing, Verification and Validation, ser. ICST’13. Luxembourg: IEEE Computer Society, 2013, pp. 342–351.

[26] M. Nebeling, M. Speicher, M. Grossniklaus, and M. C. Norrie, “Crowdsourced web site evaluation with crowdsudy,” in Proceedings of 12th International Conference on Web Engineering, ser. ICWE’12. Berlin, Germany: Springer, 2012, pp. 494–497.

[27] G. Wu, Y. Cao, W. Chen, J. Wei, H. Zhong, and T. Huang, “Appcheck: A crowdsourced testing service for android applications,” in 2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25–30, 2017, pp. 253–260.

[28] T. Zhang, J. Z. Gao, and J. Cheng, “Crowdsourced testing services for mobile apps,” in 2017 IEEE Symposium on Service-Oriented System Engineering, SOSE 2017, San Francisco, CA, USA, April 6-9, 2017, pp. 75–80.

[29] Q. Cui, J. Wang, G. Yang, M. Xie, Q. Wang, and M. Li, “Who should be selected to perform a task in crowdsourced testing?” in 41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017, Turin, Italy, July 4-8, 2017. Volume 1, 2017, pp. 75–84.

[30] S. Alyahya and D. Afrugeh, “Process improvements for crowdsourced software testing,” International Journal of Advanced Computer Science and Applications, vol. 8, no. 9, pp. 32–40, 2017.

[31] D. Liu, R. G. Bias, M. Lease, and R. Kuipers, “Crowdsourcing for usability testing,” Proceedings of the American Society for Information Science and Technology, vol. 49, no. 1, pp. 1–10, 2012.

[32] H. S. Y. H. Peng Yang, Jin Xu and J. Xu, “Crowdsourced testing ability for mobile apps: A study on mooctest,” International Journal of Performability Engineering, vol. 15, no. 11, p. 2944, 2019.

[33] S. Huang, H. Chen, Z. Hui, and Y. Liu, “A survey of the use of test report in crowdsourced testing,” in 20th IEEE International Conference on Software Quality, Reliability and Security, QRS 2020, Macau, China, December 11-14, 2020, 2020, pp. 430–441.

[34] R. Gao, Y. Wang, Y. Feng, Z. Chen, and W. E. Wong, “Successes, challenges, and rethinking - an industrial investigation on crowdsourced mobile application testing,” Empir. Softw. Eng., vol. 24, no. 2, pp. 537–561, 2019.

[35] S. Yu, “Crowdsourced report generation via bug screenshot understanding,” in 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, 2019, pp. 1277–1279.

[36] D. Liu, X. Zhang, Y. Feng, and J. A. Jones, “Generating descriptions for screenshots to assist crowdsourced testing,” in 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campanobasso, Italy, March 20-23, 2018, 2018, pp. 492–496.

[37] C. Guo, T. He, W. Yuan, Y. Guo, and R. Hao, “Crowdsourced requirements generation for automatic testing via knowledge graph,” in ISSTA ’20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020, 2020, pp. 545–548.

[38] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus, “Reformulating queries for duplicate bug report detection,” in 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24–27, 2019, 2019, pp. 218–229.

[39] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, “Duplicate bug report detection using dual-channel convolutional neural networks,” in IPC’ 20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020, 2020, pp. 117–127.

[40] P. Runeson, M. Alexandersson, and O. Nyholm, “Detection of duplicate defect reports using natural language processing,” in 29th International Conference on Software Engineering, ser. ICSE’07. Minneapolis, MN, USA: IEEE Computer Society, 2007, pp. 499–510.

[41] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, “An approach to detecting duplicate bug reports using natural language and execution information,” in 30th International Conference on Software Engineering, ser. ICSE’08. Leipzig, Germany: ACM, 2008, pp. 461–470.

[42] A. Kukkar, R. Mohana, Y. Kumar, A. Nayan, M. Bilal, and K. Kwak, “Duplicate bug report detection and classification system based on deep learning technique,” IEEE Access, vol. 8, pp. 200 749–200 763, 2020.

[43] A. Hindle and C. Onuczko, “Preventing duplicate bug reports by continuously querying bug reports,” Empir. Softw. Eng., vol. 24, no. 2, pp. 902–936, 2019.

YUXUAN YANG (M’76–SM’81–F’87) is currently pursuing the bachelor’s degree with the Hangzhou Dianzi University. His main research interests include mining software repositories.

XIN CHEN is currently a lecturer in School of Computer Science, Hangzhou Dianzi University. He has received his Ph.D. degree from Dalian University of Technology in June 2018. His main research interests include mining software repositories and search based software engineering. He is also a member of the ACM and the CCF (China Computer Federation).