Test Report Prioritization to Assist Crowdsourced Testing

Yang Feng[1,2], Zhenyu Chen[1], James A. Jones[2], Chunrong Fang[1], Baowen Xu[1]
[1]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[2]Department of Informatics, University of California, Irvine, USA
zychen@software.nju.edu.cn
{yang.feng, jajones}@uci.edu

ABSTRACT

In crowdsourced testing, users can be incentivized to perform testing tasks and report their results, and because crowdsourced workers are often paid per task, there is a financial incentive to complete tasks quickly rather than well. These reports of the crowdsourced testing tasks are called "test reports" and are composed of simple natural language and screenshots. Back at the software-development organization, developers must manually inspect the test reports to judge their value for revealing faults. Due to the nature of crowdsourced work, the number of test reports are often difficult to comprehensively inspect and process. In order to help with this daunting task, we created the first technique of its kind, to the best of our knowledge, to prioritize test reports for manual inspection. Our technique utilizes two key strategies: (1) a diversity strategy to help developers inspect a wide variety of test reports and to avoid duplicates and wasted effort on falsely classified faulty behavior, and (2) a risk strategy to help developers identify test reports that may be more likely to be fault-revealing based on past observations. Together, these strategies form our **DivRisk** strategy to prioritize test reports in crowdsourced testing. Three industrial projects have been used to evaluate the effectiveness of these methods. The results of the empirical study show that: (1) **DivRisk** can significantly outperform random prioritization; (2) **DivRisk** can approximate the best theoretical result for a real-world industrial mobile application. In addition, we provide some practical guidelines of test report prioritization for crowdsourced testing based on the empirical study and our experiences.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Crowdsourcing testing, test report prioritization, natural language processing, test diversity

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786862

1. INTRODUCTION

The idea of crowdsourced testing has gained recent attention in the software-engineering community (*e.g.,* [7, 23, 29, 37]). In crowdsourced testing, crowdsourced workers are given testing tasks to perform, and the workers choose their tasks, perform them, and submit a *test report* of the behavior of the system. In this way, the crowdsourced workers help the centralized developers reveal faults. In order to attract workers, testing tasks are often encoded in interesting tasks (such as games [4]), or testing tasks can be financially compensated. In this paradigm, the workers perform the tasks and then submit their test reports, which are simple and informal descriptions of the behavior of the software system. These test reports are composed of natural-language descriptions, sometimes accompanied with screenshots, and an assessment as to whether the worker believes that the software behaved correctly (*i.e.,* "passed" in testing parlance) or behaved incorrectly (*i.e.,* "failed") In the latter case, the test report can serve as an informal bug report.

At the software-development organization, developers and testers receive the crowdsourced test reports, and subsequently judge their merit, attempt to recreate failures, and debug any true faults. Naturally, this task can be time-consuming and tedious even in traditional testing scenarios, but moreover, in a crowdsourced setting the number test reports can be prohibitive. Further, crowdsourced workers may choose to perform many smaller and less-impactful tasks. In practice, it is often impossible to manually inspect all test reports in a limited time.

Past research has produced test-case prioritization techniques (*e.g.,* [10, 12, 14, 18, 22, 27, 33, 39–42]) in which test *cases* from a regression test suite are prioritized so that they execute in an order that reveals faults earlier. Although such techniques do not address the concept of prioritizing test *reports*, these techniques' motivating principles provide inspiration for our approach to prioritizing test reports: namely, diversification of test behavior to help identify multiple faults.

Another body of existing research attempts to automatically classify bug reports in a bug-reporting system by their level of severity (*e.g.,* [2, 9, 24, 30, 35, 38]). Although such techniques do not address the concept of prioritizing test reports, which are less structured and report both passing and failing behavior, they inspire another motivating principle of our approach: namely, recognizing patterns of risk factors that may foretell reports that reveal faults.

In this paper, we propose a strategy to prioritize test reports for use in crowdsourced testing. We adopt natural

language processing (NLP) techniques, word segmentation, and synonym replacement, to extract keywords from test reports. These keywords are used to predict failure risks of tests and calculate distances of test reports. We design two single prioritization strategies: the risk strategy (**Risk**) and the diversity strategy (**Div**). The risk strategy is designed to dynamically select the most risky test report for inspection in each iteration. As such, its purpose is to prioritize test reports most likely to reveal faults. The diversity strategy is designed to select the diversified test reports for inspection by maximizing the distances to already inspected test reports. As such, its purpose is to prioritize test reports most unlike already inspected reports to avoid redundant work and reveal different faults. Finally, in order to reveal as many faults as possible as early as possible, we combine the two strategies to a hybrid prioritization strategy (**DivRisk**).

In 2013, we conducted three projects that used crowdsourced testing with our industry partner, Baidu[1] [7]. The three projects were used to evaluate the effectiveness of test-report prioritization methods. The average percentage of faults detected (APFD) [34] and the fault-detection rate were introduced to compare four test-report prioritization methods: **Random**, **Risk**, **Div** and **DivRisk**. The **Best** and the **Worst** theoretical results of test-report prioritization were computed to discover the room for improvement of our prioritization methods. The results of empirical study indicate that: (1) **DivRisk** can outperform the random prioritization technique significantly (14.29%–34.52% improvement in terms of the APFD metric); (2) **DivRisk** can approximate the **Best** theoretical result (the gap is only 7.07% in terms of APFD) of crowdsourced testing for the mobile application testing.

The main contributions of this paper are as follows.

(1) To the best of our knowledge, this is the first work to identify the problem of test report prioritization for crowdsourced testing. Our practical experiences lead us to believe that this is an important problem in crowdsourced testing.
(2) A novel feature of our test report prioritization method **DivRisk** is to combine the risk strategy and the diversity strategy.
(3) Three industrial projects of crowdsourced testing are used to evaluate test report prioritization methods. Some practical guidelines of test report prioritization in crowdsourced testing are summarized.

2. BACKGROUND & MOTIVATION

In this section, we describe some background of crowdsourced testing to motivate the prioritization of test reports and describe our experience that motivates our prioritization strategies. We also introduce an example test report, which is used to demonstrate how our prioritization techniques work in next section.

Crowdsourced Testing Study. In 2013, we conducted three projects of crowdsourced testing with Baidu [7]. Figure 1 shows the procedure of crowdsourced testing in our projects. Testers in Baidu prepared packages for crowdsourced testing: software under test and testing tasks. Testing tasks were divided into some sub-tasks. The packages were distributed online, and workers bid on testing tasks.

[1]Baidu is the largest Chinese search service provider.

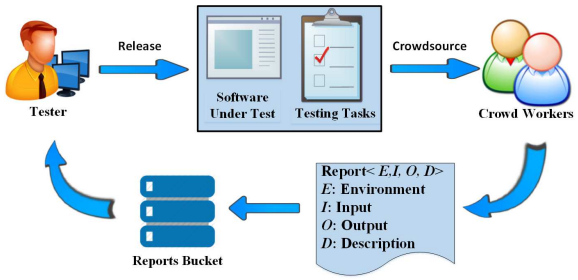


Figure 1: The procedure of crowdsourced testing

Workers were required to complete tasks in a limited time (3–5 days in our projects). Then workers submitted test reports online. Workers submitted thousands of test reports due to financial incentive and other motivations. These test reports had many false positive results (32%–80% in our projects), *i.e.,* a test report marked as "failed" that actually described correct behavior. Test reports also contained many redundant behaviors, because workers preferred to reveal simple faults instead of complex faults. Testers manually inspected all test reports to judge the workers' performance, *i.e.,* their values for revealing faults. This was a time-consuming and tedious process (nearly 12 days in our projects). Hence, it motivated us to prioritize test reports to improve the effectiveness of inspection in crowdsourced testing.

The three software systems on which we conducted crowdsourced testing are as follows:

P1: The first project is Baidu-Input[2] on Android, which can support several input methods. Testers in Baidu provide 10 functionality sets. One crowd worker can select one functionality set, and each functionality set can be selected by at most two crowd workers, who use different mobile phones and different versions of Android.
P2: The second project is Baidu-Browser[3], which is a web browser. Testers in Baidu provide seven functionality sets for regression testing. One crowd worker can select three functionality sets.
P3: The third project is Baidu-Player[4], which is a multimedia player. Testers in Baidu provide three performance testing scenarios. One crowd worker should cover all of these three scenarios.

Workers can report other problems, such as usability and compatibility problems, in test reports.

Example Test Report. Figure 2 shows a test report committed by a worker in P2[5]. Workers are required to commit test reports in a same format to facilitate processing them by testers. A test report is a four tuple $\{E, I, O, D\}$, in which E, I, O and D are as follows.

- E is test environment, including hardware and software configuration, etc.

[2]http://shurufa.baidu.com/
[3]http://liulanqi.baidu.com/
[4]http://player.baidu.com/
[5]Test reports are written in Chinese in our projects. In Figure 2, we translate them into English to facilitate understanding.

Environment (E)	Input (I)	Output (O)	Description (D)
Operating System: Windows 7-64-SP1 OS Version No: MS Windows 6.1.7601 System Language: Chinese Screen Resolution: 1920x1080	Select menu→options in the browser , set "ad block" closed in the Security page, open the link "http://www.qidian.com/Default.aspx", and floating ads or ads around the edge of the web page are found; select menu→options in the browser, set "ad block" enhanced in the security page, open the link to check; switch the browser mode, refresh the page to check again.	Screenshots: 	When the blocking mode is switched, the number of blocked ads is not consistent with the previous one.

Figure 2: Example test report from P2

- I is test input, including input data and operation steps.
- O is test output, some screenshots.
- D is test result description, any information for understanding faults.

Experience and Lessons. The crowdsourced workers submitted over 2000 test reports. Of these submitted test reports, 757 were labeled as "failed" and as such were gathered for manual inspection. Upon manual inspection of all test reports that were labeled as failed, 462 of the 757 failed test reports were false positives. In other words, 462 out of 757 test reports described behavior that was either correct behavior or behavior that was considered outside the behavior of the studied software system (*e.g.,* external problems such as advertisements).

As an example, the test report in Figure 2 provides an example of an actual submitted test report from the study. The test report describes the problem of some inconsistent advertisements in different modes. The test report is false positive, because it is not a fault of Baidu-Browser, but instead of the advertisement host site.

Through informal and extensive discussions with professional test engineers at Baidu, a number of observations and lessons were learned:

1. The number of test reports submitted by crowdsourced workers quickly became challenging to manually inspect. A larger crowdsourced testing session would have produced prohibitively many reports to manually inspect.
2. The number of false positives were more numerous than would have been expected, and presented challenges for inspection.
3. Many of the true positives and false positives were duplicates of the same underlying behavior.
4. Many crowdsourced workers performed many easy tasks and reported shallow bugs, presumably due to the incentive structures that reward quantity of submitted reports.
5. The word choice among the true positive and false positive test reports were sufficiently consistent, when accounting for word variations and synonyms.

Based on these observations by test engineers at Baidu and informed by our discussions with them, we attempted to assist with the processing and inspection of test reports, particularly for the scenario of crowdsourced testing for which the plethora of reports would be even greater. Lessons 1 and 2 simply motivate the need for some automated assistance. Lessons 3 and 4 motivate the need for looking for *diversity* in test reports — test reports that are duplicate (whether true positives or false positives) present the opportunity for wasted inspection effort and delayed identification of new true faults. Lesson 5 motivates the use of natural-language techniques to categorize test reports in an effort to automatically infer duplicate test reports.

As such, our experiences and interactions with our industrial partners motivate us to use natural language techniques (*i.e.,* NLP) to cluster test reports. Lessons 3 and 4 have motivated the need to prioritize these clusters to account for diversity (*i.e.,* our **Div** strategy).

However, because the goal of such prioritization is to reveal as many faults as early as possible, we have incorporated an additional strategy that we are calling **Risk**. The **Risk** strategy learns from already inspected test reports that were manually assessed as *true positive*, *i.e.,* true failures that revealed true faults in the software system under test. As such, the **Risk** strategy guides the prioritization order toward other test report clusters that include similar words.

Finally, we note and recognize that the motivations for **Div** and **Risk** are, in a way, at odds — **Div** seeks to find the next test-report cluster most dissimilar from the already inspected ones, whereas **Risk** seeks to find the next test-report cluster most similar to already-inspected true positives. To account for these contrasting motivations, we created a hybrid strategy, **DivRisk**, that incorporates both **Div** and **Risk** to both maximize the distance from inspected test reports (and thus reduce inspection of duplicates and false positives) and guide the search toward riskier software behavior (and thus increase discovery of new true positives).

3. METHODOLOGY

In this section, we present our test report prioritization methods in detail. Figure 3 shows the framework of test report prioritization, which mainly contains four steps: (1) test report collection, (2) test report processing, (3) keyword vector modeling, and (4) prioritizing test reports.

3.1 Test Report Collection

In our crowdsourcing projects, all test reports were committed online by workers in *Excel* files. We predefined the format of *Excel* files, such that these test reports strictly contained four parts E, I, O, and D. I and D were used to inform keywords for use by the NLP techniques. E and O were additionally used to assist testers in test report inspection.

> **Running Example.** In order to demonstrate our methods, we sample seven test reports (I and D) in P2, as shown in Table 1. TR1, TR5, TR6 and TR7 are false positive test reports. That is, workers mark them as failed test reports, but testers inspect them and judge that they

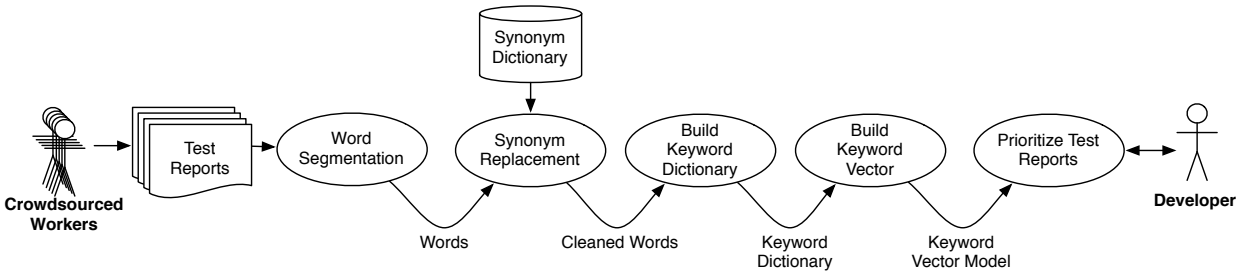


Figure 3: The framework of test report prioritization

are not. TR2 and TR4 reveal the same fault, denoted by "Fault1". TR3 reveals another fault, denoted by "Fault2". TR7 is the example in Figure 2. Please note that all test reports are written in Chinese, and our implementation is written to handle Chinese test reports. In order to facilitate understanding, we translate them into English in the paper, as shown in Table 1.

3.2 Test Report Processing

As shown in Figure 3, test report processing contains two steps: word segmentation and synonym replacement.

Word Segmentation. Word segmentation is a basic NLP task. There are many efficient tools of word segmentation for different languages [16,20]. We adopted ICTCLAS[6] for word segmentation, which is a widely used Chinese NLP platform. I and D of test reports were segmented into words marked with their Part-Of-Speech (POS) in the context, and then the POS tagging was applied. Hidden Markov models were used in the POS tagging [1]. Finally, the bi-gram model [3] was introduced to count the classes of words.

Synonym Replacement. In crowdsourced testing, test reports are committed by part-time workers or self-identified volunteers, who are often from different workplaces. Workers have different preferences of words and different habits of expression. Some words in test reports are meaningless for revealing faults. Hence, we filtered out these useless words (often referred to as "stop words" in the NLP literature). Prior studies show that verbs and nouns are most important to reflect the content of a document [31, 43]. Hence, we retained only verbs and nouns as candidate keywords of test reports and filtered out other words. Also, workers often use different words to express the same concept. For example, "thumb keyboard" and "nine-grids keyboard" refer to the same layout of keyboard in Chinese. We introduced the synonym replacement technique in NLP to alleviate this problem. In our method, we adopted the synonym library of Language Technology Platform (LTP) [6], which is largely considered as one of the best cloud-based Chinese NLP platforms.

Example. In our example, keywords are extracted from test reports, shown in Table 2. For example, "compatibility" indicates that TR1, TR2 and TR4 may report some compatibility problems; "menu" indicates that TR6 and TR7 may report some problems related to menu options.

3.3 Keyword Vector Modeling

The next step is to build the keyword vector model KV. We then create the risk vector RV and the distance matrix DM based on KV.

[6]http://ictclas.org/

Keyword Dictionary. Keywords extracted from test reports play an important role in test report prioritization. In order to summarize the information contained within the keywords, we count the frequencies (*i.e.,* the number of occurrence) of keywords. In practice, we set a threshold ε to remove some keywords with low frequency to improve the effectiveness. As a result, a keyword dictionary is built.

Example. Table 3 shows the keyword dictionary of the 7 test reports. In the example, $\varepsilon = 2$, *i.e.,* all keywords with frequency < 2 in Table 2 are removed to produce Table 3.

Keyword Vector. Based on the keyword dictionary, we construct a keyword vector for each test report $tr_i = (e_{i,1}, e_{i,2}, \cdots, e_{i,m})$, in which m is the number of keywords in keyword dictionary. We compute that $e_{i,j} = 1$ if the ith test report contains the jth keyword in keyword dictionary; and $e_{i,j} = 0$ otherwise.

Example. Table 4 shows the keyword vector model KV of the seven test reports, in which the ith row is the keyword vector of TRi, *i.e.,* $KV(i, *) = tr_i$. KV is an $n \times m$ matrix for n test reports and m keywords in keyword dictionary.

Risk Vector. Keywords in a test report reflect their values of revealing faults to some extent. For example, the most frequent word is "click" in Table 3. However, we cannot claim that "click" is the most important one for revealing faults, because "click" is a common operation in a browser. We can simply count the number of "1"s in the keyword vector as the risk value of test report, denoted by $RV(i) = \sum_{j=1}^{m} e_{i,j}$. RV is an $n \times 1$ vector for n test reports, as shown in Table 5.

Distance Matrix. Based on the keyword vector matrix KV, we can calculate the distances of each pair of test reports. In this work, we adopt the Hamming distance. That is, for two keyword vectors tr_i and tr_k, we count the number of different $e_{i,j}$ and $e_{k,j}$ in the corresponding position j, as the distance $\mathcal{D}(tr_i, tr_k)$. The inverse distance indicates the similarity of test reports.

Example. As a result, we construct an $n \times n$ distance matrix for n test reports. For example, the distance matrix of the seven test reports is shown in Table 5. $\mathcal{D}(tr_1, tr_2) = 3$, for TR1 and TR2 have 3 different keywords; $\mathcal{D}(tr_1, tr_7) = 20$, for TR1 and TR7 have 20 different keywords in the keyword dictionary.

3.4 Prioritization Strategy

In this subsection, we present three prioritization strategies: **Risk**, **Div** and **DivRisk**, based on the risk vector RV and the distance matrix DM, which are calculated based on the keyword vector model KV.

Table 4: Keyword Vector Model: KV

No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
TR1	1	0	1	0	1	0	0	1	0	1	0	1	1	0	0	0	1	0	0	0	0	0	0	1	1	1
TR2	1	0	1	0	1	0	0	1	0	1	0	1	1	0	0	0	1	0	1	0	0	0	1	1	0	1
TR3	0	0	0	1	0	0	1	0	1	0	0	0	0	1	1	0	0	1	0	0	1	1	0	1	0	0
TR4	1	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	1
TR5	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
TR6	0	1	0	0	0	0	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	0	1	1	0	0
TR7	1	1	0	0	0	1	1	0	1	0	1	0	1	1	1	1	1	1	0	1	1	1	1	1	0	0

Table 1: Seven test reports from P2

No.	I	D	Result
TR1	Login renren.com in compatibility mode, click on "Personal Homepage" or "Send a Gift"(to friends), then after clicking the back button, click the forward button.	The page content is not consistent before clicking the back button with the content after clicking the forward button.	Non-fault
TR2	Enter compatibility mode, login renren.com, click one of the friend links, then click the "Personal Homepage" button , click the back button after loading.	It can go back to friend page only after clicking the back button twice.	Fault1
TR3	Open the browser, select tools→options→security, set "ad block" enhanced , input "http://soudu.org/" in the address bar.	Ads on the lower right of the page are not blocked successfully.	Fault2
TR4	In the input box in Baidu homepage , search "group buying" in compatibility mode. Next, search "ice cream" and the "red bull", double click the back button and then click the forward button once.	The page content is not consistent before clicking the back button with the content after clicking the forward button .	Fault1
TR5	Open the browser, in maximized mode, wait for the program to load and then switch the program, which means rapid and continual full-screen switch.	Sometimes the bug appears when taskbar at the bottom of the system do not disappear, especially when open other browser simultaneously . When the system is busy, the bug is more likely to occur. Move the cursor onto some task and after the appearance of the task, the system operates correctly.	Non-Fault
TR6	Select menu→options in the browser, set "ad block" closed in the security page, open the link "http://www.narutom.com/" and pop-up ads are found while loading; select menu→options in the browser, set "ad block" enhanced in the Security page, open the link "http://www.narutom.com/" to check again.	Ads blocking failed.	Non-Fault
TR7	Select menu→options in the browser , set "ad block" closed in the Security page, open the link "http://www.qidian.com/Default.aspx" and floating ads or ads around the edge of the web page are found; select menu→options in the browser, set "ad block" enhanced in the security page, open the link to check; switch the browser mode, refresh the page to check again.	When the blocking mode is switched, the number of blocked ads is not consistent with the previous one.	Non-Fault

Risk. In order to reveal faults as early as possible, it is natural to give the top priority to inspect the most risky test report, *i.e.,* the test report TRi with the highest risk value $RV(i)$. If multiple test reports share the highest risk value, one of them is selected for inspection. Let QTR be the *ordered set* of already inspected test reports.

> **Example.** Based on the risk values in Table 5, TR7 ($RV(7) = 17$) is first selected for inspection. Then TR6 ($RV(6) = 14$) is selected for inspection, followed by TR2 ($RV(2) = 12$). At this point of processing, QTR = {TR7, TR6 TR2}.

We adopt a dynamic prioritization strategy based on the risk values and the inspection results. That is, if TRk is inspected and determined to be a true failure, all keywords of TRk in KV are increased by δ ($\delta = 0.2$ in our projects). The algorithm of updating KV is shown in Algorithm 1. Based on the new KV, the risk values in RV are updated.

Table 2: Keywords from 7 test reports

No.	Keywords
TR1	compatibility/n, mode/n, login/v, click/v, person/n, homepage/n, friend/n, gift/n, back/v, button/n, forwad/v, page/n, content/n
TR2	enter/v, compatibility/n, mode/n, login/v, click/v, friend/n, link/n, person/n, homepage/n, button/n, load/v, back/v, page/n
TR3	open/v, browser/n, tool/n, options/n, security/n, ads/n, block/v, select/v, address/n, input/v, page/n, corner/n, not/v
TR4	compatibility/n, mode/n, input/v, groupon/v, click/n, search/v, button/n, icecream/n, redbull/n, back/v, forward/v, result/n
TR5	open/v, browser/n, maximize/v, condition/n, wait/v, program/n, load/v, finish/v, do/v, switch/v, fullscreen/n, appear/v, system/n, task/n, miss/v, possibility/n, mouse/n, thumbnail/n, restore/v
TR6	browser/n, click/n, menu/n, options/n, security/n, page/n, ads/n, block/v, close/v, open/v, link/n, load/v, find/v, strength/n, check/v, fail/v
TR7	browser/n, click/n, menu/n, options/n, security/n, page/n, ads/n, block/v, closed/v, open/v, link/n, appear/v, floating/v, strength/n, check/v, switch/v, mode/n, refresh/v, button/n, select/v, change/v, number/n

Table 3: Keyword Dictionary

No.	Word	Freq.	No.	Word	Freq.
K1	button	4	K2	strength	2
K3	homepage	2	K4	input	2
K5	person	2	K6	switch	2
K7	browser	4	K8	friend	2
K9	options	3	K10	login	2
K11	check	2	K12	back	3
K13	mode	4	K14	block	3
K17	click	5	K18	ads	3
K19	load	3	K20	menu	2
K21	security	3	K22	select	2
K23	link	3	K24	page	5
K25	forward	2	K26	compatibility	3

That is, for each i, $RV(i) = \sum_{j=1}^{m} KV(i,j)$.

> **Example.** Because TR2 is determined to be a true failure, the risk values of TR1, TR3, TR4 and TR5 are updated to 13(11+0.2*10), 9.2(9+0.2*1), 8.0(7+0.2*5) and 4.2(4+0.2*1), respectively. That is, for TR1, TR3, TR4 and TR5, there are 10, 1, 5 and 1 same keywords as TR2, respectively. In this way, we can get the final prioritization result of test reports: QTR = {TR7, TR6, TR2, TR1, TR3, TR4, TR5}.

Div. The **Div** strategy is based on the diversity principle of test selection [5,8,17,27]. We prefer to select the test report tr_i with the maximal distance to QTR. Without confusion, QTR is also used to denote the set of keyword vectors $\{tr_i\}$ of already inspected test reports. The distance of tr and QTR, denoted by $\mathcal{D}(tr, QTR)$, is defined by the maximum distance between tr and each tr_i in QTR, i.e. $\mathcal{D}(tr, QTR) = Max_{tr_i \in QTR}\{\mathcal{D}(tr, tr_i)\}$.

> **Example.** We use the seven test reports to demonstrate **Div** based on the distance matrix in Table 5. Initially, the most risky test report TR7 is selected, thus QTR = {TR7}. For the next test report, since the maximum distance is $\mathcal{D}(tr_1, QTR) = 20$, TR1 is selected. Thus,

Table 5: Distance Matrix DM and Risk Vector RV

DM	TR1	TR2	TR3	TR4	TR5	TR6	TR7	RV
TR1	0	3	18	6	15	21	20	11
TR2	3	0	19	9	14	18	19	12
TR3	18	19	0	14	9	9	10	9
TR4	6	9	14	0	11	19	18	7
TR5	15	14	9	11	0	12	15	4
TR6	21	18	9	19	12	0	5	14
TR7	20	19	10	18	15	5	0	17

Algorithm 1 updateKV(KV, δ, k)

```
1: for all j do
2:   if KV(k, j) > 0 then
3:     for all i do
4:       KV(i, j) := KV(i, j) + δ
5:     end for
6:   end if
7: end for
8: return KV
```

QTR = {TR7, TR1}. And then TR5 is selected, because $\mathcal{D}(tr_5, QTR) = 15$ is the maximum distance for the remained test reports. In this way, we can get the final prioritization result of test reports: QTR = {TR7, TR1, TR5, TR3, TR4, TR6, TR2}.

DivRisk. In order to reveal faults as early as possible and as many as possible, **Risk** and **Div** are combined to a hybrid strategy **DivRisk**. The algorithm of **DivRisk** is shown in Algorithm 2. The risk value vector RV and distance matrix DM can be calculated based on KV (Line 1–2). Initially, the most risky test report is selected for inspection (Line 4–6). Then, a candidate set CTR is constructed by selecting n_c test reports with maximum distance(s) $\mathcal{D}(tr_i, QTR)$ (Line 8). The most risky test report in CTR is selected for inspection (Line 9–11). If the inspected test report is a failed one and $\delta > 0$, the keyword vector KV will be updated by Algorithm 1 and the risk value vector RV will also be updated (Line 12–15). Finally, the prioritization result QTR is returned.

Algorithm 2 DivRisk(KV, n_c, δ)

```
1: For each i, j, DM(i, j) := D(KV(i, :), KV(j, :))
2: For each i, RV(i) := Σ_{j=1}^{m} KV(i, j)
3: TR{1, 2 ··· , n}: n is the number of rows in KV
4: QTR := {TRk}: TRk with the highest risk value RV(k) in TR
5: QTR:=QTR ∪ {TRk}
6: TR:=TR − {TRk}
7: while |TR| ≠ 0 do
8:   CTR:= Select n_c reports TRi with the largest distances
     D(tr_i, QTR)
9:   Select the test report TRk with the highest risk value in CTR
     for inspection
10:  QTR:=QTR ∪ {TRk}
11:  TR:=TR − {TRk}
12:  if TRk is a failed test report by inspection AND δ > 0 then
13:    KV := updateKV(KV, δ, k)
14:    For each i, RV(i) := Σ_{j=1}^{m} KV(i, j)
15:  end if
16: end while
17: return QTR
```

Example. We use the seven test reports to demonstrate **DivRisk**. Initially, TR7 is selected for inspection, for it is most risky. QTR = {TR7}. Since the number of test reports is small in this example, we set $n_c = 2$ to facilitate demonstration. The candidate set CTR = {TR2, TR1}, for $\mathcal{D}(tr_2, QTR) = 20$ and $\mathcal{D}(tr_1, QTR) = 19$ are the two largest ones. TR2 is selected for inspection, for TR2 is more risky than TR1, i.e. $RV(2) = 12 > RV(1) = 11$. In this way, we can get the final prioritization result QTR = {TR7, TR2, TR3, TR4, TR6, TR1, TR5}.

Table 6: Summary of Test Reports

Project	P1	P2	P3
# Report	274	231	252
# F-Report	186	47	62
% F-Report	67.88%	20.35%	24.60%
# Fault	27	22	18

The hybrid strategy **DivRisk** will be reduced to the risk strategy **Risk** if $n_c \geq |TR|$, and it will be reduced to the diversity strategy **Div** if $n_c = 1$. Hence, we need to set a modest number to n_c ($n_c = 8$ in our projects) for a reasonable hybrid result.

4. EXPERIMENT DESIGN

In this study, we evaluated our test report prioritization methods: **Risk**, **Div** and **DivRisk** with three crowdsourced testing projects. In our projects, $\delta = 0.2$ and $n_c = 8$ as described above.

4.1 Comparison Baseline

In order to verify the effectiveness of our prioritization methods, three baselines for comparison are selected. The first baseline of comparison was the **Random** strategy, which is widely used in software testing. Given a set of finite number of test reports, all possible orderings of test reports could be enumerated in theory. Supposing that we know which test reports are truly fault revealing in advance, the **Best** and the **Worst** prioritization results could be determined. For example, {TR2, TR3, TR4, TR1, TR5, TR6, TR7} is one of the best prioritization results and {TR7, TR1, TR5, TR6, TR2, TR4, TR3} is one of the worst prioritization results. In order to fairly compare these prioritization methods, the experiment was repeated 50 times to collect experimental data.

4.2 Test Report

In our projects, all test reports were manually inspected by testers without any prioritization method. We carefully checked the inspection results again and get the final inspection results, as summarized in Table 6.

In Table 6, "# Report" is the number of test reports marked as failed by workers. These test reports were collected in the test report bucket. Testers inspected these test reports to judge whether they could reveal faults. "# F-Report" and "% F-Report" are the number and the percentage of test reports judged as failed ones by testers, respectively. In practice, some tests may reveal same faults. "# Fault" is the number of faults revealed by these test reports.

4.3 Research Question

In the experiment, we investigated the following research questions.

- **RQ1**: Can our prioritization methods improve the effectiveness of test report inspection?

If we have no prioritization method on-hand, testers will inspect test reports in a random order. That means, testers would be motivated to adopt a prioritization method only if it can outperform the **Random** strategy. RQ1 evaluates

the effectiveness of our prioritization methods **Risk**, **Div** and **DivRisk**.

- **RQ2**: How large is the gap between our prioritization methods and **Best**?

In practice, it is difficult to design one method that can work well in all cases. Hence, it is valuable to know the gap between the on-hand methods and the best one in theory. RQ2 evaluates the room for improvement of our prioritization methods.

4.4 Evaluation Metric

In order to measure the effectiveness of prioritization methods, we adopt the APFD (Average Percentage of Fault Detected) [34], which is a widely used evaluation metric in test case prioritization [22]. For each fault, we mark the index of the first test report which reveals it. Based on the order of the test reports and information about which test reports revealed which faults, we can calculate the APFD values to measure the effectiveness of the prioritization methods. A higher APFD value indicates a better prioritization result. That is, it can reveal more faults earlier than the other methods do. APFD is formalized as follows.

$$APFD = 1 - \frac{T_{f1} + T_{f2} + ... + T_{fM}}{n \times M} + \frac{1}{2 \times n} \quad (1)$$

in which, n denotes the number of test reports and M denotes the total number of faults revealed by all test reports. T_{fi} is the index of the first test report that reveals fault i.

APFD indicates the fault detection rate of all test reports. However, testers cannot inspect a large number of test reports in limited time. In practice, testers will stop inspecting test reports when the limited resource is used up. At that time, testers may only inspect 25% or 50% test reports. Therefore, we should evaluate how APFD varies for permutations of the same set of test reports [15]. We use the linear interpolation [22] as follows.

- M denotes the total number of faults revealed by all test reports.
- $p \in \{25\%, 50\%, 75\%\}$, the percentage used in our experiment.
- $Q = M \times p$, which is the number of faults corresponding to a percentage. Let $int(Q)$ and $frac(Q)$ be the integer part and fractional part of Q, respectively. If $frac(Q) = 0$, the linear interpolation is needed.
- i, j are the indexes of reports that reveal at least Q and Q+1 faults respectively. The linear interpolation is calculated as $i + (j - i) \times frac(Q)$

The linear interpolation value indicates the cost of testing to detect the given number of faults. Hence, a lower value of linear interpolation indicates a better prioritization result.

5. RESULT ANALYSIS

In this section, we analyze the experimental results to answer RQ1 and RQ2. The results of all prioritization methods are shown in Figure 4. Figure 4 (a, c, and e) shows the boxplots of APFD results of the three projects (P1–P3) for the 50 experimental runs. The prioritization methods are shown on the horizontal axis, and the APFD values are shown on the vertical axis. The blue horizontal line in Figure 4 (a, c, and e) denotes the **Best** APFD value, in theory, for that subject. Figure 4 (b, d, and f) shows the average growth curves of the three projects (P1–P3). The percentage of the inspected test reports is shown on the horizontal axis, the the percentage of revealed faults is shown on the vertical axis.

Table 7: Bonferroni Means Separation Tests

Method	APFD Means	Improvement $\frac{X-Random}{Random}$	Gap $\frac{Best-X}{X}$
P1: $F(3, 200) = 1549.27$, p-value ≤ 0.0001			
DivRisk	0.8879	29.66%	7.07%
Div	0.8094	18.20%	17.46%
Risk	0.7639	11.55%	24.45%
Random	0.6848	—	38.83%
Best	0.9507	38.83%	—
P2: $F(3, 200) = 474.15$, p-value ≤ 0.0001			
DivRisk	0.8113	34.52%	17.39%
Div	0.7167	18.84%	32.89%
Risk	0.7158	18.69%	33.05%
Random	0.6031	—	57.92%
Best	0.9524	57.92%	—
P3: $F(3, 200) = 90.42$, p-value ≤ 0.0001			
DivRisk	0.7686	14.29%	25.46%
Risk	0.7165	6.54%	34.58%
Div	0.6962	3.52%	38.51%
Random	0.6725	—	43.39%
Best	0.9643	43.39%	—

5.1 Addressing RQ1

RQ1: Can our prioritization methods improve the effectiveness of test report inspection?

Based on the results shown in Figure 4 (a, c, and e), we can find that all of our prioritization methods outperform **Random**. In particular, **DivRisk** can outperform **Random** significantly. The hybrid strategy **DivRisk** can also improve the single strategies **Risk** and **Div**. Moreover, the box-plots show that our methods are substantially more stable than **Random**. Figure 4 (b, d, and f) show the average growth curves. The line charts in Figure 4 (b and d) show that **DivRisk** presents smooth curves to the top (**Best**).

In order to further investigate our test report prioritization methods, we do Bonferroni means separation tests for all results in Table 7. All F-values are very large and the all p-values are much smaller than 0.001 in Table 7. Compared with the **Random** strategy, the percentage of improvement of **DivRisk** ranges 14.29%–34.52%. In summary, the experimental results are encouraging for the use of the hybrid **DivRisk** strategy in practice.

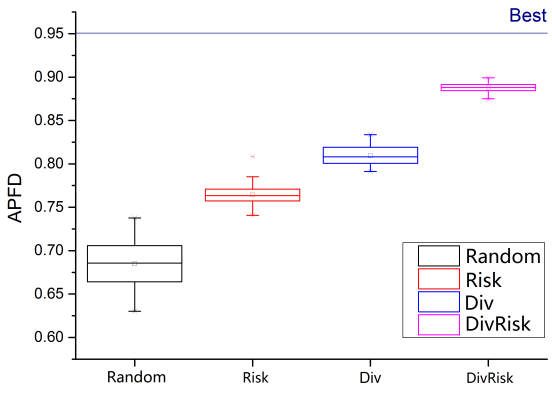
In summary, we find that our prioritization methods can improve the effectiveness of test report inspection.

5.2 Addressing RQ2

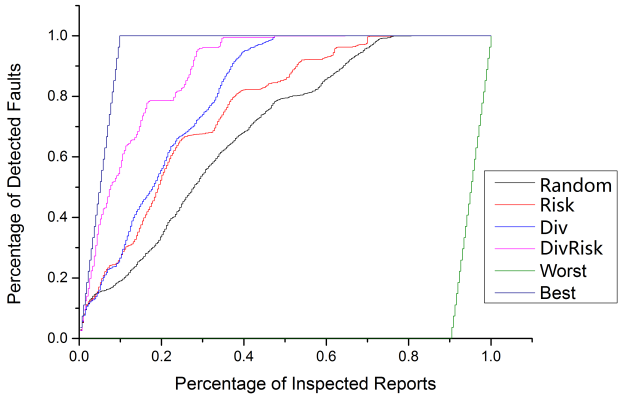
RQ2: How large is the gap between our prioritization methods and **Best**?

Figure 4 shows that the hybrid strategy **DivRisk** provides the best approximation of the **Best** result in P1 and P2. For P3, **DivRisk** provides one of the best results, but there is a larger gap between its results and the **Best** result than we found for P1 and P2. For more details, we can observe the growth curves in Figure 4. The curves of **Best** grow very fast. The curves of **DivRisk** reach the curves of **Best** when we have inspected nearly 30% test reports in P1–P2 and nearly 60% test reports in P3.

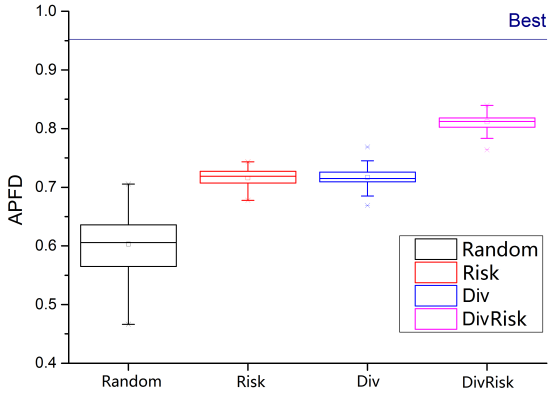
Table 7 shows the gaps between our prioritization methods and **Best**. The gap between **DivRisk** and **Best** on P1 is small (7.07%). Please recall that the results of **Best** are



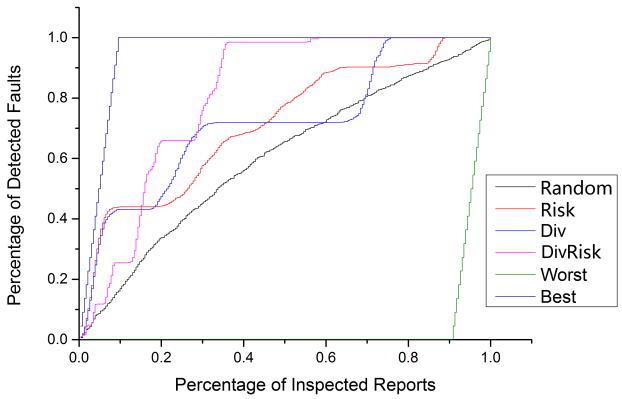
(a) APFD on P1



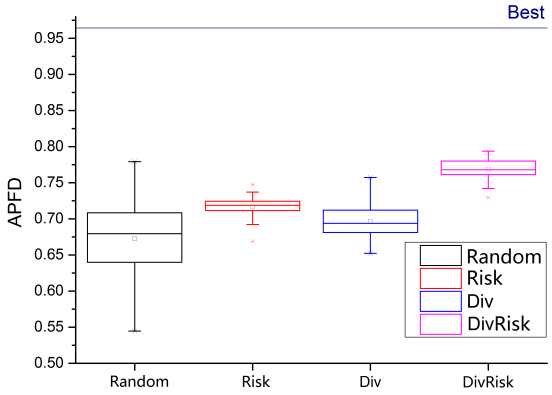
(b) Average Fault Detection Rates on P1



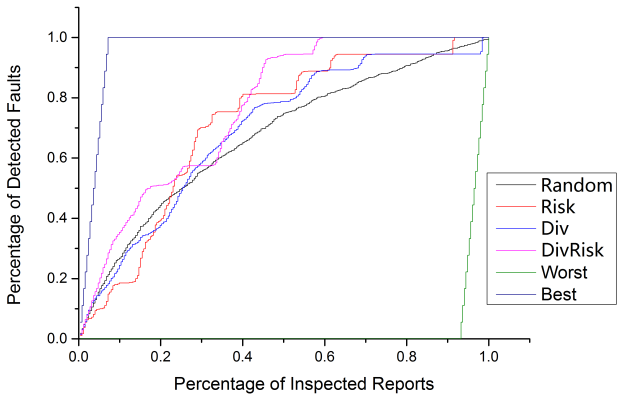
(c) APFD on P2



(d) Average Fault Detection Rates on P2



(e) APFD on P3



(f) Average Fault Detection Rates on P3

Figure 4: Experimental Results (50 times)

Table 8: Linear Interpolation (the average number of inspected test reports)

Pro.	Tech.	25%	50%	75%	100%
P1	Random	35.34	75.83	116.96	196.6
	Risk	21.12	51.64	94.19	190.7
	Div	22.39	46.80	81.27	123.5
	DivRisk	**8.885**	**20.38**	**43.09**	**99.20**
	Best	6.750	13.50	20.25	27.00
P2	Random	33.37	74.86	138.3	217.5
	Risk	**8.780**	56.22	106.4	201.2
	Div	9.200	47.46	121.2	170.1
	DivRisk	21.97	**36.24**	**66.93**	**98.30**
	Best	5.500	11.00	16.50	22.00
P3	Random	22.25	61.72	122.2	226.8
	Risk	32.90	57.14	**83.01**	230.2
	Div	23.88	61.16	104.4	246.4
	DivRisk	**14.90**	**42.44**	95.94	**145.3**
	Best	4.500	9.000	13.50	18.00

purely hypothetical and based on an unrealistically omniscient best-case analysis. Hence the result of **DivRisk** may be, or at least approximate, the best one in practice. The gaps on P2 and P3 may be, thus, acceptable (17.79% and 25.49%) in practice, and moreover, do improve the ordering of unordered or random ordering.

In order to explain the results more clearly, we calculate the linear interpolations shown in Table 8. Table 8 shows the average numbers of inspected test reports in the cases of detecting 25%, 50%, 75% and 100% faults. If we need to reveal 25% or 50% faults, **DivRisk** is near to **Best**. However, if we need to reveal more faults, there may be room for additional improvement. A strange phenomenon is worthy of attention: **Risk** outperforms **DivRisk** for the 25% level of inspected faults faults for P2 and the 75% level of inspected faults for P3. This result may be due to the heuristic nature of these methods and will be a subject of additional investigation in the future.

*In summary, we find that our prioritization methods can provide a reasonable approximation for the theoretical **Best** result for some software subjects, and for other subjects provide some of the smallest gaps. In all cases that we studied, it provided better than the unordered or random ordered test reports.*

5.3 Discussion

Method Selection. The idea of prioritization is widely used in software engineering, especially in software testing. Crowdsourced testing is usually conducted in rapidly iterative software development. In this situation, we can only inspect a subset of test reports for revealing and fixing faults before software release. Hence, test report prioritization plays a key role for a cost-effective result of crowdsourced testing. Our prioritization methods contain two key parts: the risk strategy (**Risk**) and the diversity strategy (**Div**). In software development, we need to reveal as many faults as possible, *i.e.,* **Div**. In contrast, we need to inspect the most probable "true failure" test reports early, *i.e.,* **Risk**. These two requirements of crowdsourced testing drive us to design a hybrid prioritization method **DivRisk** by combining **Risk** and **Div**. Therefore, it is not surprising that **DivRisk** can outperform the random prioritization technique significantly.

Mobile Application Testing. **DivRisk** shows different effectiveness in different crowdsourcing projects. The P1 project involves mobile application testing. The effectiveness of **DivRisk** in P1 was very encouraging and approximated **Best**. We reviewed the test reports in P1 and discussed with testers in Baidu. Since workers used different mobile phones and different versions of Android, they reported many compatibility problems of the application under test. The compatibility problems were easier to identified than other problems for mobile applications. Moreover, part-time workers (crowd workers here) preferred to select testing tasks of mobile applications, because it could be done anywhere and any time. Therefore, it is not surprising that the prioritization results of P1, as shown in Table 6, were more effective than on P2 and P3. Workers committed more test reports and revealed more faults on P1 than on P2 and P3. The percentage of useful test reports (*i.e.,* F-report) is 67.88%, which was better than P2 (20.35%) and P3 (24.60%). The high quality test reports can help our test report prioritization methods, because our methods rely on keywords from test reports. As such, such crowdsourced testing and prioritization methods may be a good fit for mobile application testing.

Cost and Scalability. The total cost of test report processing in our projects is less than 10 minutes. Please note that our prioritization algorithms only involve numerical calculation on KV, RV and DM. Hence, the cost of test report prioritization methods may be negligible. The **DivRisk** algorithm is flexible. For example, we can set $\delta = 0$ in Algorithm 2, and as a result, the dynamic prioritization strategy are reduced to a static prioritization strategy. The static prioritization strategy does not rely on inspection. Hence, it can be fully automated and be more efficient, although the results may be worse. Moreover, **DivRisk** does not rely on the languages of test reports. **DivRisk** can also be used for test reports written in other languages by using other NLP tools for other natural languages. For example, we can adopt Stanford CoreNLP[7] for word segmentation [19] and WordNet[8] for synonym replacement [26] to process English test reports, and build keyword vector model KV. Based on KV, we can use the **DivRisk** algorithm for English test report prioritization.

5.4 Threats to Validity

There are some general threats to validity in our empirical study. For example, we need more projects and different parameter values to reduce the threat to validity.

Subject selection bias. These three crowdsourced testing projects are from industry. The software products are widely used on the Internet, and were not especially designed for our study. Due to the limited cost, we only required the industry partner to provide crowdsourced testing tasks that could be finished in 5 days. The cost of conducting our empirical study was very expensive (involving more than 200 people), so we have only three projects in our empirical study. This may threaten the generalization of our conclusions. However, the software products used in our crowdsourced testing projects are diverse. This may reduce the threat to some extent.

Crowd worker relation. "Crowdsourcing" often requires workers from a large pool of individuals that one has no direct relationship with the others [21]. In our experiment, the students play the roles of crowd worker [7], which means

[7]http://nlp.stanford.edu/software/corenlp.shtml
[8]http://wordnet.princeton.edu/wordnet/

our crowd workers have certain social relations, and we have only nearly 230 crowd workers. The results may be different if the crowd workers are from Internet with open calls. However, Salman *et al.* [36] found that that if a technique is new to both students and professionals, similar performance can be expected to be observed. As such, we believe that this may not be a key point for our test report prioritization techniques.

Data quality. The materials of crowdsourced testing are prepared and distributed by the industrial testers. All test reports are committed by workers online directly. We checked all data and participated in the discussions of the final inspection results. In summary, all data used in this paper are from industry and the results were checked carefully by professional testers of Baidu. This may reduce the threat to the validity of data quality.

6. RELATED WORK

In this section, we discuss three areas related to our work: test case prioritization, failure report classification, and crowdsourced testing.

Test Case Prioritization. Test case prioritization has been intensively studied in the past decades [41]. We only discuss some test case prioritization techniques using distance. W. Dickinson *et al.* [10] studied cluster filtering techniques and proposed an adaptive sampling strategy to select all tests in a cluster when a failed test is inspected. Yan *et al.* [40] proposed ESBS, inspired by the intuitions of fault localization, that uses spectra information in clustered test selection. Jiang *et al.* [18] studied test case prioritization in regression testing and proposed a new family of coverage-based adaptive random testing techniques to replace traditional random testing. Fang *et al.* [14] introduced ordered sequences of program entities to improve the effectiveness of test case prioritization and proposed several novel similarity-based test case prioritization techniques based on the edit distances of ordered sequences. Ledru *et al.* [22] proposed techniques to prioritize test cases based on the text of test data rather than code or specifications and provided empirical results on different distances. Yoo *et al.* [42] proposed a cluster-based test case prioritization technique incorporating expert knowledge to reduce the cost of pair-wise comparisons.

Test case prioritization techniques rely on surrogates for fault detection (such as statement coverage), and hope that satisfying these surrogates earlier will lead to an increasing fault detection rate [15, 32]. Most of test case prioritization techniques use execution profiles, whereas our prioritization techniques use test reports in natural language. Test case prioritization increases fault detection rate by executing test cases. Test report prioritization increases fault detection rate by inspecting test reports.

Failure Report Classification. Failure report classification is also related to test report prioritization. Runeson *et al.* [35] investigated using NLP techniques to support the identification of duplicated failure reports. They took the words in the failure reports in plain English, processed the text, and then used the statistics on the occurrences of the words to identify similar reports. Lo *et al.* [24] addressed software reliability issues by proposing a method to classify software behaviors based on past history or runs. Bowring *et al.* [2] classified program behavior using execution data. Dhaliwal *et al.* [9] reduced the bug fixing time by using the stack traces and runtime information to group the crash reports triggered in different usage scenarios. Wang *et al.* [38] proposed a technique combining natural language and execution information to detect duplicate failure reports.

These techniques take some information (whether natural language or runtime information) about failures and attempt to classify them. Test report prioritization, in contrast, attempts to not only find duplicates, but to order the test cases in a way that facilitates faster inspection by testers.

Crowdsourced Testing. Crowdsourced testing is already popular in industry, and it is a fairly new trend in software engineering research community. Crowdsourcing is the act of taking a job traditionally performed by a designated agent (usually an employee) and outsourcing it to an undefined, generally large group of people in the form of an open call [13, 25]. Liu *et al.* [23] applied crowdsourced testing in usability testing. They studied both methodological differences for crowdsourcing usability testing and empirical contrasts to results from more traditional, face-to-face usability testing. Pastore *et al.* [29] studied whether it is possible to exploit crowdsourcing to solve the oracle problem: generated test input depends on a test that oracle requires human input in one form or another. Dolstra *et al.* [11] used crowdsourced testing to accomplish the expensive tasks on GUI testing. They use virtual machines to run the system under test and enable semi-automated GUI testing by crowd workers. Nebeling *et al.* [28] evaluated the usability of web sites and web-based services with crowdsourcing data, where they showed that crowdsourcing data could provide an efficiently and effective testing method to the web interfaces.

All the studies above use crowdsourced testing to solve some problems in traditional software testing activities. However, we propose test report prioritization methods to solve the problem of crowdsourced testing.

7. CONCLUSIONS

In this paper, we proposed a novel test report prioritization method **DivRisk** to reduce the cost of inspection in crowdsourced testing. The keywords are extracted from test reports by using NLP techniques. These keywords construct a keyword vector model KV. We calculate the risk vector RV based on KV to predict failure risk of tests. We construct the distance matrix DM based on KV to design the diversity strategy for prioritization. The risk strategy and the diversity strategy are combined to a hybrid strategy **DivRisk** to fulfill effective test report prioritization. Three crowdsourced testing projects from industry have been used to evaluate the effectiveness of test report prioritization methods. The results of empirical study encourage us to use **DivRisk** for test report prioritization in practice, especially for mobile application testing. We also provide guidelines to extend our prioritization methods to deal with test reports written in other languages.

Acknowledgments. This work is partially supported by the National Basic Research Program of China (973 Program 2014CB340702), the National Natural Science Foundation of China (61170067, 61373013), and the National Science Foundation under awards CAREER CCF-1350837 and CCF-1116943. The authors would like to thank the testers in Baidu for their great efforts in supporting the three crowdsourced testing projects.

8. REFERENCES

[1] P. Awasthi, D. Rao, and B. Ravindran. Part of speech tagging and chunking with hmm and crf. *Proceedings of NLP Association of India Machine Learning Contest*, 2006.

[2] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. *ACM SIGSOFT Software Engineering Notes*, 29(4):195–205, 2004.

[3] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4):467–479, 1992.

[4] N. Chen and S. Kim. Puzzle-based automatic testing: bringing humans into the loop by solving puzzles. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 140–149. ACM, 2012.

[5] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[6] W. Chen, Z. Li, and T. Liu. Ltp: A chinese language technology platform. In *Proceedings of the 23rd International Conference on Computational Linguistics: Demonstrations*, pages 13–16. Association for Computational Linguistics, 2010.

[7] Z. Chen and B. Luo. Quasi-crowdsourcing testing for educational projects. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion, pages 272–275. ACM, 2014.

[8] Z. Chen, J. Zhang, and B. Luo. Teaching software testing methods based on diversity principles. In *Proceedings of the 24th IEEE-CS Conference on Software Engineering Education and Training*, pages 391–395. IEEE Computer Society, 2011.

[9] T. Dhaliwal, F. Khomh, and Y. Zou. Classifying field crash reports for fixing bugs: A case study of mozilla firefox. In *Proceeding of the 2011 IEEE International Conference on Software Maintenance*, pages 333–342. IEEE, 2011.

[10] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. 26(5):246–255, 2001.

[11] E. Dolstra, R. Vliegendhart, and J. Pouwelse. Crowdsourcing GUI tests. In *Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation*, pages 332–341. IEEE, 2013.

[12] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *Software Engineering, IEEE Transactions on*, 28(2):159–182, 2002.

[13] E. Estellés-Arolas and F. González-Ladrón-de Guevara. Towards an integrated crowdsourcing definition. *Journal of Information science*, 38(2):189–200, 2012.

[14] C. Fang, Z. Chen, K. Wu, and Z. Zhao. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal*, 22(2):335–361, 2014.

[15] C. Fang, Z. Chen, and B. Xu. Comparing logic coverage criteria on test case prioritization. *SCIENCE CHINA Information Sciences*, 55(12):2826–2840, 2012.

[16] S. Foo and H. Li. Chinese word segmentation and its effect on information retrieval. *Information processing & management*, 40(1):161–190, 2004.

[17] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):6, 2013.

[18] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse. Adaptive random test case prioritization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244. IEEE, 2009.

[19] D. Jurafsky and H. James. *Speech and language processing an introduction to natural language processing, computational linguistics, and speech*. Pearson Education, 2000.

[20] A. Kao and S. R. Poteet. *Natural language processing and text mining*. Springer, 2007.

[21] M. Lease and E. Yilmaz. Crowdsourcing for information retrieval. In *ACM SIGIR Forum*, volume 45, pages 66–75. ACM, 2012.

[22] Y. Ledru, A. Petrenko, and S. Boroday. Using string distances for test case prioritisation. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 510–514. IEEE, 2009.

[23] D. Liu, R. G. Bias, M. Lease, and R. Kuipers. Crowdsourcing for usability testing. *American Society for Information Science and Technology*, 49(1):1–10, 2012.

[24] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 557–566. ACM, 2009.

[25] K. Mao, Y. Yang, M. Li, and M. Harman. Pricing crowdsourcing-based software development tasks. In *Proceedings of the 35th International Conference on Software Engineering*, pages 1205–1208, 2013.

[26] G. A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[27] D. Mondal, H. Hemmati, and S. Durocher. Exploring test suite diversification and code coverage in multi-objective test case selection. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015.

[28] M. Nebeling, M. Speicher, M. Grossniklaus, and M. C. Norrie. *Crowdsourced web site evaluation with crowdstudy*. Springer, 2012.

[29] F. Pastore, L. Mariani, and G. Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *Proceedings of the IEEE 6th International Conference on Software Testing, Verificationand Validation*, pages 342–351. IEEE, 2013.

[30] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, pages 465–475. IEEE, 2003.

[31] A.-M. Popescu and O. Etzioni. Extracting product features and opinions from reviews. In *Natural language processing and text mining*, pages 9–28. Springer, 2007.

[32] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the 1998 International Conference on Software Maintenance*, pages 34–43. IEEE, 1998.

[33] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: an empirical study. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188, Aug 1999.

[34] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[35] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering*, pages 499–510. IEEE, 2007.

[36] I. Salman, A. T. Misirli, and N. Juristo. Are students representatives of professionals in software engineering experiments? In *Proceedings of the 37th International Conference on Software Engineering*. ACM, 2015.

[37] Y.-H. Tung and S.-S. Tseng. A novel approach to collaborative testing in a crowdsourcing environment. *Journal of Systems and Software*, 86(8):2143–2153, 2013.

[38] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*, pages 461–470. ACM, 2008.

[39] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 264–274, Nov 1997.

[40] S. Yan, Z. Chen, Z. Zhao, C. Zhang, and Y. Zhou. A dynamic test cluster sampling strategy by leveraging execution spectra information. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 147–154. IEEE, 2010.

[41] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[42] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 201–212. ACM, 2009.

[43] K. Zhang, H. Xu, J. Tang, and J. Li. Keyword extraction using support vector machine. In *Advances in Web-Age Information Management*, pages 85–96. Springer, 2006.