Fuzzy Clustering of Crowdsourced Test Reports for Apps

HE JIANG, Dalian University of Technology
XIN CHEN, Dalian University of Technology
TIEKE HE, Nanjing University
ZHENYU CHEN, Nanjing University
XIAOCHEN LI, Dalian University of Technology

As a critical part of DevOps, testing drives seamless mobile Application (App) cycle from development to delivery. However, traditional testing is hard to cover diverse mobile phones, network environments, and operating systems, etc. Hence, many large companies crowdsource their App testing tasks to workers from open platforms. In crowdsourced testing, test reports submitted by workers may be highly redundant and their quality may vary sharply. Meanwhile, multi-bug test reports may be submitted and their root causes are hard to be diagnosed. Hence, it is a time-consuming and tedious task for developers to manually inspect these test reports. To help developers address the above challenges, we issue the new problem of FUzzy cLustering TEst Reports (FULTER). Aiming to resolve FULTER, a series of barriers need to be overcome. In this study, we propose a new framework named TEst Report Fuzzy clUstering fRamework (TERFUR) by aggregating redundant and multi-bug test reports into clusters to reduce the number of inspected test reports. First, we construct a filter to remove invalid test reports to break through the *invalid barrier*. Then, a preprocessor is built to enhance the descriptions of short test reports to break through the *uneven barrier*. Last, a two-phase merging algorithm is proposed to partition redundant and multi-bug test reports into clusters which can break through the *multi-bug barrier*. Experimental results over 1,728 test reports from five industrial Apps show that TERFUR can cluster test reports by up to 78.15% in terms of *AverageP*, 78.41% in terms of *AverageR*, and 75.82% in terms of *AverageF1* and outperform comparative methods by up to 31.69%, 33.06%, and 24.55%, respectively. In addition, the effectiveness of TERFUR is validated in prioritizing test reports for manual inspection.

CCS Concepts: •**Software and its engineering → Software libraries and repositories;** Requirements analysis;

Additional Key Words and Phrases: Crowdsourced testing, test report, fuzzy clustering, unsupervised method, duplicate detection

ACM Reference format:
He Jiang, Xin Chen, Tieke He, Zhenyu Chen, and Xiaochen Li. 2017. Fuzzy Clustering of Crowdsourced Test Reports for Apps. *ACM Trans. Internet Technol.* 9, 4, Article 39 (June 2017), 28 pages.
DOI: 0000001.0000001

This work is partially supported by the National Key Research and Development Program of China, under grant 2016YFC0800805, and the National Natural Science Foundation of China, under grant 61370144.

Author's addresses: H. Jiang, X. Chen, and X. Li, School of Software, Dalian University of Technology, Dalian-China, and Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian-China. E-mail: jianghe@dlut.edu.cn, chenxin4391@mail.dlut.edu.cn, li1989@mail.dlut.edu.cn; T. He, Z. Chen, School of Software, Nanjing University, Nanjing-China. E-mail: dg1232002@smail.nju.edu.cn, zychen@nju.edu.cn.

1 INTRODUCTION

Along with the sharp growth of mobile phones, strong business motivation pushes mobile Applications (Apps) to be delivered to market rapidly. One of the most important characteristics of mobile Apps is continuous evolution, which requires efficient communication, collaboration, and integration between development, quality assurance, and operations. Thoroughly holistically automatic DevOps is particularly suitable to these expectations and mechanisms [52]. DevOps is a new approach to drive seamless App cycle from development to delivery [1, 41]. As a critical part to promote the successful implementation of DevOps, testing can help detect and repair bugs, thus significantly improving team productivity and reliably delivering better user experience for Apps [18]. However, traditional testing, such as laboratory testing or company testing, is hard to cover diverse mobile phones (e.g., iPhone6s Plus, Samsung Galaxy S7, Huawei Mate 8), network environments (e.g., WiFi, 3G, GPRS, 4G), operating systems (e.g., Android, iOS, Blackberry, Symbian) [15, 22], etc. Therefore, many large companies or organizations tend to crowdsource their testing tasks for mobile Apps to an undefined, potentially large group of online individuals (workers) [17, 31]. As a result, crowdsourced testing based on collaboration and openness has become a new trend and attracted a lot of interests from both industry and academy [29, 38].

In crowdsourced testing, workers submit test reports for abnormal software behaviors to help developers reveal software bugs. In contrast to traditional testers (such as laboratory testers), workers have their own characteristics. On the one hand, they can provide a wide variety of testing environments, including mobile phones, network environments, and operating systems. On the other hand, workers are usually inexperienced and unfamiliar with testing activities. Those test reports submitted by workers in a short time may be highly redundant and unstructured, i.e., many test reports may detail the same bug with different free-form texts. Meanwhile, the quality (readability, reproducibility, etc.) may vary sharply among test reports. Furthermore, compared against traditional testing, crowdsourced testing has a peculiar and interesting feature, i.e., workers tend to submit multiple bugs in a single test report. Such typical multi-bug test reports generally contain more natural language information than single-bug ones, but relatively less information for each contained bug. Based on the above discussions, it is perfect if those test reports can be automatically partitioned into clusters, in which the test reports in one cluster detail the same bug. In such a way, developers only need to inspect one representative test report from every cluster, rather than all the test reports.

In the literature, researchers have investigated some similar tasks for reducing the cost of manual inspection for software artifacts. Some researchers propose many methods for crash report bucketing [3, 34, 39]. These methods leverage stack traces to extract features and calculate the similarities between crash reports, then assign them to corresponding buckets to form bug reports. In contrast to crowdsourced test reports, crash reports are usually automatically produced in structured forms and collected by software systems. Some researchers focus on automatically duplicate bug report detection [36, 43, 51] in a repository of bug reports for open source software. Natural Language Processing (NLP) [43] and Information Retrieval (IR) [46] techniques are the most common methods to resolve this problem. They calculate the similarity by either constructing vector space model or extracting features, then recommend a list of the most relevant historical bug reports for a new one. Compared against crowdsourced test reports for Apps, bug reports are usually submitted by professional testers.

In this study, we issue the new problem of FUzzy cLustering TEst Reports (FULTER) and present our attempts towards resolving FULTER, such that the test reports in one cluster detail the same bug, meanwhile multi-bug test reports can be partitioned into multiple clusters. To resolve FULTER, we need to overcome a series of barriers as follows:

- *Invalid barrier:* The crowdsourced datasets include many false positive test reports and null test reports containing no information of bugs. For example, some test reports submitted by workers describe correct system behaviors to demonstrate that test cases are successfully executed.
- *Uneven barrier:* Since test reports are written by workers in different descriptive terms, an extreme gap exists in the length of the contents among test reports. For example, some test reports only contain several terms, which do not provide enough information to analyze what cause these bugs. In contrast, some contain many sentences, which detail the steps for reproducing bugs.
- *Multi-bug barrier:* As to our observation on some industrial Apps, some workers may report multiple bugs (2-3 bugs in general) in one test report. For example, a multi-bug test report contains three functional defects, namely a downloading bug, a sharing bug, and a picture-viewing bug. Thus, it should be simultaneously partitioned into several clusters.

FULTER can be viewed as a special form of the fuzzy document clustering, in which test reports (namely documents) are assigned to clusters and a multi-bug test report can be deterministically partitioned into multiple clusters. Although fuzzy document clustering has been investigated for several decades [28], no method can be adopted to directly partition multi-bug test reports into clusters. To the best of our knowledge, Fuzzy C-Means (FCM) is the most popular fuzzy document clustering method with easy implementation and extensive adaptability [40]. However, similar as fuzzy hierarchical clustering algorithms [42], FCM needs the prior knowledge about "the number of clusters". Fuzzy k-means is adopted to the problem with unknown number of clusters, but does not take contextual information into consideration [6]. Fuzzy k-Medoids [24] highly depends on the initialization and has poor robustness. In addition, some other methods are proposed for fuzzy document clustering. They either specialize the granularity of clustering [45] or are strictly restricted in specific domains [48][9], which makes them difficult to be generalized. Therefore, such aforementioned fuzzy clustering methods cannot work well to cluster test reports.

In order to effectively resolve the new problem of FULTER, we construct a TEst Report Fuzzy clUstering fRamework (TERFUR) consisting of three components. First, two heuristic rules, namely null rule and regular rule, are well designed to automatically remove the invalid test reports to break through the *invalid barrier*. The second component proceeds to preprocess test reports by NLP and selectively enhance the description of a test report by its input to break through the *uneven barrier*. To break through the *multi-bug barrier*, the third component uses a two-phase merging algorithm to automate the fuzzy clustering so as to partition test reports into clusters, meanwhile multi-bug test reports are partitioned into multiple clusters. In such a way, we hope to help developers reduce the cost of manual inspection.

To show the effectiveness of TERFUR, we collect 1,728 test reports in total from five industrial Apps. We invite three experienced graduate students from the School of Software at Dalian University of Technology to independently complete the annotation of redundant, invalid, and multi-bug test reports and the annotation results are validated by developers of Apps to determine the standard datasets. We investigate five Research Questions (RQs) and select FCM as the baseline for comparisons. We employ microAverage Precision (*AverageP*), microAverage Recall (*AverageR*), and microAverage F1-measure (*AverageF1*) to evaluate the performance of TERFUR on the five Apps. Experimental results show that TERFUR can cluster test reports in terms of *AverageP*, *AverageR*, and *AverageF1* by up to 78.15%, 78.41%, and 75.82% and outperform comparative methods by up to 31.69%, 33.06%, and 24.55%. Finally, we leverage DivRisk [13], a prioritization technique combining a diversity strategy and a risk strategy for predicting the priority of test reports, to generate the recommendation sequence of clustering by prioritizing test reports and experimentally verify whether TERFUR can reduce manual efforts. The experimental results show that TERFUR can greatly reduce the number of inspected test reports for developers by up to 148 when all the bugs are detected.

The main contributions of this study are as follows:

(1) To the best of our knowledge, this is the first work aiming to investigate multi-bug test reports and resolving the new problem of FULTER by leveraging fuzzy clustering.
(2) We propose a new framework, namely TERFUR, to automate the fuzzy clustering for crowdsourced test reports. TERFUR introduces a filter, a preprocessor, and a two-phrase merging component to break through the *invalid barrier*, the *uneven barrier*, and the *multi-bug barrier*, respectively.
(3) We evaluate TERFUR over five industrial Apps. The experimental results show that TERFUR can cluster test reports with higher accuracy and recommend more representative test reports to developers than baseline methods.

The rest of this paper is structured as follows. Section 2 shows the background and motivation for our study. In Section 3, we detail the structure and characteristics of the five datasets. Our proposed framework TERFUR is presented in Section 4. In Section 5, we present the experimental design and summarize the experimental results. Section 6 and Section 7 discuss the threats to validity and the related work. Finally, we conclude this paper in Section 8.

2 BACKGROUND AND MOTIVATION

In this section, we detail the background of crowdsourced testing which motivates us to explore new methods for FULTER.

Software testing is a time-consuming and labour-intensive activity in software engineering. Compared against traditional software testing, crowdsourced testing recruits not only professional testers, but also end users for testing [31]. Fig. 1 shows the whole procedure of crowdsourced testing. Our industrial partners are responsible for preparing packages for crowdsourced testing: software under test and testing tasks. Then we release testing tasks online, recruit workers, and evaluate them. Workers choose testing tasks according to their own testing devices and environments, perform testing, and submit test reports for abnormal system behaviors in a prescribed time [13]. These test reports are written in natural language and delivered to a database via a tool installed on mobile phones, sometimes attached with some screenshots which capture system status when the bugs occur. Each test report can be represented with a quadri-tuple $\{\mathbf{E}, \mathbf{I}, \mathbf{O}, \mathbf{D}\}$ as follows:

- **E:** test environment, including hardware and software configuration, etc.
- **I:** test input, including input data and operation steps.
- **O:** test output, including some screenshots for arisen bug.
- **D:** test result description, namely natural language information for understanding bugs.

In the crowdsourced market, it is hard to recruit a large number of experienced workers. Even if workers pass the evaluation, their performance is still hard to guarantee [8], which may affect the quality of submitted test reports, e.g., readability and reproducibility. Moreover, since workers prefer to revealing simple bugs rather than complex bugs, a mass of highly redundant test reports revealing the same bugs may be submitted by different workers in a short time. Workers are usually paid for per testing task, they may quickly complete testing and submit test reports without adequate information, even invalid test reports. Furthermore, some test reports revealing multiple bugs may be submitted. These multi-bug test reports commonly carry more natural language information than single-bug ones, but relatively less information for each contained bug. Due to the above characteristics in test reports, it is a time-consuming and tedious task for developers to manually inspect these crowdsourced test reports, one by one.

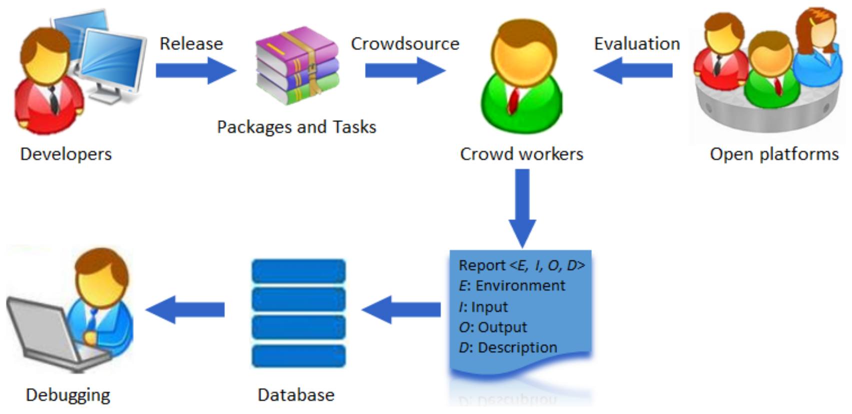


Fig. 1. The whole procedure of the crowdsourced testing for Apps.

From October 2015, we perform five crowdsourced testing tasks for industrial Apps, including Justforfun, SE-800, iShopping, MusicCloud, and UBook through our crowdsourced android testing platform, i.e., the kikbug.net [1]. In this stage of experiment, the workers are mostly students who have already acquired some knowledge of software testing and have completed tasks recorded in our platform, i.e., they are qualified for the testing task. For all these testing sessions, the testing time is two weeks. We collect 1,728 crowdsourced test reports from five industrial Apps. By gaining an insight into the results of manual annotation, most of test reports reveal simple bugs and only dozens of bugs are revealed for each App. 781 out of 1,728 test reports are labeled as invalid ones, i.e., these test reports either describe correct system behaviors or contain no description information of bugs. 184 out of 1,728 test reports are labeled as multi-bug ones, in which more than one bug are revealed. By an investigation of test reports and informal communication with our industrial partners, we discover the following findings:

(1) The quantity of crowdsourced test reports is usually a large number with high redundancy, meanwhile invalid test reports containing no bug information may be frequently submitted. Thus it will be a waste of time for developers to analyze and handle redundant and invalid test reports.
(2) In order to complete the test tasks quickly or pursue benefits, workers may report some simple bugs or submit short test reports. These test reports are either insignificant to improve the quality of Apps or hard to diagnose their root causes.
(3) Many multi-bug test reports are submitted by workers. Their contents are usually longer than those of single-bug test reports, but the description for each contained bug may be insufficient for understanding.

Based on the above mentioned findings, we issue the new problem of FULTER. In this study, a test report can be regarded as a document and a bug can be regarded as a topic or scheme. Multi-bug test reports naturally contains multiple topics or schemes. Hence, FULTER can be viewed as a special form of the fuzzy document clustering. At first, let $TR = \{TR_1, TR_2, \ldots, TR_m\}$ denote a set of test reports. Based on [24, 45], the task of FULTER is to partition these test reports into a set of clusters $G = \{G_1, G_2, \ldots, G_c\}$ according to their topics or schemes and multi-bug test reports can belong to more than one cluster, where c is the number of clusters. We try to explore an automated method to help

[1]http://kikbug.net

Table 1. Examples of crowdsourced test report

No.	Environment	Input (test steps)	Description	Output
TR_1	Operating System: Android 4.4.2 Phone: Xiaomi Redmi note System Language: Chinese Screen Resolution: 4.6 inch	1. Setting is normal. 2. The recommendation feature is available. 3. The comment feature is available. 4. The feedback message can be viewed.		
TR_2	Operating System: Android 4.4.2 Phone: Xiaomi Redmi System Language: Chinese Screen Resolution: 5.0 inch	1. Click on the button on the bottom. 2. Choose a picture and share it to friends. 3. Click on the feedback button. 4. Input advices for the app. 5. Click on the submit button. 6. Check the submitted advices.	No bug.	
TR_3	Operating System: Android 4.4.2 Phone: GiONEE GN9000 System Language: Chinese Screen Resolution: 4.6 inch	It is failed to share pictures to friends or circle of friends by WeChat.	Sharing	
TR_4	Operating System: Android 5.1 Phone: Xiaomi M1 note System Language: Chinese Screen Resolution: 4.6 inch	Check the setting and sharing feature.	Share pictures to friends or circle of friends by WeChat or micro-blogs, the system shows "operation is running in the background" or "sharing failed".	
TR_5	Operating System: Android 4.2.2 Phone: Samsung GT-18558 System Language: Chinese Screen Resolution: 3.9 inch	Complete testing and find a bug.	When the downloading is completed, the system recommends no an application for users to open the downloaded pictures.	
TR_6	Operating System: Android 4.2.2 Phone: HUAWEI G6-U00 System Language: Chinese Screen Resolution: 4.6 inch	1. Click on the recommendation button on the bottom. 2. Select a theme to view pictures. 3. Vertically scroll the screen to check the feature of "no more pictures". 4. Horizontally scroll the screen to check the feature of "no more pictures". 5. Download pictures and share pictures.	1. Horizontally scroll the screen to the end, the system does not remind "No more picture". 2. After the completion of the downloading, the system does not remind users to choose which application to open the downloaded pictures. 3. Sharing problem.	
TR_7	Operating System: Android 4.2.2 Phone: Xiaomi MI 4LTE System Language: Chinese Screen Resolution: 4.6 inch	1. Click on the category list. 2. Select a category and enter it to view pictures. 3. Horizontally scroll the screen to check the feature of "no more pictures". 4. Download pictures and share pictures.	1. When the pictures are downloaded, the system recommends no applications for opening the downloaded pictures. 2. When sharing pictures to friends or circle of friends by WeChat, the system presents the message"failed".	
TR_8	Operating System: Android 5.1 Phone: Meizu MX 5 System Language: Chinese Screen Resolution: 4.6 inch	1. Search for a topic you are interested in. 2. Click on the recommendation list and select a category to view pictures. 3. Horizontally scroll the screen to check the feature of "no more pictures". 4. Download pictures and share pictures.	Horizontally scroll the screen to the end to check pictures, the system does not show "no more pictures".	
TR_9	Operating System: Android 5.1 Phone: Meizu MX 5 System Language: Chinese Screen Resolution: 4.6 inch	1. Search for a topic you are interested in. 2. Click on the recommendation list and select a category to view pictures. 3. Horizontally scroll the screen to check the feature of "no more pictures". 4. Download pictures and share pictures.	When completing the downloading, the system does not remind users how to open the pictures.	
TR_{10}	Operating System: Android 4.4.2 Phone: Xiaomi MI 4 System Language: Chinese Screen Resolution: 5.0 inch	When scrolling horizontally the screen to the end to view the pictures, the system does not remind "no more pictures".	No more pictures.	

our industrial partners resolve FULTER, and recommend representative and informative test reports for them.

3 TEST REPORT DATASETS

In this section, we describe test report datasets and data annotation in detail.

3.1 Test Report Dataset

From October 2015 to January 2016, we perform five crowdsourced testing tasks for Justforfun, SE-1800, iShopping, CloudMusic, and UBook with five companies. Five test report datasets are collected from workers. The brief introductions for the five Apps are presented as follows:

- *Justforfun:* an interesting photo sharing App. Users can share and exchange photos with others online using Justforfun.
- *SE-1800:* an electrical monitoring App developed by Intelligent & Technology Co. Ltd. It can provide a complete and mature monitoring solution for all sizes of electricity substations.
- *iShopping:* an online shopping guideline App developed by Alibaba. Users can buy what they want online.
- *CloudMusic:* a music playing and sharing App developed by Netease. Users can build their own homepages and share their music with others.
- *UBook:* an online education App developed by New Orientation. It contains massive course resources and allows users to download them.

Workers from the kikbug platform are recruited to perform testing and submit test reports for five Apps. Those test reports are collected and transferred to the database. Table 1 presents some examples of crowdsourced test reports. Please note that all test reports are written in Chinese. For easy understanding, we translate them into English. The first column is testing environment. The following one shows the input, namely test steps, and workers perform testing according to these steps. The description is detailed in the third column. The final column presents some screenshots when bugs occur, for example, the screenshot of TR_9 implies that the prompting function does not work. In general, a test report contains two texts of natural language, namely the input and the description. However, some workers do not exactly write test reports in accordance with the guideline. They may detail their work (e.g., TR_4 and TR_5 in Table 1) or report software bugs (e.g., TR_3 and TR_{10} in Table 1) in the inputs, thus resulting in some special test reports in datasets:

Null test reports: The descriptions of test reports contain no information, only some reproduction steps are listed in the input. For example, Table 1 enumerates a null test report TR_1, of which both the description and the output are null. Hence, it is hard to identify bugs from either the environment or the input.

False positive test reports: The descriptions of test reports may declare that test cases (namely test steps in crowdsourced testing) are executed successfully. These test reports are submitted to demonstrate that workers have completed the testing tasks and the App works well. A false positive test report TR_2 is illustrated in Table 1, "No bug" hints that the test case is executed successfully.

Utmost short test reports: Some test reports only contain one or several terms which cannot provide enough information, thus resulting in poor readability for inspection. In Table 1, the description of TR_3 only contains one term "sharing". Compared against another test report TR_4 with enough natural language information, it is hard to distinguish the root causes for TR_3. However, further explanation of this bug is given in the input.

Multi-bug test reports: Workers may report multiple bugs in one test report. These test reports generally contain much natural language information, but the information for each contained bug may be insufficient. Two multi-bug test reports TR_6 and TR_7 are given in Table 1. As to TR_7, lines 1 to 3 elaborate that the prompting function does not work when scrolling horizontally the screen to the end, lines 4 to 7 detail that the App does not remind how to open the downloaded pictures, line 8 briefs a sharing problem.

Table 2. The annotation results of five test report datasets

	Justforfun	SE-1800	iShopping	CloudMusic	UBook
#Report	291	348	408	238	443
#Validated bug	25	32	65	21	30
#Invalid Report	61(20.96%)	146(41.95%)	193(47.30%)	149(62.61%)	238(53.72%)
#Multi-bug Report	55(18.90%)	36(10.34%)	28(6.86%)	8(3.36%)	57(12.87%)

#Report: The number of test reports collected by the database.
#Validated bug: The number of bugs validated by developers.
#Invalid Report: The number of test reports validated by developers as invalid ones.
#Multi-bug Report: The number of test reports revealing multiple bugs validated by developers.

Table 3. The number of multi-bug test reports revealing different numbers of bugs

#Bug	Justforfun	SE-1800	iShopping	CloudMusic	UBook
2	38(69.09%)	34(94.44%)	19(67.86%)	6(75%)	52(91.23%)
3	12(21.82%)	2(5.56%)	7(25%)	2(25%)	2(3.51%)
>3	5(9.09%)	0(0%)	2(7.14%)	0(0%)	3(5.26%)

In summary, invalid test reports including null and false positive test reports may impose negative impacts to fuzzy clustering, thus resulting in the *invalid barrier*. Utmost short test reports have low similarities with relatively long ones, thus resulting in the *uneven barrier*. Bugs in multi-bug test reports are hard to be distinguished and partitioned into correspondingly potential clusters, thus resulting in the *multi-bug barrier*.

3.2 Data Annotation

Since no dataset with annotated test reports for experiments is available and developers have no enough time to annotate the large number of test reports, three graduate students from the School of Software at Dalian University of Technology are invited to annotate the redundant, invalid, and multi-bug test reports. On average, the students have six years of programming experience and some experience of working with test reports. We distribute five datasets to three students, everyone independently completes the annotation of datasets. When inspecting a test report, students first determine whether it is a valid or invalid, single-bug or multi-bug test report. When detecting a new bug, they need to record it in summary information and create a new cluster. If a test report revealing the same bug is inspected, it is put into this cluster. When more than two students consider that two test reports describe the same bug, we put them into the same cluster. For the ambiguous test reports without consistent agreement, we report them to developers of Apps to make the final judgment. In this study, students take more than a week to annotate these test reports and developers take several hours to make the final judgments.

We totally collect more than 1,700 test reports, including 291 test reports on Justforfun, 348 test reports on SE-1800, 408 test reports on iShopping, 238 test reports on CloudMusic, and 443 test reports on UBook. Tables 2 and 3 show the results of data annotation. In the test report datasets, invalid test reports including null and false positive test reports account for 20.96%-62.61%, multi-bug test reports account for 3.36%-18.90%. Most of multi-bug test reports reveal two or three bugs, only 9.09%, 0%, 7.14%, 0%, and 5.26% of multi-bug test reports reveal more than three bugs in five Apps, respectively.

4 FUZZY CLUSTERING FRAMEWORK

In this section, we detail TERFUR consisting of three components, as shown in Fig. 2. The first component constructs a filter in which the null rule and the regular rule are proposed to filter out invalid test reports. The second one is a preprocessor which uses NLP to process the crowdsourced test reports and selectively

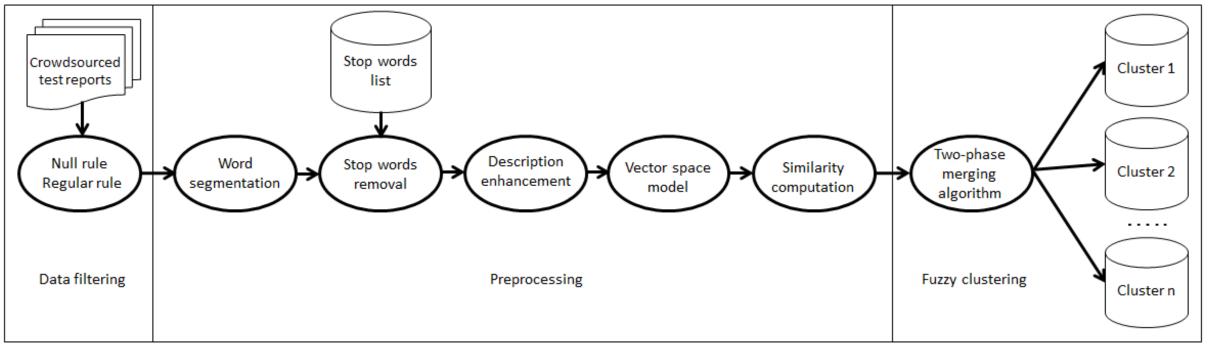


Fig. 2. TERFUR framework.

enhance the descriptions with the inputs. The third component uses a two-phase merging algorithm to implement fuzzy clustering for test reports.

Running Example. In order to facilitate understanding, ten test reports are chosen to illustrate how TERFUR works, as shown in Table 1. TR_1 and TR_2 are invalid test reports. TR_3 and TR_{10} are utmost short test reports and TR_4 reveals the same bug of TR_3. The inputs of TR_3 and TR_5 list no test steps. TR_6 and TR_7 are multi-bug test reports. TR_8 and TR_9 revealing different bugs are submitted by the same worker.

4.1 Data Filtering

In practice, invalid test reports contain no information of bugs in the descriptions and should not receive further attention from developers. Hence, invalid test reports including null and false positive test reports should be removed before the preprocessing to break through the *invalid barrier*.

On one hand, null test repots can be discarded straightforwardly since the descriptions are null. On the other hand, the descriptions of false positive test reports are written basically in declarative sentences with very simple structure, thus these test reports can be easily identified and filtered out using heuristic rules. We randomly sample 15% of false positive test reports from each dataset. By an investigation and analysis to the sample, their descriptions usually contain some special strings which are used to describe the normal system behaviors, e.g., “test pass”, “test success”, “implementation success”, “no bug”, “not detect a bug”, and “not detect any bug”. Obviously, these strings can be divided into two categories, namely affirmative strings and negative strings. All relevant words are gathered in despite of their frequencies. Based on the Chinese grammatical style and our experience, we classify these words into six types and define them as *behavior (B)*, *positive(P)*, *negative (N)*, *defect (D)*, *quantifier (Q)*, and *object (O)*, respectively. Detailed information about the two categories of strings and six types of words are shown in Table 4. Obviously, we can design a regular rule to remove false positive test reports.

To process this problem, a test report (the description) is mapped into a string which is a sequence of words. We define two rules as follows:

- *Null rule: Null.* If the string is null, we filter out the test report.
- *Regular rule: ([B][P])|([N][D]?[Q]?[O]).* If the string contains a substring matching this regular rule, we filter out this test report.

Where B, P, N, D, Q, and O are sets with a certain number of words. Table 4 lists some words contained in each set. The regular rule is composed of two parts, *([B][P])* and *([N][D]?[Q]?[O])*, which

Table 4. The detailed description for features

Feature	Category	Explanation	Examples
String category	Affirmative string	An illustration for test cases being executed successfully	Test pass. Test success.
	Negative string	An illustration for no bug existing in the Ap-p	NO bug. Not detect a bug. Not detect any bug.
Word type	Behavior	Behavior description	Test, implementation, execution
	Positive	Positive description	Pass, success, normal
	Negative	Negative description	No, not, none, never
	Detect	Action description	Detect, find, discover
	Quantifier	Number description	Any, some, a, an, one
	Object	Object description	Bug, defect, fault, error, problem

symbolize an affirmative pattern and a negative pattern, respectively. In the substring generated by the regular rule, each type of words occurs at most once. For example, "No problem" has no D and Q types of words

Examples. TR_1 is a null test report and TR_2 is a false positive test report. They match the null rule and the regular rule, respectively. The two test reports are discarded and TR_3, TR_4, TR_5, TR_6, TR_7, TR_8, TR_9, and TR_{10} are remained.

4.2 Preprocessing

In our datasets, test reports are mostly composed of Chinese characters and extremely few English words. The preprocessing component consists of five steps: word segmentation, stop words removal, description enhancement, vector space model, and similarity computation.

Word segmentation: In contrast to English words with natural spaces, we need a tool for Chinese segmentation. Many efficient tools have been developed for word segmentation of Chinese documents, e.g. ICTCLAS [2] and IKAnalyzer [3], which have been widely adopted by existing studies [13, 58]. In our study, we adopt IKAnalyzer, since it is an open-source Chinese NLP tool and can be used for English word segmentation as well.

Stop words removal: This step removes stop words which are considered to be unhelpful for similarity computation. The used stop word list contains 1208 common words [4] as well as some serial numbers consisting of numbers and symbols.

Example. We implement word segmentation and stop words removal for both the inputs and the descriptions, together with stemming [5] in view of English processing. The results are shown in Table 5. After the two steps, the descriptions of TR_3 and TR_{10} only contain a word.

Description enhancement: A typical test report contains two parts of nature language information, namely the input and the description, which can provide potential information to help developers discriminate bugs and diagnose their root causes. Due to the poor performance of workers, the inputs of some test reports may contain mixed information, e.g., test steps, testing details, and bug information. In addition, sometimes the descriptions containing only several words have a great effect on the similarity computation. Hence, we try to selectively enhance the descriptions with the inputs in the preprocessing and leverage the enhanced descriptions to calculate similarities. To do so, we adopt the number of words

[2]http://ictclas.nlpir.org/
[3]http://www.oschina.net/p/ikanalyzer
[4]http://www.oscar-lab.org/chn/resource.htm
[5]https://github.com/kristopolous/Porter-Stemmer

Table 5. The results after word segmentation and stop words removal

No.	Input	Description
TR_3	fail, share, pictur, friend, circl, friend, wechat	share
TR_4	check, set, share, featur	share, pictur, friend, circl, friend, wechat, micro, blog, system, show, oper, run, background, share, fail
TR_5	complet, test, find, bug	download, complet, system, recommend, applic, user, open, download, pictur
TR_6	click, recommend, button, bottom, select, theme, view, pictur, vertic, scroll, screen, check, featur, pictur, horizont, scroll, screen, check, featur, pictur, download, pictur, share, pictur	horizont, scroll, screen, end, system, remind, pictur, complet, download, system, remind, user, choos, applic, open, download, pictur, share, problem
TR_7	click, categori, list, select, categori, enter, view, pictur, horizont, scroll, screen, check, featur, pictur, download, pictur, share, pictur	pictur, download, system, recommend, applic, open, download, pictur, share, pictur, friend, circl, friend, wechat, system, present, messag, fail
TR_8	search, topic, interest, click, recommend, list, select, categori, view, pictur, horizont, scroll, screen, check, featur, pictur, download, pictur, share, pictur	horizont, scroll, screen, end, check, pictur, system, show, pictur
TR_9	search, topic, interest, click, recommend, list, select, categori, view, pictur, horizont, scroll, screen, check, featur, pictur, download, pictur, share, pictur	complet, download, system, remind, user, open, pictur
TR_{10}	scroll, horizont, screen, end, view, pictur, system, remind, pictur	pictur

Table 6. The results after implementing the description enhancement strategy

No.	Enhanced description
TR_3	fail, share, pictur, friend, circl, friend, wechat
TR_4	share, pictur, friend, circl, friend, wechat, micro, blog, system, show, oper, run, background, share, fail
TR_5	download, complet, system, recommend, applic, user, open, download, pictur
TR_6	horizont, scroll, screen, end, system, remind, pictur, complet, download, system, remind, user, choos, applic, open, download, pictur, share, problem
TR_7	pictur, download, system, recommend, applic, open, download, pictur, share, pictur, friend, circl, friend, wechat, system, present, messag, fail
TR_8	horizont, scroll, screen, end, check, pictur, system, show, pictur
TR_9	complet, download, system, remind, user, open, pictur
TR_{10}	scroll, horizont, screen, end, view, pictur, system, remind, pictur

as the metric. Empirically, when the number of words within the description is less than both that of the input and four words, the description can be replaced by the input.

Example. In our examples, TR_3 and TR_{10} are utmost short test reports in which the descriptions just contain one word after word segmentation and stop words removal, but their inputs contain seven and nine words, respectively. As a result, the descriptions are replaced with the inputs by implementing the description enhancement strategy, as shown in Table 6. For the other test reports, the descriptions keep unchanged.

Vector space model: Vector space model is universally used to process the unstructured text information [43, 51]. In vector space model, we represent each test report as a vector $\overrightarrow{TR_i} = (e_{i,1}, e_{i,2}, \ldots, e_{i,n})$ based on an established dictionary, where n is the size of the dictionary. Each dimension in the vector corresponds to a term in the dictionary. The number of occurrences of a term in the description can be used to denote the weight of the corresponding dimension.

Example. We establish a dictionary including 30 terms for the enhanced descriptions of eight test reports, as shown in Table 7. With the established dictionary, we construct a vector space model, as shown in Table 8.

Table 7. Dictionary

No.	Word	No.	Word	No.	Word	No.	Word	No.	Word	No.	Word
D_1	applic	D_6	circl	D_{11}	friend	D_{16}	oper	D_{21}	remind	D_{26}	show
D_2	background	D_7	complet	D_{12}	horizont	D_{17}	pictur	D_{22}	run	D_{27}	system
D_3	blog	D_8	download	D_{13}	messag	D_{18}	present	D_{23}	screen	D_{28}	user
D_4	check	D_9	end	D_{14}	micro	D_{19}	problem	D_{24}	scroll	D_{29}	view
D_5	choos	D_{10}	fail	D_{15}	open	D_{20}	recommend	D_{25}	share	D_{30}	wechat

Table 8. Vector space model for eight test reports

	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	D_{16}	D_{17}	D_{18}	D_{19}	D_{20}	D_{21}	D_{22}	D_{23}	D_{24}	D_{25}	D_{26}	D_{27}	D_{28}	D_{29}	D_{30}
TR_3	0	0	0	0	0	1	0	0	0	1	2	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1
TR_4	0	1	1	0	0	1	0	0	0	1	2	0	0	1	0	1	1	0	0	0	0	1	0	0	2	1	1	0	0	1
TR_5	1	0	0	0	0	0	1	2	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	1	1	0	0
TR_6	1	0	0	0	1	0	1	2	1	0	0	1	0	0	1	0	2	0	1	0	2	0	1	1	1	0	2	1	0	0
TR_7	1	0	0	0	0	1	0	2	0	1	2	0	1	0	1	0	3	1	0	1	0	0	0	0	1	0	2	0	0	1
TR_8	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	2	0	0	0	0	0	1	1	0	1	1	0	0	0
TR_9	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0
TR_{10}	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	2	0	0	0	1	0	1	1	0	0	1	0	1	0

Similarity computation: After transforming test reports into vectors, we measure the similarities by the cosine similarity for each pair of test reports. The cosine similarity is calculated by the following formula [30, 43]:

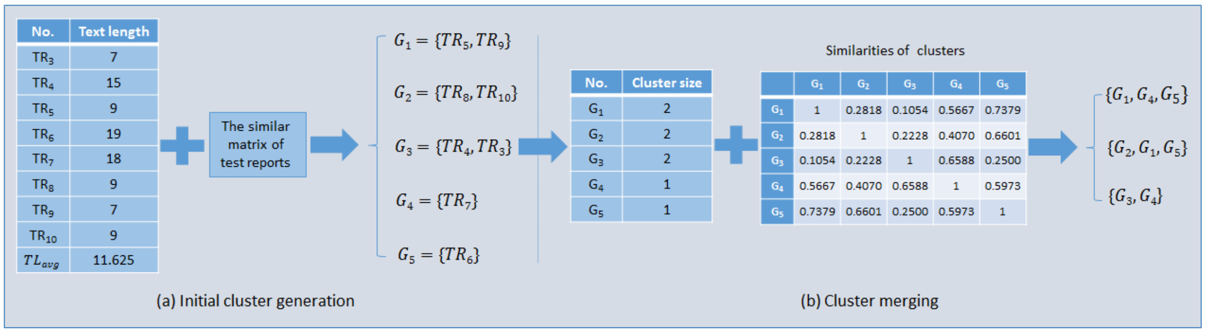
$$Sim(\overrightarrow{TR_1}, \overrightarrow{TR_2}) = \frac{\overrightarrow{TR_1} \cdot \overrightarrow{TR_2}}{|\overrightarrow{TR_1}||\overrightarrow{TR_2}|} \tag{1}$$

Example. Table 9 shows the similarity results of pairs of test reports.

4.3 Fuzzy Clustering

Various methods related to clustering analysis have been proposed, including hierarchical methods, partitioning relocation methods, density-based partitioning methods, and grid-based methods [4]. Many two-phase clustering algorithms [16, 26] derived from hierarchical methods have been widely proposed in document clustering, e.g., Maximum Capturing (MC) [59], Frequent Word Sequences (CFWS) [26], and Frequent Word Meaning Sequences (CFWMS) [26]. These methods either extract features or build a vector space model to calculate similarities for generating initial clusters in the first phase, and then merge these clusters further according to their similarities or overlap degrees in the second phase. Unfortunately, they are not fuzzy document clustering methods and cannot partition multi-bug test reports into multiple clusters. Nevertheless, inspired by these algorithms, we propose a new fuzzy version of two-phase merging algorithm. In our algorithm, we build a vector space model to calculate similarities for initial cluster generation and prescribe that a test report is partitioned prohibitively into different clusters in the first phase. In the second phase, we implement the merger of clusters so as to assign multi-bug test reports into different clusters using a new similarity measurement.

4.3.1 Initial cluster generation. As to document clustering, it is important to determine the cluster centroids named seed test reports. If a multi-bug test report is chosen as a seed, test reports revealing different bugs may be incorrectly partitioned into the same cluster. In contrast, if an utmost short test report is chosen as a seed, test reports revealing the same bug may be incorrectly assigned to other



No.	Text length
TR$_3$	7
TR$_4$	15
TR$_5$	9
TR$_6$	19
TR$_7$	18
TR$_8$	9
TR$_9$	7
TR$_{10}$	9
TL_{avg}	11.625

$G_1 = \{TR_5, TR_9\}$

$G_2 = \{TR_8, TR_{10}\}$

$G_3 = \{TR_4, TR_3\}$

$G_4 = \{TR_7\}$

$G_5 = \{TR_6\}$

No.	Cluster size
G$_1$	2
G$_2$	2
G$_3$	2
G$_4$	1
G$_5$	1

Similarities of clusters

	G$_1$	G$_2$	G$_3$	G$_4$	G$_5$
G$_1$	1	0.2818	0.1054	0.5667	0.7379
G$_2$	0.2818	1	0.2228	0.4070	0.6601
G$_3$	0.1054	0.2228	1	0.6588	0.2500
G$_4$	0.5667	0.4070	0.6588	1	0.5973
G$_5$	0.7379	0.6601	0.2500	0.5973	1

$\{G_1, G_4, G_5\}$

$\{G_2, G_1, G_5\}$

$\{G_3, G_4\}$

The similar matrix of test reports
(a) Initial cluster generation
(b) Cluster merging

Fig. 3. The clustering procedure of two-phase merging algorithm.

Table 9. The similarities of pairs of test reports

No.	TR$_3$	TR$_4$	TR$_5$	TR$_6$	TR$_7$	TR$_8$	TR$_9$	TR$_{10}$
TR_3	1	0.7647	0.1005	0.1925	0.6694	0.2010	0.1260	0.2010
TR_4	0.7647	1	0.1383	0.2649	0.5864	0.2767	0.1734	0.2075
TR_5	0.1005	0.1383	1	0.6963	0.6606	0.2727	0.7977	0.2727
TR_6	0.1925	0.2649	0.6963	1	0.5973	0.5803	0.8001	0.6963
TR_7	0.6694	0.5864	0.6606	0.5973	1	0.4404	0.5521	0.4404
TR_8	0.2010	0.2767	0.2727	0.5803	0.4404	1	0.3419	0.8182
TR_9	0.1260	0.1734	0.7977	0.8001	0.5521	0.3419	1	0.4558
TR_{10}	0.2010	0.2075	0.2727	0.6963	0.4404	0.8182	0.4558	1

clusters. Based on the above considerations, we adopt the text length as the metric to determine seed test reports. Algorithm 1 shows the procedure of initial cluster generation. Given a set TR of test reports $TR_1, TR_2, \ldots, TR_m$, we calculate the text length $TL(TR_i)$ of each test report by counting the number of words as well as the average text length TL_{avg} of all test reports. First, the test report TR_k whose text length being closest to TL_{avg} is chosen as a seed and removed from TR. Then a new cluster is created and other test reports whose similarities with TR_k exceeding a threshold value δ_1 can be put into this cluster. These test reports are removed from TR. The above steps repeat until the set TR becomes empty, then the algorithm terminates and the initial clusters are returned.

The threshold value δ_1 is an important parameter which controls the number of clusters. A large value of δ_1 leads to more clusters and a small value leads to fewer clusters.

Example. As shown Fig. 3.a, the average text length is TL_{avg}=11.625, so TR_5 is selected as the first seed test report and a new cluster G_1 is created. Assuming that the similarity threshold value δ_1=0.7, TR_9 will be put into G_1. Similarly, two new clusters G_2 and G_3 are created and include two test reports, respectively. Finally, TR_7 and TR_6 are selected as seed test reports and two new clusters G_4 and G_5 are created, respectively.

4.3.2 Cluster merging. Algorithm 1 is run to form the initial clusters $G = \{G_1, G_2, \ldots, G_c\}$. The number of initial clusters is usually larger than the ground-truth and these clusters can be further merged. Existing studies [10, 16, 26] have adopted different strategies to merge clusters. However, these strategies cannot be directly adapted to our datasets without taking multi-bug test reports into consideration. Consequently, we propose a new strategy to merge clusters in our study. In the newly obtained clusters, each one contains a certain number of test reports. We treat all test reports in one cluster as a new document and re-calculate the similarities for pairs of clusters. Similarly, it is also important to determine

Algorithm 1: Initial cluster generation

```
Input: $m$ test reports $TR$=$\{TR_1, TR_2, \ldots, TR_m\}$ and similarity matrix $Sim\_TR$
Output: $c$ clusters
$G$=$\emptyset$; $c$=0;
For each test report $TR_i$, calculate the text length $TL(TR_i)$=$\sum_{i=1}^{n} e_{i,j}$;
For all test reports, calculate the average text length $TL_{avg}$=$\sum_{i=1}^{m} TL(TR_i)/m$;
while $TR \neq \emptyset$ do
    Select the test report $TR_k$ from $TR$, s.t. $TL_k = \underset{TL(TR_i)}{\arg\min}(|TL(TR_i) - TL_{avg}|)$;
    $c$=$c+1$;
    $G_c$=$\{TR_k\}$; // Select $TR_k$ as a seed test report
    $TR$=$TR \backslash \{TR_k\}$;
    // Add other test reports to the new cluster
    for ($TR_i \in TR$) do
        if ($Sim\_TR(TR_i, TR_k) \geq \delta_1$) then
            $G_c$=$G_c \bigcup \{TR_i\}$;
            $TR$=$TR \backslash \{TR_i\}$;
        end
    end
    $G$=$G \bigcup \{G_c\}$;
end
Return $G$;
```

the seed clusters. Before that, we introduce two kinds of prior knowledge related to crowdsourced test reports.

First, workers prefer to revealing simple bugs rather than complex ones, thus resulting in massive redundant test reports revealing same bugs. The numbers of test reports in some clusters heavily outnumber those of some others. As a result, we adopt the cluster size as a metric to help determine seed clusters for merging.

Second, in general, multi-bug test reports have high similarity values with distinct test reports. By an in-depth observation to the annotation results as shown in Table 3, most of multi-bug test reports reveal two or three bugs. For example, there are 54 test reports revealing two or three bugs over UBook. In contrast, only 3 test reports reveal more than three bugs. As a result, we prescribe that a cluster can be merged into no more than three seed clusters.

With the above prior knowledge, we let $Num(G_i)$ denote the times of being merged of G_i. The merging procedure is shown in Algorithm 2. First, all clusters subjected to $Num(G_i) = 0$ are selected to construct a candidate set SC. The largest sized cluster G_k is selected as a seed from SC and removed from G. Then other clusters whose similarities with G_k exceeding a threshold value δ_2 can be merged into G_k. These merged clusters are still retained in G, but no longer selected as seed clusters. If a cluster is merged for three times, namely $Num(G_i) = 3$, it should be removed from G. The above steps repeat until no new seed cluster is available or G becomes empty, and the algorithm terminates and the results are returned.

Example. As shown Fig. 3.b, G_1 is selected as the first seed cluster. Assuming that the similarity threshold value δ_2 is set to 0.5, in this case, G_4 and G_5 can be merged into G_1. Similarly, G_2 and G_3 are selected as the second and the third seed clusters, respectively. Although TR_6 reveals three bugs, but G_5 containing test report TR_6 is only merged for two times due to limited information for the third bug.

Algorithm 2: Cluster merging

```
Input: $c$ clusters $G$={$G_1, G_2, \ldots, G_c$} and similarity matrix $Sim\_C$
Output: $d(d < c)$ clusters
$S$=$\emptyset$;
For each cluster $G_i$, $Num(G_i)$=0;
while $G \neq \emptyset$ do
    $SC$=$\{G_i | G_i \in G, Num(G_i) = 0\}$;
    if ($SC$==$\emptyset$) then
        break;
    end
    Select the largest sized cluster $G_k$ from $SC$; // Select $G_k$ as the seed cluster
    $G$=$G \backslash \{G_k\}$;
    for ($G_i \in G$) do
        if ($Sim\_C(G_i, G_k) \geq \delta_2$) and ($Num(G_i)$ <3) then
            $G_k$=$G_k \bigcup G_i$;
            $Num(G_i)$=$Num(G_i) + 1$;
        end
        if ($Num(G_i)$==3) then
            $G$=$G \backslash \{G_i\}$;
        end
    end
    $S$=$S \bigcup \{G_k\}$;
end
Return $S$; // $d$=$|S|$ clusters are returned;
```

5 EXPERIMENTAL SETUP

In this section, we detail the experiment setup, including experiment platform, parameter settings, experimental datasets, and evaluation metrics.

5.1 Experiment Platform and Parameter settings

All the experiments are conducted with JavaJDK1.8.0_60, compiled with Eclipse 4.5.1, and run on a PC with 64-bitWin 8.1, Intel Core(TM) i7-4790 CPU, and 8G memory.

Our TERFUR framework involves two parameters, namely the similarity threshold values δ_1 and δ_2, which may impact the clustering results. As for the empirical evaluation, we set δ_1=0.8 and δ_2=0.3 as the default parameter values and present the tuning results in Section 6.1.

5.2 Experimental Datasets

We collect five crowdsourced test report datasets from industrial Apps, namely Justforfun with 291 test reports, SE-1800 with 348 test reports, iShopping with 408 test reports, CloudMusic with 238 test reports, and UBook with 443 test reports. All test reports are manually annotated by students. We carefully check the annotation results and submit them to developers for further validation. Five datasets include 773 invalid test reports and 184 multi-bug test reports. 174 out of 184 multi-bug test reports reveal two or three bugs. In TERFUR, all test reports are taken as the input of the filter and 745 invalid test reports are successfully filtered out. Then the remaining test reports are processed further by NLP in the preprocessor and automatically partitioned into clusters by the two-phase merging algorithm.

5.3 Metrics

Cluster validity assessment, namely clustering result analysis, is divided into internal validity assessment and external validity assessment [19, 32]. Internal validity assessment considers both the inter-cluster similarity and the inner-cluster similarity, which usually uses objective functions to measure the performance of algorithms. Thus ideal results may be not intrinsically appropriate to the actual data. External validity assessment considers the consistency between clustering results and expected results, which usually adopts manual evaluation to measure the performance of algorithms. In document clustering, the quality of clustering results depends on the subjective judgments of professionals, thus external validity assessment is more suitable to evaluate document clustering results [32].

Precision, recall, and F-measure derived from external validity assessment are the most commonly used metrics, which have been widely employed to evaluate the performance of algorithms in document clustering [10, 16] and document classification [20, 27]. In the same way, we first employ precision, recall, and F-measure to evaluate the local result for each cluster, and then the global result by averaging over the results of all clusters, namely microAverage Precision (*AverageP*), microAverage Recall (*AverageR*), and microAverage F1-measure (*AverageF1*) [16, 20].

AverageP is an important metric to measure the cohesion degree of each cluster in the clustering results. Assuming that $G = \{G_1, G_2, \ldots, G_k\}$ and $P = \{P_1, P_2, \ldots, P_c\}$ represent the clustering results and the results of manual annotation, respectively, k does not necessarily equate to c, and G_i corresponds P_j. The formula of *AverageR* is as follow [20]:

$$AverageP = \frac{\sum_{i=1}^{k} TP_i}{\sum_{i=1}^{k} (TP_i + FP_i)} \tag{2}$$

AverageR is an important metric to evaluate the level of consistency between the clustering results and the manual annotation. The formula is as follow [20]:

$$AverageR = \frac{\sum_{i=1}^{k} TP_i}{\sum_{i=1}^{k} (TP_i + FN_i)} \tag{3}$$

AverageF1 is an external evaluation metric based on the combination of *AverageP* and *AverageR*. The better the results of clustering, the higher the value of *AverageF1* is. The formula is as follow [20]:

$$AverageF1 = \frac{2AverageP * AverageR}{AverageP + AverageR} \tag{4}$$

Where TP_i is the number of true positives, namely the numbers of test reports belonging to G_i and P_j. FP_i is the number of false positives, namely the number of test reports belonging to non-G_i and P_j. FN_i is the number of false negatives, namely the number of test reports belonging to G_i and non-P_j.

6 EXPERIMENTAL RESULTS

In this section, we investigate five research questions and conduct experiments on five industrial datasets to evaluate TERFUR. In addition, we also verify whether TERFUR can greatly reduce the cost of manual inspection for developers.

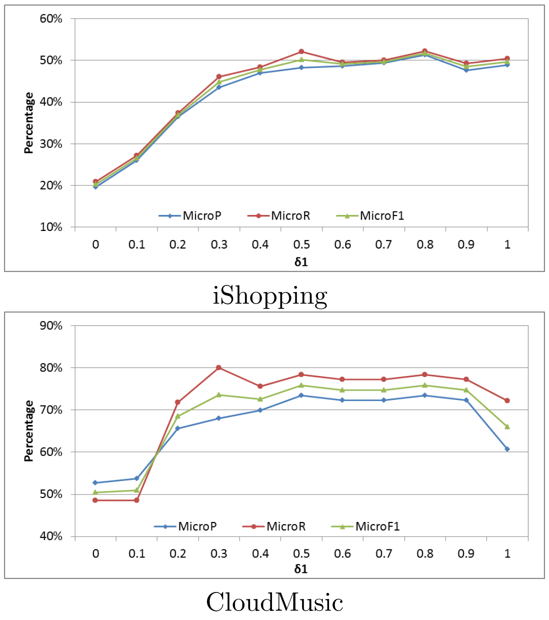


Fig. 4. Results of TERFUR with different δ_1 values.

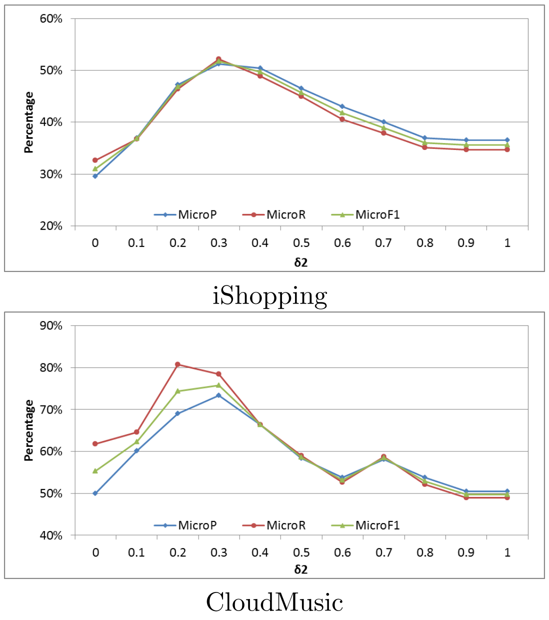


Fig. 5. Results of TERFUR with different δ_2 values.

6.1 Investigation to RQ1

RQ1. How do the two parameters affect the performance of TERFUR?

Motivation. Two parameters in the two-phase merging algorithm, namely the similarity threshold values δ_1 and δ_2, play a decisive role in clustering test reports. In this RQ, we mainly investigate the impacts of the two parameters and attempt to find appropriate values to make sure that they can be applied to all datasets.

Approach. Since there are two parameters in TERFUR, we gradually increase the value of one parameter from 0 to 1 and keep the other one in a fixed value. We predefine a tuning step to 0.1 and run TERFUR over iShopping and CloudMusic. Since developers expect accurate clustering results, we regard *AverageP* as a main metric to evaluate the behavior of TERFUR over the two datasets. The best values of parameters are determined and then applied to others. In order to facilitate observation, the tuning results are presented in Fig. 4 and Fig. 5 over iShopping and CloudMusic with respect to δ_1 and δ_2, respectively.

Results. In Fig. 4, we use two sub-figures to present the tuning results when δ_2=0.3 and δ_1 varies in [0, 1]. As the change of δ_1, TERFUR achieves different results in terms of *AverageP*, *AverageR*, and *AverageF1* over iShopping. When δ_1 is set to 0.8, TERFUR rises to the peak and achieves 51.29% in terms of *AverageP*, 52.19% in terms of *AverageR*, and 51.74% in terms of *AverageF1*. The similar findings can be observed over CloudMusic. When δ_1 is equal to 0.8, TERFUR achieves the best value 73.40% in terms of *AverageP*, the corresponding values of *AverageR* and *AverageF1* are 78.41% and 75.82%, respectively. Hence, δ_1=0.8 is a good choice in our datasets.

In Fig. 5, we also use two sub-figures to present the tuning results when δ_1=0.8 and δ_2 varies in [0, 1]. As shown in the figure, TERFUR shows basically similar tendency with the change of δ_2. For example, TERFUR rises from 29.50% to 51.29% and falls from 51.29% to 36.52% in terms of *AverageP* with the continuous growth of δ_2 from 0 to 1 over iShopping. *AverageR* and *AverageF1* follow almost the same change trends as well. When δ_2 is set to 0.3, TERFUR achieves the best results in terms of *AverageP*, *AverageR*, and *AverageF1*. Hence, δ_2=0.3 is a good choice in our datasets. In the following experiments, we keep the two parameters in their tuned values, namely δ_1=0.8 and δ_2=0.3.

Conclusion. The two parameters influence the results of TERFUR. The values of three metrics vary with the changes of the two parameters and are more sensitive to δ_1. According to the tuning results, δ_1=0.8 and δ_2= 0.3 may be effective to apply to all datasets.

6.2 Investigation to RQ2

RQ2. Can TERFUR outperform classical Fuzzy C-Means clustering algorithm?

Motivation. Given multi-bug test reports in our datasets, FULTER can be viewed as a special fuzzy document clustering problem. We explore a new two-phase merging algorithm to automatically partition test reports into clusters. To further evaluate FULTER, we carry out another experiment by comparison with Fuzzy C-Means algorithm (FCM), a classical fuzzy clustering algorithm, which has been widely applied to fuzzy document clustering [33]. In this RQ, we mainly investigate whether TERFUR outperforms FCM.

Approach. FCM tries to minimize the squared error objective function [5, 33] by iteratively updating the prototypes (namely cluster centroids) and the fuzzy membership of objects (namely test reports in this study) to all clusters. FCM usually requires Euclidean relations between objects [45]. In our experiment, we replace the two-phase merging algorithm with FCM in the third component and transform the text relations into Euclidean ones by normalizing vector space model. There are two key parameters in FCM, namely number of clusters and weighting exponent [33]. In order to conduct convincing comparison, we gradually increase the number of clusters to determine the best results while setting the parameter of weighting exponent to an empirical value 2 [5, 33].

Results. Table 10 shows the comparative results in terms of *AverageP*, *AverageR*, and *AverageF1* between TERFUR and FCM. Obviously, TERFUR outperforms FCM over all datasets. For example, TERFUR improves FCM by 14.04% in terms of *AverageP*, 28.20% in terms of *AverageR*, and 22.73% in terms of *AverageF1* over Justforfun. As seen from the results, TERFUR performs well over Justforfun, CloudMusic, and UBook. Due to the poor quality of test reports of SE-1800 and iShopping, both TERFUR and FCM achieve low *AverageP*, *AverageR*, and *AverageF1*. As for SE-1800, the descriptions of most test reports contain a lot of mixed information (e.g., testing details and test steps), which makes a great negative impact on similarity computation. As for iShopping, 230 valid test reports reveal 68 bugs (namely clusters) which have been validated by developers, as shown in Talbe 2. Many clusters may contain only one test report which may be falsely partitioned to other clusters with a high probability.

FCM works poorly in partitioning crowdsourced test reports into clusters over all the datasets. For example, FCM only achieves 23.37% in terms of *AverageP*, 32.53% in terms of *AverageR*, and 27.19% in terms of *AverageF1* over iShopping. Although FCM performs well over Jusforfun and CloudMusic and achieves 64.11% and 64.81% in terms of *AverageP*, respectively, there is still a wide gap between FCM and TERFUR.

Conclusion. FCM can be applied to resolve the FULTER problem, however, it is not an effective method to partition test reports into clusters. TERFUR works well and achieves better results than FCM.

6.3 Investigation to RQ3

RQ3. Do the two filtering rules really work to break through the invalid barrier?

Table 10. Results of TERFUR and FCM over all datasets.

Dataset	TERFUR			FCM		
	AverageP	AverageR	AverageF1	AverageP	AverageR	AverageF1
Justforfun	78.15%	73.01%	75.49%	64.11%	44.81%	52.76%
SE-1800	53.62%	47.43%	50.34%	38.72%	42.64%	40.58%
iShopping	51.29%	52.19%	51.74%	23.37%	32.53%	27.19%
CloudMusic	73.40%	78.41%	75.82%	64.81%	45.35%	53.36%
UBook	67.68%	55.35%	60.90%	35.99%	43.26%	39.29%

Table 11. Some statistical results after applying two rules

	Justforfun	SE-1800	iShopping	CloudMusic	UBook
#Report	291	348	408	238	443
#Invalid Report	61	146	193	149	238
#Falsely F-Report	0	1	0	0	0
#Correctly F-Report	59	143	178	145	220
#Unsuccessfully F-Report	2	3	15	4	22

#Falsely F-Report: The number of valid test reports filtered out falsely by the two rules.
#Correctly F-Report: The number of invalid test reports filtered out correctly by the two rules.
#Unsuccessfully F-Report: The number of invalid test reports filtered out unsuccessfully by the two rules.

Motivation. A mass of invalid reports in our datasets may waste plenty of time of developers. Meanwhile, these test reports may impose negative impacts on our clustering algorithm. We elaborately design two rules and integrate them into the first component of TERFUR to filter out invalid test reports to break through the *invalid barrier*. In this RQ, we mainly validate whether the two rules work to remove invalid test reports.

Approach. Invalid test reports have been carefully annotated by the students and tagged as "invalid". We take all test reports as the input of the filter, then independently run the filter. We record the results and compare the remaining test reports with the original datasets.

Results. Table 11 shows the results after applying the filtering rules. In our datasets, invalid test reports account for 20.96%-62.61%. With the filter, 59, 143, 178, 145, and 220 invalid test reports are filtered out, respectively. The numbers of remaining test reports in five datasets are 232, 204, 230, 93, and 223, respectively, which means the numbers of inspected test reports have been greatly reduced. Only 2, 3, 15, 4, and 22 invalid test reports are not filtered out by our rules. In our framework, we randomly sample a subset of invalid test reports and consequently design the two heuristic rules. However, there are still some invalid test reports which do not match the heuristic rules. Nevertheless, most of invalid test reports can be filtered out. Only one valid test report is falsely filtered out by our rules over SE-1800, which has little impact on TERFUR for clustering test reports.

Conclusion. Our filtering rules work well to filter out invalid test reports and ensure that nearly all the valid test reports are not filtered out. The two well-designed rules can effectively reduce the unnecessary cost in inspecting test reports for developers.

6.4 Investigation to RQ4

RQ4. Can the description enhancement strategy effectively improve the effectiveness of TERFUR?

Motivation. Due to the poor quality of test reports, we adopt the description enhancement strategy to selectively enhance the descriptions with the inputs following certain rules to break through the *uneven barrier*. In this RQ, we mainly investigate whether the description enhancement strategy can help clustering.

Table 12. Impact of description enhancement strategy on experimental results

Dataset	TERFUR			TERFUR-D			TERFUR-ID		
	AverageP	AverageR	AverageF1	AverageP	AverageR	AverageF1	AverageP	AverageR	AverageF1
Justforfun	78.15%	73.01%	75.49%	71.49%	57.86%	63.96%	81.19%	78.34%	79.74%
SE-1800	53.62%	47.43%	50.34%	51.69%	47.56%	49.54%	45.32%	44.29%	44.80%
iShopping	51.29%	52.19%	51.74%	53.91%	53.91%	53.91%	50.43%	51.10%	50.77%
CloudMusic	73.40%	78.41%	75.82%	70.21%	75.86%	72.93%	61.17%	74.12%	67.02%
UBook	67.68%	55.35%	60.90%	74.70%	70.19%	72.37%	64.12%	60.87%	62.45%

Approach. In our datasets, both the inputs and the descriptions contain natural language information. Two intuitive strategies for similarity measurement are to utilize either the descriptions or the combination of both the inputs and the descriptions. In TERFUR, we consider the two strategies to calculate similarity as baselines for comparisons in the preprocessing. To do so, we name them TERFUR-D (the description) and TERFUR-ID (the combination of both the input and the description). Likewise, we tune the two parameters of TERFUR-D and TERFUR-ID over Justforfun and CloudMusic, respectively, and then apply the tuned parameters to all datasets.

Results. Table 12 presents the evaluation results in terms of *AverageP*, *AverageR*, and *AverageF1* of TERFUR, TERFUR-D, and TERFUR-ID. TERFUR and TERFUR-D outperform TERFUR-ID in terms of *AverageP*, *AverageR*, and *AverageF1* over all the datasets but Justforfun. For example, compared with TERFUR-ID, TERFUR and TERFUR-D achieve 12.23% and 9.04% improvements in terms of *AverageP*, 4.29% and 1.74% improvements in terms of *AverageR*, and 8.80% and 5.91% improvements in terms of *AverageF1* over CloudMusic, respectively. TERFUR achieves better results than TERFUR-D in terms of *AverageP*, *AverageR*, and *AverageF1* over Justforfun, SE-1800, and CloudMusic. For example, TERFUR improves TERFUR-D by up to 6.66% in terms of *AverageP*, 15.15% in terms of *AverageR*, and 11.53% in terms of *AverageF1* over Justforfun, respectively. In contrast, TERFUR-D achieves better results than TERFUR over iShopping and UBook. For example, TERFUR-D improves TERFUR by up to 7.02% in terms of *AverageP*, 14.84% in terms of *AverageR*, and 11.47% in terms of *AverageF1* over UBook. Meanwhile, TERFUR-ID achieves better results than both TERFUR and TERFUR-D over Justforfun.

Conclusion. The inputs of test reports can enhance the descriptions, but sometimes may bring noises. As a consequence, better results can be available when choosing an appropriate strategy by prejudging the degree of similarity between the inputs and descriptions. Clearly, our description enhancement strategy is more effective than using the combination of both the input and the description.

6.5 Investigation to RQ5

RQ5. Can TERFUR reduce the number of inspected test reports for developers?

Motivation. Developers cannot check all test reports in reality. Driven by the motivation, we thus issue the new problem of FULTER. In an ideal case, test reports in one cluster detail the same bug after fuzzy clustering, developers only need to inspect one representative test report from each cluster. However, it is hard to achieve the 100% accuracy by automated clustering algorithms. In this RQ, we investigate experimentally whether TERFUR can reduce manual efforts by prioritizing test reports.

Approach. In the literature, a prioritization technique combining a Diversity strategy and a Risk strategy (DivRisk) has been proposed to prioritize test reports [13]. More specifically, keywords are extracted first from test reports to build a keyword vector model. Then the risk values of test reports are calculated and the distance matrix is constructed based on the keyword vector model. Next, the most risky test report is selected as the first one for inspection. After that, n_c (n_c=8) test reports with the largest distance(s) with the inspected ones are selected to form a candidate set, and the most risky

Table 13. The average number of inspected test reports

App	Algorithm	25%	50%	75%	100%
Justforfun	Best	1.25	8.5	14.75	21
	DivRisk	3.25	47.5	117.5	208
	TERFUR-DivRisk	3.25	18.5	64.25	207
SE-1800	Best	4	10	18	26
	DivRisk	18	47	92	241
	TERFUR-DivRisk	12	27	50	145
iShopping	Best	6	18	35	52
	DivRisk	13	39	79	228
	TERFUR-DivRisk	22	42	91	206
CloudMusic	Best	2.25	7.5	12.75	18
	DivRisk	10.25	34	50.5	128
	TERFUR-DivRisk	5	10.5	15.75	39
UBook	Best	2	7	14.5	22
	DivRisk	8.5	34	92	223
	TERFUR-DivRisk	6.5	23	45.5	75

test report is selected from the set for inspection. When detecting a real bug, all keywords of this test report in the keyword vector model are increased by a given value δ ($\delta = 0.2$). Finally, the prioritization sequence is returned.

Although DivRisk achieves competitive results for developers, there is still a room for improvement for reducing manual inspection. Thus, we propose an improved test repost prioritization technique by combining TERFUR and DivRisk (TERFUR-DivRisk). In TERFUR-DivRisk, we first run TERFUR to generate the clustering results. Then the most risky test report is selected from the largest sized cluster and removed from all clusters. Next, $n_c(n_c\!=\!8)$ test reports with the largest distance(s) with the inspected ones are selected from the second largest sized cluster to construct the candidate set, and the most risky test report is selected from the set. We remove this test report from all clusters and update the keyword vector model as well. After all clusters are operated once, we start a new round from the largest sized cluster. The above steps repeat until all clusters become empty, then the procedure terminates and the recommendation sequence is returned.

In order to verify the efficiency of TERFUR-DivRisk, we employ the cost of inspection to detect the given number (25%, 50%, 75%, and 100%) of bugs as the metric and introduce linear interpolation [13, 25] to determine the number of inspected test reports. We choose **Best** and DivRisk [13] prioritization strategies as baselines for comparisons. Assuming that we know in advance which test reports reveal true bugs or are multi-bug ones. Naturally, multi-bug test reports should be inspected first in the **Best** strategy. For example, there are 25 bugs in Justforfun, two multi-bug test reports revealing four and three different bugs will be inspected first. In this case, only 1.25 test reports need to be inspected in the case of detecting 25% bugs (namely 6.25 bugs) according to the linear interpolation. In addition, we run DivRisk over the original datasets.

Results. Table 13 records the numbers of inspected test reports in detecting 25%, 50%, 75%, 100% bugs. As shown in the table, we observe that **Best** just needs to inspect a small amount of test reports over all the datasets. For example, **Best** needs to inspect 1.25, 8.5, 14.75, and 21 test reports for detecting 25%, 50%, 75%, and 100% bugs over Justforfun, respectively. Correspondingly, DivRisk needs to inspect 3.25, 47.5, 117.5, and 208 test reports and TERFUR-DivRisk needs to inspect 3.25, 18.5, 64.25, and 207 test reports, respectively.

TERFUR-DivRisk outperforms DivRisk over all the datasets but iShopping and can provide best approximation to the **Best** results. For example, TERFUR-DivRisk needs to inspect 5, 10.5, 15.75, and 39 test reports for detecting 25%, 50%, 75%, 100% bugs over CloudMusic, the numbers of inspected test reports are close to those of **Best**. Compared with DivRisk, TERFUR-DivRisk can achieve 51.22%, 69.12%, 68.81%, and 69.53% improvements with respect to the number of inspected test reports, respectively. Unfortunately, TERFUR-DivRisk needs to inspect more test reports than DivRisk in the case of detecting 25%, 50%, and 75% bugs over iShopping. The reason for this may be due to that TERFUR provides the lowest $AverageP$ over this dataset, i.e., TERFUR does not work well to cluster test reports of iShopping.

Conclusion. The results of TERFUR-DivRisk are significantly better than those of DivRisk. By combining TERFUR and DivRisk, developers only need to inspect a small number of test reports for detecting 25%, 50%, and 75% bugs.

7 THREATS TO VALIDITY

In this section, we discuss the threats to validity, including external threats and internal threats.

7.1 External Threats

In crowdsourced testing, workers are recruited in an open call format. In general, they have no social relationship with each other in this situation [17, 31]. In this study, students are invited as workers to perform testing and submit test reports. In such a way workers have certain social relations, so the results may be different from those of workers from open platforms. In the literature [44], an empirical study shows that both students and professionals present similar performance for a new emerging technology. Thus, this threat has been reduced.

In this study, developers have no enough time to annotate all test reports. Therefore, we invite three graduate students to annotate the datasets independently. Their experience and personal opinions may influence the annotation results. In order to minimize the bias, each test report belongs to a cluster if it gets two or more votes from students. We submit the ambiguous test reports to developers for exact decisions.

7.2 Internal Threats

In our experiment, all test reports are written in Chinese, which may threaten the generalization of our technique to other natural languages. However, NLP techniques are widely used in text processing and can be applied to various natural languages. Using other NLP tools, such as Stanford NLP toolkit[6], the vector space model can be built for test reports written in other natural languages. Meanwhile, the two-phase merging algorithm is associated with text similarity but independent of textual information. Therefore, this threat will be minimized.

In TERFUR, we propose a new fuzzy version of two-phase merging algorithm based on hierarchy clustering. As a consequence, a potential drawback exists in our algorithm. That is, if a test report is assigned into a wrong cluster in the first phase, it will always belong to this cluster in the second phase. In order to overcome this drawback, we try to ensure that test reports in one cluster detailing the same bug by setting a large similarity threshold value in the first phase.

[6]http://nlp.stanford.edu/software

Table 14. A comparison among the three tasks from distinct aspects

	Crash reports bucketing	Duplicate bug report detection	Crowdsourced test report fuzzy clustering
Task	Find a similar bucket	Find a similar class	Partition all test reports into clusters
Submission Frequency	Very high	Low	High
Completeness	Part of a bug report	A complete bug report	A complete test report
Redundancy Degree	Very high	Low	High

8 RELATED WORK

In this section, we summarize previous studies related to our work. There are four major areas: crash report bucking, duplicate bug report detection, crowdsourced testing, and fuzzy document clustering.

8.1 Crash Report Bucketing

Crash reporting system collects crash reports and classifies similar ones to the same bucket to automatically produce bug reports. Cause analysis and classification for crash reports are important issues for developers.

Many studies explore call stack information to classify crash reports. The failure similarity classifier is built to calculate the similarities between crash reports based on the tuned call stack edit distances [3]. However, it is hard to be implemented since the whole model will take great computing cost on generating a total of 11 penalty parameters. A crash graph is constructed by extracting call stacks from multiple crash reports in one bucket, which can provide aggregated information about crashes for developers [21]. ReBucket measures the similarities using Position Dependent Model (PDM) by extracting simplified call stacks and then partitions crash reports into distinct buckets using the hierarchical clustering method [11]. In addition, the literature implements the call stack matching by measuring the similarities of their function names to quickly identify the recurrences of crashes [34].

The target of crash report bucketing is to assign newly arrived crash reports to corresponding buckets to automatically form bug reports. Its resolutions usually leverage structured stack traces to extract features and calculate the similarities between crash reports. In contrast, our work aims to cluster all test reports. Table 14 summarizes some differences between the two tasks from different aspects. Different from crash reports, crowdsourced test reports for Apps are usually written with different free-form texts on mobile phones and have relatively low redundancy and submission frequency.

8.2 Duplicate Bug Report Detection

In bug report resolutions [55–57], duplicate bug report detection is one of the most important tasks. Assigning duplicate bug reports to different developers for fixing will take up a great deal of manpower, which motivates researchers to seek efficient solutions to this problem.

Many methods are proposed for duplicate bug report detection [7, 43]. NLP techniques are one of the most used methods and five key steps including tokenization, stemming, stop words removal, vector space representation, and similarity calculation are sequentially performed [43]. Some researchers also leverage IR techniques for duplicate bug report detection. For example, IR techniques are uniformly adopted to calculate the similarities for both natural language information and execution information [51]. However, it requires additional efforts to create the execution information. To overcome the differences of different descriptive terms in bug reports, IR techniques and topic modeling are combined for detecting duplicate bug reports [36]. Another body of method is machine learning. A discriminative model is trained by Support Vector Machine (SVM) to retrieve duplicate bug reports from a collection [47]. However, SVM requires a long time to build the model. To overcome the shortcoming, a retrieval function extending the BM25F is proposed in [46].

In duplicate bug report detection, existing methods either build vector space model or extract features for similarity computation and recommend the most relevant historical bug reports for an emerging one. In contrast, our work aims to partition all test reports into clusters. Table 14 summarizes the differences between the two tasks from different aspects. Compared against bug reports, crowdsourced test reports for Apps have the following characteristics. First, test reports are often submitted by non-professional testers. Second, test reports are written in natural language on mobile phones, thus they are generally shorter and less informative, but also include more screenshots due to the ease of capturing screenshots on mobile phones. Finally, test reports submitted by workers in a short time may be highly redundant and unstructured.

8.3 Crowdsourced Testing

Crowdsourcing is proposed by Howe and Robinson in 2006, which is the process of an organization crowdsourcing their work to undefined, online individuals in an open call form [17, 31]. It tries to resolve problems by combining both human and machine computing power. Crowdsourced testing has become a fairly new trend in software engineering due to its cost-effectiveness, impartiality, diversity, and high device and configuration coverage [15].

To investigate the potential of crowdsourced usability testing, the similar laboratory usability testing is simultaneously performed as the comparison in distinct implementation [29]. Crowdsourced testing is also introduced to perform costly GUI testing. The system under test is run in virtual machines and crowdsourced workers are recruited to remotely perform semi-automated continuous testing [12]. Aiming to validate whether oracle problems can be resolved by crowdsourced testing, the problems are split into subtasks and solved by a group of workers online [38]. A tool for crowdsourced testing is developed to efficiently recruit larger amounts of workers and evaluate the usability of web sites and web-based services under many different conditions [35]. Although crowsourced testing gains a great success, there are still a number of challenges. An empirical evaluation is conducted to investigate whether crowdsourced testing and laboratory testing can compensate each other [15].

Some studies concentrate on addressing problems existing in crowdsourced testing. To overcome the shortcomings (e.g., quality, management) in crowdsourced testing, Quasi-Crowdsourced Testing (QCT) is put forward by introducing crowdsourced testing to education platforms [8]. To help developers inspect test reports more quickly, a text-based technique called DivRisk is explored by combining a diversity strategy and a risk strategy to prioritize test reports [13]. Given the ambiguity of natural language, a hybrid analysis technique is proposed by leveraging both textual information and image information [14]. To reduce unnecessary inspection on false positive test reports, some researchers adopt a cluster-based classification approach to discriminate the true positives from a large number of test reports [49]. However, this method often requires abundant manually labelled training data. To overcome this obvious drawback, some researchers consider to use more the efficient activity learning technique [50]. Likewise, our study aims to cluster test reports to reduce the cost of manual inspection.

8.4 Fuzzy Clustering for Documents

Document clustering plays an important role in information retrieval and text processing, which can help users systematically organize and manage documents. The most common document clustering is hard clustering, i.e., each document is deterministically partitioned into a single cluster. However, a document may often contain multiple topics or themes (e.g., a multi-bug test report in this study) and can be partitioned into multiple clusters by some probabilities, which is called fuzzy document clustering [37, 53] or soft document clustering [23, 28].

There are many algorithms focusing on fuzzy document clustering. FCM is possibly the most popular fuzzy clustering algorithm and has been successfully applied to various fuzzy document clustering problems [33], which iteratively updates cluster prototypes and the fuzzy membership of objects to all clusters in minimizing the squared error objective function [40]. Differing from FCM, fuzzy k-Medoid algorithms employ actual objects as cluster prototypes [24]. Possibilistic C-Means (PCM) algorithms relax the constraint that the sum of membership values of a object to all clusters equals 1 [2]. Fuzzy k-means algorithms introduce a penalty term to minimize a different objective function [54]. In contrast, some other fuzzy clustering algorithms have been proposed for fuzzy document clustering. Fuzzy Relational Eigenvector Centrality-based Clustering Algorithm (FRECCA) constructs a graph representation of objects in which nodes denote objects and weighted edges denote similarities of objects for fuzzy clustering of sentence-level text [45]. Fuzzy Ontological Document Clustering (FODC) combining ontological knowledge representation and fuzzy logic control provides a better solution to patent knowledge clustering [48]. Fuzzy Latent Semantic Clustering (FLSC) is proposed to mine the latent semantics in web documents [9].

FULTER can be viewed as a special form of fuzzy document clustering in which multi-bug test reports should be deterministically partitioned into multiple clusters. The above methods are not suitable for resolving FULTER without considering multi-bug test reports. Given domain knowledge, we explore a fuzzy version of two-phase merging algorithm based on agglomerative hierarchical clustering.

9 CONCLUSION AND FUTURE WORK

In this paper, we proposed a novel fuzzy clustering framework named TERFUR to cluster crowdsourced test reports for reducing the cost of manual inspection. Aiming to reduce an unnecessary inspection, TERFUR first constructs a filter which leverages the null rule and the regular rule to filter out invalid test repots. Then, a preprocessor is built to process test reports by NLP techniques and selectively enhance the descriptions of test reports with the inputs, thus the similarities between relevant test reports are more accurate. Finally, TERFUR uses a two-phase merging algorithm to implement fuzzy clustering for crowdsourced test reports. We collect five crowdsourced test report datasets to evaluate the effectiveness of TERFUR. The experimental results show that TERFUR can cluster redundant test reports with high accuracy and significantly outperform comparative methods. In addition, experimental results also demonstrate that TERFUR can greatly reduce the cost of test report inspection in prioritizing test reports. In the future, we will deploy the framework for our industrial partners and continue to collect crowdsourced test reports for validating our framework.

ACKNOWLEDGMENTS

The authors would like to thank the students who make a good effort on annotating the crowdsourced test reports for Apps. The authors also greatly thank developers who devote their time on determining the final datasets.

REFERENCES

[1] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2015. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In *Advances in Service-Oriented and Cloud Computing - Workshops (ESOCC'15)*. Springer, Taormina, Italy, 201–215.

[2] Mauro Barni, Vito Cappellini, and Alessandro Mecocci. 1996. Comments on "A Possibilistic Approach to Clusterin". *IEEE Trans. Fuzzy Systems* 4, 3 (1996), 393–396.

[3] Kevin Bartz, Jack W. Stokes, John C. Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihle. 2008. Finding Similar Failures Using Callstack Similarity. In *Proceedings of Third Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML'08)*. USENIX Association, San Diego, CA, USA.

[4] Pavel Berkhin. 2006. A Survey of Clustering Data Mining Techniques. In *Grouping Multidimensional Data - Recent Advances in Clustering*. 25–71.

[5] James C Bezdek. 2013. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Springer Science & Business Media.

[6] Gloria Bordogna and Gabriella Pasi. 2009. Hierarchical-Hyperspherical Divisive Fuzzy C-Means (H2D-FCM) Clustering for Information Retrieval. In *2009 IEEE/WIC/ACM International Conference on Web Intelligence (WI'09)*. IEEE Computer Society, Milan, Italy, 614–621.

[7] Yguaratã Cerqueira Cavalcanti, Eduardo Santana de Almeida, Carlos Eduardo Albuquerque da Cunha, Daniel Lucrédio, and Silvio Romero de Lemos Meira. 2010. An Initial Study on the Bug Report Duplication Problem. In *14th European Conference on Software Maintenance and Reengineering (CSMR'10)*. IEEE Computer Society, Madrid, Spain, 264–267.

[8] Zhenyu Chen and Bin Luo. 2014. Quasi-crowdsourcing Testing for Educational Projects. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, Hyderabad, India, 272–275.

[9] I-Jen Chiang, Charles Chih-Ho Liu, Yi-Hsin Tsai, and Ajit Kumar. 2015. Discovering Latent Semantics in Web Documents Using Fuzzy Clustering. *IEEE Trans. Fuzzy Systems* 23, 6 (2015), 2122–2134.

[10] Hung Chim and Xiaotie Deng. 2008. Efficient Phrase-Based Document Similarity for Clustering. *IEEE Trans. Knowl. Data Eng.* 20, 9 (2008), 1217–1229.

[11] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity. In *34th International Conference on Software Engineering (ICSE'12)*. IEEE Computer Society, Zurich, Switzerland, 1084–1093.

[12] Eelco Dolstra, Raynor Vliegendhart, and Johan A. Pouwelse. 2013. Crowdsourcing GUI Tests. In *Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST'13)*. IEEE Computer Society, Luxembourg, 332–341.

[13] Yang Feng, Zhenyu Chen, James A. Jones, Chunrong Fang, and Baowen Xu. 2015. Test Report Prioritization to Assist Crowdsourced Testing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, Bergamo, Italy, 225–236.

[14] Yang Feng, James A. Jones, Zhenyu Chen, and Chunrong Fang. 2016. 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*. 202–213.

[15] Fabio Guaiani and Henry Muccini. 2015. Crowd and Laboratory Testing, Can They Co-exist? An Exploratory Study. In *2nd IEEE/ACM International Workshop on CrowdSourcing in Software Engineering (CSI-SE'15)*. IEEE Computer Society, Florence, Italy, 32–37.

[16] Khaled M. Hammouda and Mohamed S. Kamel. 2004. Efficient Phrase-Based Document Indexing for Web Document Clustering. *IEEE Trans. Knowl. Data Eng.* 16, 10 (2004), 1279–1296.

[17] Jeff Howe. 2006. The Rise of Crowdsourcing. *Wired magazine* 14, 6 (2006), 1–4.

[18] Michael Httermann. 2012. *DevOps for Developers*. Apress.

[19] A.K. Jain, M.N. Murty, and P.J. Flynn. 1999. Data Clustering: A Review. *ACM computing surveys (CSUR)* 31, 3 (1999), 264–323.

[20] Jung-Yi Jiang, Ren-Jia Liou, and Shie-Jue Lee. 2011. A Fuzzy Self-Constructing Feature Clustering Algorithm for Text Classification. *IEEE Trans. Knowl. Data Eng.* 23, 3 (2011), 335–349.

[21] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2011. Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'11)*. IEEE Compute Society, Hong Kong, China, 486–493.

[22] B. Kirubakaran and Dr. V. Karthikeyani. 2013. Mobile Application Testing - Challenges and Solution Approach through Automation. In *Proceedings of the 2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering (PRIME'13)*. IEEE Computer Society, Salem, India.

[23] Ravikumar Kondadadi and Robert Kozma. 2002. A Modified Fuzzy ART for Soft Document Clustering. In *Proceedings of the 2002 International Joint Conference on Neural Networks (IJCNN'02)*. IEEE Computer Society, Honolulu, Hawaii, 2545–2549.

[24] Raghu Krishnapuram, Anupam Joshi, and Liyu Yi. 1999. A Fuzzy Relative of the K-Medoids Algorithm with Application to Web Document and Snippet Clustering. In *IEEE InternationalFuzzy Systems Conference Proceedings (FUZZ-IEEE'99)*. IEEE Computer Society, Seoul, Korea, 1281–1286.

[25] Yves Ledru, Alexandre Petrenko, and Sergiy Boroday. 2009. Using String Distances for Test Case Prioritisation. In *24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*. IEEE Computer Society, Auckland, New Zealand, 510–514.

[26] Yanjun Li, Soon M. Chung, and John D. Holt. 2008. Text Document Clustering Based on Frequent Word Meaning Sequences. *Data Knowl. Eng.* 64, 1 (2008), 381–404.

[27] Yanjun Li, Congnan Luo, and Soon M. Chung. 2008. Text Clustering with Feature Selection by Using Statistical Data. *IEEE Trans. Knowl. Data Eng.* 20, 5 (2008), 641–652.

[28] King-Ip Lin and Ravikumar Kondadadi. 2001. A Similarity-Based Soft Clustering Algorithm for Documents. In *Proceedings of the 7th International Conference on Database Systems for Advanced Applications (DASFAA'01)*. IEEE Computer Society, Hong Kong, China, 40–47.

[29] Di Liu, Randolph G. Bias, Matthew Lease, and Rebecca Kuipers. 2012. Crowdsourcing for Usability Testing. *Proceedings of the American Society for Information Science and Technology* 49, 1 (2012), 1–10.

[30] Christopher D. Manning and Hinrich Schütze. 2001. *Foundations of statistical natural language processing.* Cambridge: MIT Press.

[31] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. 2015. A Survey of the Use of Crowdsourcing in Software Engineering. *RN* 15, 01 (2015).

[32] Louis Massey. 2005. Evaluating and Comparing Text Clustering Results. In *International Conference on Computational Intelligence ('05)*. IASTED/ACTA Press, Alberta, Canada, 85–90.

[33] MES Mendes and Lionel Sacks. 2004. Dynamic Knowledge Representation for E-learning Applications. In *Enhancing the Power of the Internet.* Springer, 259–282.

[34] Natwar Modani, Rajeev Gupta, Guy M. Lohman, Tanveer Fathima Syeda-Mahmood, and Laurent Mignet. 2007. Automatically Identifying Known Software Problems. In *Proceedings of the 23rd International Conference on Data Engineering Workshops (ICDE'07)*. IEEE Computer Society, Istanbul, Turkey, 433–441.

[35] Michael Nebeling, Maximilian Speicher, Michael Grossniklaus, and Moira C. Norrie. 2012. Crowdsourced Web Site Evaluation with CrowdStudy. In *Proceedings of 12th International Conference on Web Engineering (ICWE'12)*. Springer, Berlin, Germany, 494–497.

[36] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. 2012. Duplicate Bug Report Detection with A Combination of Information Retrieval and Topic Modeling. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*. ACM, Essen, Germany, 70–79.

[37] Rahul R Papalkar and G Chandel. 2013. Fuzzy Clustering in Web Text Mining and Its Application in IEEE Abstract Classification. *International Journal of Computer Sciences and Management Research* 2, 2 (2013), 1529–1533.

[38] Fabrizio Pastore, Leonardo Mariani, and Gordon Fraser. 2013. CrowdOracles: Can the Crowd Solve the Oracle Problem?. In *Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST'13)*. IEEE Computer Society, Luxembourg, 342–351.

[39] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. 2003. Automated Support for Classifying Software Failure Reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Compute Society, Portland, Oregon, USA, 465–477.

[40] M.E.S. Mendes Rodrigues and L. Sacks. 2004. A Scalable Hierarchical Fuzzy Clustering Algorithm for Text Mining. In *Proceedings of the 5th international conference on recent advances in soft computing (RASC'04)*. IEEE Computer Society, Nottingham, UK, 269–274.

[41] Guoping Rong, He Zhang, and Dong Shao. 2016. CMMI Guided Process Improvement for DevOps Projects: An Exploratory Case Study. In *Proceedings of the International Conference on Software and Systems Process (ICSSP'16)*. ACM, Austin, Texas, USA, 76–85.

[42] Dmitri Roussinov and Hsinchun Chen. 1999. Document Clustering for Electronic Meetings: An Experimental Comparison of Two Techniques. *Decision Support Systems* 27, 1-2 (1999), 67–79.

[43] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. 2007. Detection of Duplicate Defect Reports Using Natural Language Processing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE Computer Society, Minneapolis, MN, USA, 499–510.

[44] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo Juzgado. 2015. Are Students Representatives of Professionals in Software Engineering Experiments?. In *37th IEEE/ACM International Conference on Software Engineering (ICSE'15)*. IEEE Computer Society, Florence, Italy, 666–676.

[45] Andrew Skabar and Khaled Abdalgader. 2013. Clustering Sentence-Level Text Using a Novel Fuzzy Relational Clustering Algorithm. *IEEE Trans. Knowl. Data Eng.* 25, 1 (2013), 62–75.

[46] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards More Accurate Retrieval of Duplicate Bug Reports. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*. IEEE Computer Society, KS, USA, 253–262.

[47] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on*

Software Engineering - Volume 1 (ICSE'10). ACM, Cape Town, South Africa, 45–54.

[48] Amy J. C. Trappey, Charles V. Trappey, Fu-Chiang Hsu, and David W. Hsiao. 2009. A Fuzzy Ontological Knowledge Document Clustering Methodology. *IEEE Trans. Systems, Man, and Cybernetics, Part B* 39, 3 (2009), 806–814.

[49] Junjie Wang, Qiang Cui, Qing Wang, and Song Wang. 2016. Towards Effectively Test Report Classification to Assist Crowdsourced Testing. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, September 8-9, 2016*. 6:1–6:10.

[50] Junjie Wang, Song Wang, Qiang Cui, and Qing Wang. 2016. Local-based Active Classification of Test Report to Assist Crowdsourced Testing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 190–201.

[51] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An Approach to Detecting Duplicate Bug reports Using Natural Language and Execution Information. In *30th International Conference on Software Engineering (ICSE'08)*. ACM, Leipzig, Germany, 461–470.

[52] Johannes Wettinger, Vasilios Andrikopoulos, and Frank Leymann. 2015. Automated Capturing and Systematic Usage of DevOps Knowledge for Cloud Applications. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E) (IC2E'15)*. IEEE Computer Society, New York, NY, 60–65.

[53] Thaung Thaung Win and LinMon. 2010. Document Clustering by Fuzzy C-Mean Algorithm. In *34th International Conference and Exposition on Advanced Ceramics and Composites (ICACC'10)*. IEEE Computer Society, Daytona Beach, FL, USA, 239–242.

[54] Jinglin Xu, Junwei Han, Kai Xiong, and Feiping Nie. 2016. Robust and Sparse Fuzzy K-Means Clustering. In *25th Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'2016)*. IJCAI/AAAI Press, New York, USA, 2224–2230.

[55] Jifeng Xuan, He Jiang, Yan Hu, Zhilei Ren, Weiqin Zou, Zhongxuan Luo, and Xindong Wu. 2015. Towards Effective Bug Triage with Software Data Reduction Techniques. *IEEE Trans. Knowl. Data Eng.* 27, 1 (2015), 264–280.

[56] Jifeng Xuan, He Jiang, Zhilei Ren, Jun Yan, and Zhongxuan Luo. 2010. Automatic Bug Triage using Semi-Supervised Text Classification. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'10)*. Knowledge Systems Institute Graduate School, Zurich, Switzerland, 209–214.

[57] Jifeng Xuan, He Jiang, Zhilei Ren, and Weiqin Zou. 2012. Developer Prioritization in Bug Repositories. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE Computer Society, Redwood City, San Francisco Bay, CA, USA, 25–35.

[58] Rongrong Zhang, Qingtian Zeng, and Sen Feng. 2010. Data Query Using Short Domain Question in Natural Language. In *2010 IEEE 2nd Symposium on Web Society (SWS'10)*. IEEE Computer Society, Beijing, China, 351–354.

[59] Wen Zhang, Taketoshi Yoshida, Xijin Tang, and Qing Wang. 2010. Text Clustering Using Frequent Itemsets. *Knowl.-Based Syst.* 23, 5 (2010), 379–388.

Received September 2016; revised February 2017; accepted June 2017